ELSEVIER

The Journal of Systems and Software 61 (2002) 145–169

The Journal of Systems and Software

www.elsevier.com/locate/jss

Software requirements validation via task analysis

Hong Zhu [a,*], Lingzi Jin [b], Dan Diaper [c], Ganghong Bai [d]

[a] School of Computing and Mathematical Sciences, Oxford Brookes University, Wheatley Campus, Oxford OX33 1HX, UK
[b] AIT Ltd., The Malthouse, 45 New Street, Henley-on-Thames, Oxfordshire RG9 2BP, UK
[c] School of Design, Engineering and Computing, Bournemouth University, Poole, Dorset, BH12 5BB, UK
[d] Bell Labs, Lucent Technologies, 9th Floor, Hao Ran High Tech. Bldg, 1954 SJTU, Hua Shan Rd., Shanghai 200030, China

Received 25 April 2000; received in revised form 15 March 2001; accepted 5 May 2001

Abstract

As a baseline for software development, a correct and complete requirements definition is one foundation of software quality. Previously, a novel approach to static testing of software requirements was proposed in which requirements definitions are tested on a set of task scenarios by examining software behaviour in each scenario described by an activity list. Such descriptions of software behaviour can be generated automatically from requirements models. This paper investigates various testing methods for selecting test scenarios. Data flow, state transition and entity testing methods are studied. A variety of test adequacy criteria and their combinations are formally defined and the subsume relations between the criteria are proved. Empirical studies of the testing methods and the construction of a prototype testing tool are reported.

Keywords: Software requirements; Requirements definition; Data flow diagram; State transition diagram; Entity relation diagram; Software testing; Task analysis; Activity list; Test adequacy criteria; Subsume relation; Combination of test criteria; Automatic test case generation

1. Motivations

Software requirements definitions are the documents of users' requirements of computer systems and serve as the baseline for the development of the software (Stokes, 1991). Many studies have shown that errors in requirement definitions are very costly, even impossible, to rectify at later stages of software development (Boehm, 1981). Therefore, the validation of requirements definitions is of vital importance to software development.

As Goodenough and Gerhart (1975) pointed out, the central problem of software testing is the adequacy problem, that is, what constitutes adequate testing. Over the past several decades a great number of test adequacy criteria have been proposed and investigated as rules that guide the selection of test cases and as measures for objective specification of the testing requirements and measurement of testing quality. Existing software testing methods can be classified into two types: static testing and dynamic testing. Dynamic testing involves the execution of program code so that the dynamic behaviour of the system can be observed and analysed. Dynamic software testing methods can be further classified into specification-based methods and program-based methods. A specification-based testing method derives test cases from the requirements specification such as from algebraic specifications (Bouge et al., 1986; Beront et al., 1991; Chen et al., 2000, 2001), Z specifications (Amla and Ammann, 1992; Stocks and Carrington, 1993; Ammann and Offut, 1994), finite state machines (Fujiwara et al., 1991), Petri nets (Pezze and Young, 1996; Morasca and Pezze, 1990), and from design specifications such as from UML (Offutt and Abdurazik, 1999; Abdurazik and Offut, 2000) and from software architectural descriptions (Richardson and Wolf, 1996; Bertolino et al., 2000; Rosenblum, 1997). A program-based testing method selects test cases according to the information contained in the program. Among the

* Corresponding author. Tel.: +44-1865-484580; fax: +44-1865-484545.
E-mail addresses: hzhu@brookes.ac.uk (H. Zhu), lingzi.jin@ait.co.uk (L. Jin), ddiaper@bournemouth.ac.uk (D. Diaper), whitebai@lucent.com (G. Bai).

0164-1212/02/$ - see front matter
PII: S0164-1212(01)00109-1

most well-known testing methods are *structural testing*, which include control flow testing methods such as statement coverage, branch coverage and path testing (Howden, 1976; Hetzel, 1984), data flow testing methods such as definition/use path coverage methods (Laski and Korel, 1983; Ntafos, 1984; Rapps and Weyuker, 1985; Frankl and Weyuker, 1988), and dependence coverage methods (Podgurski and Clarke, 1989, 1990); *functional testing*, which include domain partition testing methods such as boundary value analysis (Myers, 1979; Clarke et al., 1982; Richardson and Clarke, 1985; Afifi et al., 1992) and functional analysis (Howden, 1987a); and *fault-based testing*, which include mutation testing (DeMillo et al., 1978; Budd, 1980), weak mutation testing (Howden, 1987b), perturbation testing (Zeil, 1983, 1989), and the methods based on Richardson and Thompson's (1993) RELAY model. Readers are referred to (Zhu et al., 1997) for a survey of research on software testing methods. Such dynamic testing methods are not directly applicable to the validation of requirements definitions at the requirements stage, however, unless an executable prototype of the system is available. Prototyping involves the development and demonstration of prototype software to test if the software satisfies users' requirements. It provides visualised demonstrations of a system's behaviour, which enables users/customers to examine whether the functions and behaviour are as required. It is effective for validating the user interface and certain specific features of the system, such as timing and scheduling aspects in real-time applications, see e.g. (Luqi et al., 1998). It is costly to develop prototypes, however, that contain sufficiently accurate detail related to requirements as the amount of detail that the prototype can demonstrate determines the maximum level of testing strictness.

Static testing is concerned with the review of software documents and readable high-level code. It is applicable to requirements definitions. Typical static testing methods include checklist guided inspection (Wheele, 1996; Gild and Gramham, 1993) and structured walkthroughs in formal review (Yourdon, 1989b). They have been widely used in practice and are claimed to be effective at detecting errors (Wheele, 1996; Yourdon, 1989b). Static testing is by its nature a labour intensive process and for the following reasons it is difficult. First, in formal reviews and inspections of requirements definitions, the human testers are required to understand the notations used in the requirements definition. Since there are no tested documents against which the correctness of the requirements definition can be checked, it is a common practice that users and/or customers are involved in requirements validation (Kamsties et al., 1998). It is difficult, however, for users to understand the technical notations used in requirements definitions. Second, during the testing process, the tester has to translate in their mind the static description of the requirements into the dynamic behaviour of the described system. Such translations often have to coordinate information scattered over the requirements definition documents, especially when the system is described by multiple views (Yourdon, 1989a; Nuseibeh et al., 1994). Such translations are further complicated due to the difficulty of casting two-dimensional diagrams into one-dimensional temporal sequences of events. Not only is this process time consuming and expensive, but its effectiveness is dependent on the experience and training of the human tester (Basili and Selby, 1987). Moreover, there are few software tools available to support the static testing of software requirements definitions, although the need for such tools is widely recognised (Macdonald et al., 1995, 1996). Finally, there are few theoretical and systematic methods reported in the literature that address the static testing adequacy problem. In dynamic testing, test adequacy criteria enable software testers to specify the goal of testing before a testing process starts and to measure and control the strictness of the testing process objectively (Ould and Unwin, 1986; Weyuker, 1986; Parrish and Zweben, 1993; Zhu and Hall, 1993). No such criteria have been proposed in the literature for the validation of requirements by static testing or prototyping.

This paper is concerned with static testing of software requirements and aims at improving its efficiency and effectiveness by systematically applying well-defined test adequacy criteria and deploying automated software testing tools.

Our approach is based on the notion of scenarios or use cases (Carroll, 2000). Generally speaking, a scenario represents a set of situations that might occur during the operation of a software system. These situations have some common characteristics in the types of users involved in the operation of the system and in the purposes and conditions of use. In recent years scenario analysis has attracted researchers in requirements engineering. Its potential in requirements validation has been recognised by a number of researchers, e.g. (Anderson and Durley, 1993; Jin and Zhu, 1998; Zhu and Jin, 2000; Haumer et al., 1998). The basic idea is to check requirements in various scenarios that may occur during software operation. One of the main advantages of scenario directed requirements validation is that the process of requirements validation can be naturally divided into a number of testing tasks of manageable size and complexity. Since each task only concentrates on one typical situation in the operation of the specified software, testing requirements on one scenario is independent of the testing on other scenarios. This facilitates the application and management of the testing process. Another potential advantage of scenario directed requirements validation is that it provides a potential method of controlling test strictness. The set of scenarios used in requirements testing can be large or small in size and can also be low or high in complexity. Testing requirements on more, and more complicated, scenarios will usually result in higher levels of strictness and be better at detecting errors. Of course, it also becomes

more expensive and time consuming. Thus, the strictness and cost of requirements testing can be controlled by the selection of an appropriate set of scenarios.

There are two approaches to scenario directed requirements validation. The first approach is to describe the required behaviour of the software on each scenario independently of the requirements definition being tested. Such descriptions are then checked against the requirements definition under test for their consistency. Techniques have been developed to check automatically the consistency between scenario description and requirements models for this purpose (Jin and Zhu, 1998). The second approach is to derive the description of the behaviour of the specified software from the requirements definition. Such descriptions are then checked by the tester for their correctness. The second approach is often preferred because checking the correctness of a description is usually easier than creating a description from scratch. This is the approach that this paper uses.

For both approaches, there are many important questions to be answered, for example: how scenarios should be described so that testing can be performed effectively and efficiently; where such descriptions of scenarios come from or how to derive such descriptions; and how to select scenarios.

Attempting to answer the first two questions above, Zhu et al. (1999a,b) proposed activity lists as a format for describing software behaviour and they advanced a method of automatic generation of such descriptions from requirements definitions. A set of transformation rules and algorithms were proposed and a prototype tool was constructed for the automatic generation of activity lists from requirements definitions when given a set of task scenarios. Activity lists are inspired by task analysis techniques (Diaper, 1989) developed in Human–Computer Interaction (HCI). Task analysis has been previously used for both analysis and design during software development activities, e.g. (O'Neill et al., 1999). In this paper, however, while as in HCI an activity list is a prose description of a linear sequence of events in temporal order, they here describe what happens primarily inside the computer system for a particular scenario. Each event, an activity, is described by one sentence in structured natural language that combines all the relevant information originally scattered over the requirements definition documents in various diagrams and dictionaries. Information irrelevant to the scenario is filtered out. It is this feature of activity lists that differs from existing representations of scenarios, such as diagrams (Jin and Zhu, 1998; Jacobson et al., 1992) and formal notations (Hsia et al., 1994). It thus offers a more testable description of system's behaviour so that formal review should be performed more effectively and efficiently.

This paper further investigates how to select and generate testing scenarios. Obviously, reuse of the scenarios obtained during requirements acquisition is not an ideal solution because validation should be independent of the production of requirements models. The paper explores various objective approaches to the selection of test scenarios by defining the selection criteria and studying the relationships between them and the methods of combining them.

The paper is organised as follows. Section 2 defines the basic notions and notations used in this paper, which include the syntax of activity lists and diagrammatic notations of requirements models. Section 3 describes a requirements definition that is then used as an example throughout the paper. Section 4 investigates the methods of generating or selecting test scenarios. Data flow testing, state transition testing and entity testing methods are discussed. A variety of test adequacy criteria are formally defined. Section 5 is a theoretical study of the relationship between the testing methods. Subsume relations between adequacy criteria are proved. Methods for the combination of adequacy criteria are discussed. Section 6 briefly reports on the implementation of a prototype test tool and an empirical study of the testing methods. Section 7 concludes the paper with a discussion of the advantages and some remaining problems of the proposed approach, related work and directions for future research.

2. Basic notions and notations

This section defines the ideas that the theory and method developed in this paper are based on and the notations used. It defines what are the software artefacts under test, what is a test case, and what is a test criterion.

2.1. Requirements definitions

Requirements definitions are the software artefacts of testing that are addressed by the theory and methods described in this paper. In particular, the paper is concerned with the validation of requirements definitions as a correct documentation of users' requirements. For the sake of practicality, the testing theory and methods will be based on a standard method of structured analysis (Yourdon, 1989a). The particular syntax of the diagrams used is from the requirements definition language NDRDL (Xu et al., 1995), and is shown in Fig. 1.

A requirements definition is a combination of natural language text, Entity Relationship Diagrams (ERDs), Data Flow Diagrams (DFDs), State Transition Diagrams (STDs) and their associated dictionaries.

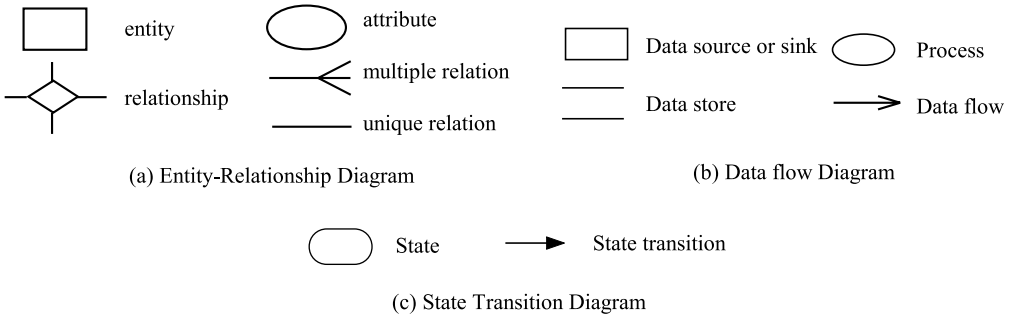

Fig. 1. Diagram notations.

An ERD provides a data model for a required system by specifying the entities of the system, their attributes and their relationships with other entities. A DFD describes the external sources of data, the processing of data within the system and the external receivers of the processed data. A STD is a directed graph with a start node and at least one exit node. A node in the graph represents a state, which must be on a path from the start node to an exit node. In a STD, each arc is associated with an event. A path from the start node to the exit node represents a sequence of events that can happen in the operation of the software. Only such sequences of events are allowed to happen. The events associated with the arcs in a STD can either be an invocation of a process or a predicate describing a condition for state transition.

Dictionaries contain more detailed information concerning the contents of the diagrammatic models. There are three dictionaries in a NDRDL requirements definition: data dictionary, process dictionary and relationship dictionary. The data dictionary defines the structure and usage of data in a required system. An entry in a data dictionary consists of four fields: (a) the name of the data to be defined, (b) an informal description of the data in natural language, (c) a formal description of the data that gives the type, hence the structure, of the data, and (d) a constraint on the value of the data. The data's type is represented in a manner similar to a data type in a programming language. The constraint is in the form of a first-order predicate. The relationship dictionary defines the relationship between entities. An entry in a relationship dictionary contains four fields: (a) the name of the relationship, (b) the entities involved in the relationship, (c) an informal description of the relationship in natural language, and (d) a formal definition of the relationship in the form of a predicate. The process dictionary provides definitions of processes. An entry in a process dictionary consists of five fields: (a) the name of the process to be defined, (b) the input parameters of the process, (c) the output of the process, (d) an informal description of the function of the process, and (e) a formal definition of the process in the form of a predicate that relates the output to the input with references to data stores and defines the update to data stores. It should be noted that the data flows into or from a data store are not considered as input or output of processes because data stores are considered as globally accessible and represent a system's internal state.

Consistency and completeness constraints are imposed on the models and dictionaries by the NDRDL language so that different views provide a consistent and internally complete requirements definition. Table 1 summarises these constraints; see also (Xu et al., 1995).

The consistency and completeness constraints given in Table 1 can be automatically checked (Jin and Zhu, 1997). In this paper, it is assumed that the requirements definitions under test satisfy these constraints.

Table 1
Completeness and consistency constraints of NDRDL

Views	Constraints
DFD/ERD	The collection of data in a DFD must be the same as the collection of data represented as the entities or their attributes in the corresponding ERD
STD/DFD	The set of processes associated with the arcs in a STD must be the same as the set of processes in the corresponding DFD Any sequence of events in a STD (i.e. a path in the STD from the start node) must satisfy the permissible condition, i.e. all the input data of an event must be generated by the earlier events in the sequence so that they are available to execute the event
STD/ERD	Any data used in a STD must be contained in the collection of data in the ERD
ERD/DD	Every entity in an ERD must be defined in the data dictionary. The attributes of an entity in an ERD must be consistent with the definition of attributes of the entity in the data dictionary
ERD/RD	Every relationship in an ERD must be defined in the relationship dictionary with the same participant entities
DFD/PD	Every process in a DFD must be defined in the operation dictionary such that the signature of the operation must be consistent with the data flowing inwards to, and outwards from, the process node

2.2. Activity lists as descriptions of system behaviour

The testing methods studied in this paper are structured testing methods of test adequacy that are determined by observations of a putative system's behaviour made during the testing process. The formal definition of adequacy criteria is therefore dependent on the representation of system behaviour provided by the activity lists.

Activity lists are inspired by task analysis techniques, which have been developed by the ergonomics, human factors and HCI communities over the past three decades, e.g. (Annett and Duncan, 1967). Task analysis techniques are collectively known as "methods of collecting, classifying and interpreting data on human performance in work situations ... (that) reflect(s) both our current understanding of human performance and the design of systems that best serve the needs of human users" (Annett and Stanton, 1998, p. 1529). There are many advantages to the task centred view of the world and task analysis has the ability to represent a wide variety of types of data from many sources (Johnson et al., 1984). Existing HCI models of task analysis generally ignore events internal to software systems, thus they are not directly applicable to the analysis of software behaviour.

In general, tasks are the means by which a *worksystem* which includes both people and tools, such as computers, effect changes on the real world (Long, 1989, 1997). Thus the worksystem model of tasks is that it is not people or computers alone that carry out tasks but their collaboration that achieves changes in the world. Diaper et al. (1998) Pentanalysis Technique is a recently developed task analysis method that, while grounded in HCI, is explicitly designed to support systems analysts reasoning about software design. It extends the worksystem task model that treats users, components of the system and other agents in the environment equally and tasks are considered as interactions between all kinds of agents. It is this task model that enables task analysis derived techniques to be applied to the validation of software requirements.

The most common input to HCI task analysis methods is a "task protocol" or "activity list" (Diaper, 1989). An activity list consists of a prose description of a task which, for convenience, is described as a series of task steps. Usually, each task step contains one action, carried out by an agent on one or more objects. For example, the following is a typical task step.

`The user types the command on the keyboard.`

What is novel about the use of task analysis in the proposed method in this paper is that the actions, agents and objects model the dynamic behaviour of a computer system. In particular, the paper is concerned with: (a) how information is exchanged between components of the software as well as between components and the agents in its environment; (b) what sequence of computations are performed by the components; (c) how the software reacts to stimuli from its environment; and (d) how internal states of the software are updated and affect the behaviour of the system. Such an analysis involves the generation of scenarios that are then used to examine the correctness of the requirements definition. A scenario is a detailed description of system behaviour in the form of an activity list.

An activity list consists of a list of activities that have a structure defined in Extended Bachus–Naur Form (EBNF) as follows:

⟨Activity⟩ ::= ⟨Agent⟩ ⟨Action⟩ ⟨Object⟩ [⟨Receiver agent⟩][⟨Constraint on the object⟩].

For requirements definitions in structured analysis, there are two types of agents: (a) the components of the software system which can be either a process or a data store in the DFD, and (b) the agents in a software's environment, which are terminators in a DFD, i.e. external data sources and receivers. Each agent in the activity list is represented by its name modified with a keyword to indicate its type:

⟨Agent⟩ ::= *process* ⟨Process name⟩ |*terminator* ⟨Terminator name⟩|*data store* ⟨data store name⟩

There are five types of activities used in the description of a system's behaviour. These are: (a) receiving information from another agent; (b) sending information to another agent; (c) obtaining information from the internal state of the software; (d) updating the internal state of the software; (e) performing computation by a component of the software. Hence:

⟨Action⟩ ::= *sends*|*receives*|*updates*|*obtains*|*performs*

The object of an action can either be data or a computation. In addition to the name of the data, more information about its type is also provided in the activity list. Similarly, for computational activities, any additional information about the computation available from the requirements definition is also provided in the activity list, which may include detailed informal descriptions and formal definitions of the computation.

⟨Object⟩ ::= data ⟨Data name⟩ of type ⟨Type name⟩
|computation ⟨Informal description⟩ or ⟨Formal definition⟩

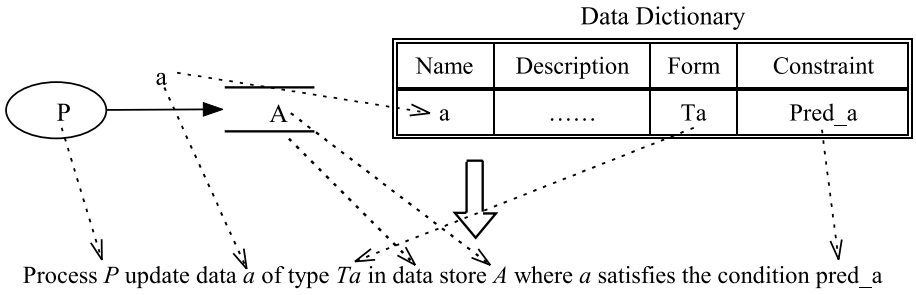

Fig. 2. Generating the activity of updating data in a data store.

For example, the following sentence is an activity.

$$\underbrace{\text{Process P}}_{agent}\ \underbrace{\text{sends}}_{action}\ \underbrace{\text{data } x \text{ of type real}}_{object}\ \underbrace{\text{to the terminator display}}_{receiver}\ \underbrace{\text{where } x > 0}_{constraint}.$$

The generation of an activity involves gathering related information from various models and dictionaries and placing the information in a structured natural language sentence. The following figure gives one of the rules for the generation of different types of activities and illustrates how information scattered over various parts of a requirements definition are put together to form an item in an activity list. Details of the generation process and a complete set of rules can be found in Zhu et al. (1999b) (see Fig. 2).

2.3. Test cases and adequacy criteria

As argued in Zhu et al. (1999a), a path in a STD from the start node to an exit node can be considered as a task scenario, i.e. as a set of situations with common characteristics that may occur in the operation of the system. The events associated with the arcs are the sub-tasks performed by the software in a scenario. The state transition conditions, which are the predicates associated with the arcs, are the characteristic conditions of the scenario. Conversely, since the system is only allowed to perform activities as specified by the STD, any task is an instance of a path from the start node to an exit node. Such a scenario is considered as one test case for the static testing of a requirements definition. Test cases are abstract in the sense that they may contain variables that represent any arbitrary concrete data or objects as sets satisfying some constraints. Concrete test cases are obtained by substituting actual values for the variables in scenario descriptions. A distinction between concrete and abstract test cases is unnecessary for static testing, however, as it makes no difference to the formal definitions of test criteria.

Test criteria are rules about what should be tested and they play a central role in software testing methods (Goodenough and Gerhart, 1975). A test criterion can appear in a number of equivalent forms (Zhu et al., 1997). For example, it can be in the form of a test data selection criterion, which is a rule for the selection of test cases. It can also be in the form of a test data adequacy criterion, which is a rule to determine whether a set of test cases is adequate for testing a piece of software. In this paper, test criteria are defined in the form of adequacy criteria. Formally, a test adequacy criterion for testing software requirements can be defined as follows.

Let RE be the set of requirements definitions that consists of a DFD, a STD, an ERD and their three associated dictionaries. It is assumed that the diagrams and dictionaries satisfy the consistency and completeness constraints given in Table 1. Let STD be the state transition diagram of a given requirements definition. We define $Path(STD)$ to be the set of all paths from the start node to an exit node in the state transition diagram STD and $T = 2^{Path(STD)}$. The variable r, lower case letter in italic font, denotes a requirements definition and ranges over the set RE. The variable t, lower case letter in italic font, denotes a set of test cases and ranges over T. A test adequacy criterion is a predicate C defined on RE and the set T of test suites, i.e., $C : RE \times T \to \{true, false\}$. The expression $C(r, t) = true$ means the test set t is adequate for testing r according to adequacy criterion C; otherwise t is inadequate.

3. An example

This section provides an example that will be used throughout the remainder of the paper to illustrate the testing methods.

3.1. The requirements definition

A simplified bank account management system provides the following functions:

- Enquiry about the balance of an account;
- Update the balance of an account when a deposit to the account is made;
- Update the balance of an account when a withdrawal is made.

Figs. 3–5 give the STD, ERD and DFD models of the system, respectively. The dictionaries are in Appendix A.

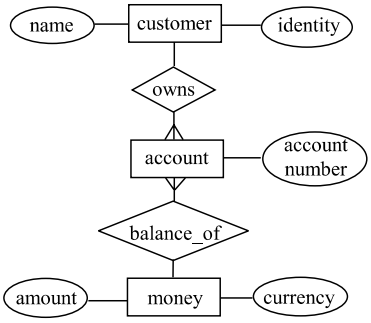

Fig. 3. ERD of the bank system.

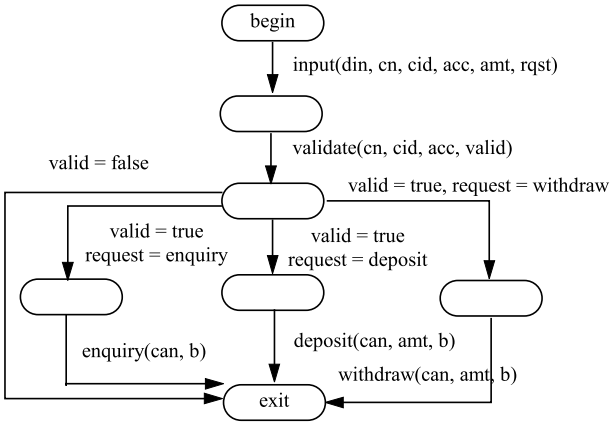

Fig. 4. STD of the bank system.

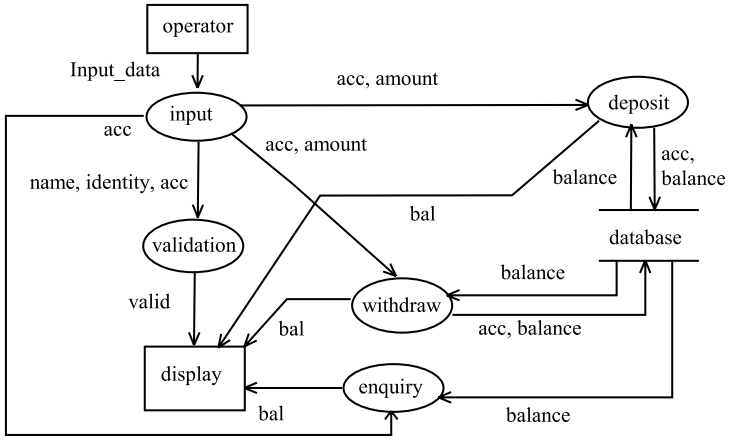

Fig. 5. DFD of the bank system.

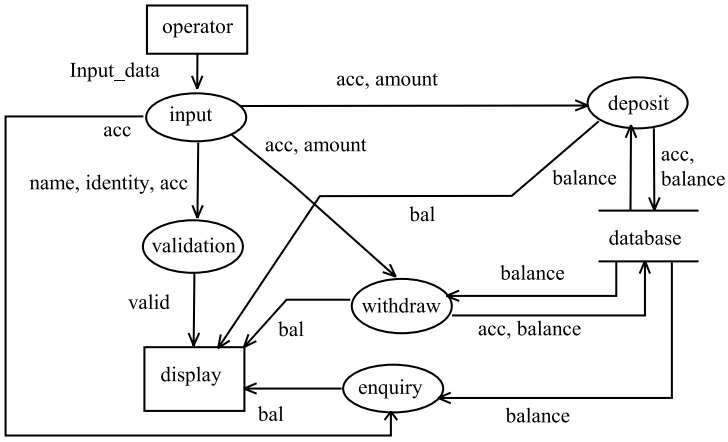

Fig. 5. DFD of the bank system.

3.2. Scenarios and activity lists

There are four paths in the STD of the Bank system (Fig. 4). Each path corresponds to one of the following scenarios for using the system.

(a) *invalid input*, i.e. a customer tries to access an account but the information provided is invalid;
(b) *withdrawal from an account*, i.e. a customer requested a withdrawal from an account with valid input data;
(c) *deposit into an account*, i.e. a customer requested a deposit into an account with valid input data;
(d) *enquiring about an account*, i.e. a customer requests an enquiry about an account with valid input.

The following are the paths in the Bank system STD that correspond to these task scenarios.

Scenario of invalid input:
{begin}
input(din, cn, cid, acn, amt, rqst);
validate(cn, cid, acn, valid);
In this case, valid = false;
{exit}

Scenario of withdraw:
{begin}
input(din, cn, cid, acc, amt, rqst);
validate(cn, cid, acc, valid);
In this case, valid = true, request = withdraw;
withdraw(acc, amt, bal);
{exit}

Scenario of enquiry:
{begin}
input(din, cn, cid, acc, amt, rqst);
validate(cn, cid, acc, valid);
In this case, valid = true, request = enquiry;
enquiry(acc, bal);
{exit}

Scenario of deposit:
{begin}
input(din, cn, cid, acc, amt, rqst);
validate(cn, cid, acc, valid);
In this case, valid = true, request = deposit;
deposit(acc, amt, bal);
{exit}

The following are the activity lists for the invalid input scenario. Activity lists for other scenarios are given in Appendix B.

Activity list for the invalid input scenario:
{begin}
Process *input* receives data *din* of type *INPUT_DATA* from terminator *operator*;
Process *input* performs computation *input*, or formally, "*and(cn = din.name, cid = din.iden, acn = din.account, amt = din. amount, rqst = din.request)*";
Process *input* sends data *cn* of type *string* to process *validate*;
Process *input* sends data *cid* of type *integer* to process *validate*;
Process *input* sends data *acn*_of type integer to process *validate*;
Process *validate* receives data *cn* of type *string* from the process *input*;
Process *validate* receives data *cid*_of type *integer* from the process *input*;
Process *validate* receives data *acn* of type *integer* from process *input*;

Process *validate* performs "validate if *cn* of *cid* owns the account *acn* according to the information stored in the database", or formally, "$\exists r \in$ database. ((acn = r.account) $\wedge$ (cid = r.customer.identity) $\wedge$ (cn = r.customer.name)) $\Rightarrow$ valid = true;$\neg\exists$ r$\in$ database. ((acn = r.account) $\wedge$ (cid = r.customer.identity) $\wedge$ (cn = r.customer.name)) $\Rightarrow$ valid = false";
Process *validate* sends data *valid* of type bool to terminator *display terminal*;
In this case, *valid* = *false*;
{exit}

4. Testing methods

This section explores various ways of selecting test scenarios. Three types of testing methods will be discussed and formally defined: data flow testing; state transition testing; and entity testing.

4.1. Data flow testing

Data flow testing methods test software requirements according to the information contained in the data flow model and tests focus on the data flow between the processes, terminators and data stores. A modest requirement of adequate data flow testing is perhaps to test each process in the data flow diagram at least once. This is formally defined below as a test adequacy criterion called *process coverage*.

Definition 1 (*Process coverage criterion*). A test set t is said to be adequate according to the *process coverage criterion* with respect to r's DFD, if for each process P in DFD, there is at least one path p in t such that p contains an arc to which an invocation of the process P is associated.

For example, the path of the invalid input scenario of the Bank system covers processes *input* and *validate*. It does not cover the processes *deposit*, *withdraw*, nor *enquiry*. The set of paths consisting of withdrawal, deposit and enquiry scenarios covers all the processes in the Bank system DFD. Hence, the set of paths is adequate according to the process coverage criterion.

Similarly, we require a test of software requirements to cover all terminators and data stores in the data flow model of software requirements. The adequacy criteria of *terminator coverage* and *data store coverage* are defined as follows.

Definition 2 (*Terminator coverage criterion and data store coverage criterion*). A test set t is said to be adequate according to the *terminator coverage* criterion with respect to r's DFD, if for each terminator M in DFD, there is at least one path p in t such that the activity list of p contains an activity that receives data from, or sends data to, the terminator M.

A test set t is said to be adequate according to the *data store coverage* criterion, if for each data store R, there is at least one path p in t such that the activity list of p contains at least one activity that obtains data from R or updates data in R.

For example, the test path of the invalid input scenario of the Bank system covers the terminators "operator" and "display". It is adequate according to the *terminator coverage* criterion with respect to the Bank system's DFD. It does not cover, however, the data store "database".

A data flow x from process A to process B in r's DFD is covered by a path p in r's STD, if the activity list for the path p contains the activities "Process A sending data x of type Tx to process B" and "Process B receiving data x of type Tx from process A".

For example, the data flow from input process to validate process in the data flow diagram of the Bank system is covered by the path above. The data flow from input process to enquiry is not covered by this path, however, because the activity list only contains the activity connected with the process of sending input data to the enquiry process, i.e. it does not contain an activity involving the enquiry process receiving the data.

A data flow x from a terminator M to process A is covered by a path p, if the activity list of p contains an activity that process A receives the data x from terminator M. Similarly defined, a data flow to a terminator or a data flow from/to a data store is covered by a path according to the contents in the activity list of the path.

Definition 3 (*Definition*, *use*, *input*, *output and parameter coverage criteria*). A set t of test cases is said to be adequate according to the *definition coverage*, if for all data flow d that flow into a data store, there is at least one path p in t such that p covers d. The set t is said to be adequate according to the *use coverage*, if for all data flows d that flow out from a

data store, there is at least one path p in t such that p covers d. The set t is said to be adequate according to the *input coverage* criterion, if for all data flows d that flow from a terminator, there is at least one path p in t such that p covers d. The set t is said to be adequate according to the *output coverage* criterion, if for all data flows d that flow into a terminator, there is at least one test case p in t such that p covers d. The set t is adequate according to the *parameter coverage* criterion, if for all data flows d that flow into a process, there is at least one path p in t such that p covers d.

The *data flow coverage* criterion requires a test suite that covers all data flows in the data flow diagram. Formally, this has the following definition.

Definition 4 (*Data flow coverage criterion*). A test set t is said to be adequate according to the *data flow coverage* criterion with respect to r's DFD, if for all data flows d in DFD, there is at least one path p in t such that p covers d.

A sequence $q = \langle n_1, n_2, \ldots, n_k \rangle$ of nodes in a DFD is called a *path of data flow* if there is a data flow from node n_i to n_{i+1}, for all $i = 1, 2, \ldots, k-1$, in the DFD. A path of data flow represents a process of transferring data from one node to another. To test such propagation of information within a system, a test set is required to cover such a path of data flow. Let data $x_{i,1}, x_{i,2}, \ldots, x_{i,k}$ be associated to the data flow from n_i to n_{i+1}, $Tx_{i,j}$ be the type of $x_{i,j}$ according to the data dictionary.

A path p in a STD is said to cover the path q of data flow, if p' s activity list contains a sub-sequence of activities $A_1, \ldots, A_2, \ldots, A_{k-1}$ such that:

1. if the data flow from n_i to n_{i+1} is from a process node P to a process node Q, then A_i consists of the following activities:
   Process P performs computation ...;
   Process P sends output data $x_{i,1}$ of type $Tx_{i,1}$ to Process Q;
   Process P sends output data $x_{i,2}$ of type $Tx_{i,2}$ to Process Q;
   ...
   Process P sends output data $x_{i,k}$ of type $Tx_{i,k}$ to Process Q;
   Process Q receives data $x_{i,1}$ of type $Tx_{i,2}$ from Process P;
   Process Q receives data $x_{i,2}$ of type $Tx_{i,2}$ from Process P;
   ...
   Process Q receives data $x_{i,k}$ of type $Tx_{i,k}$ from Process P;
   Process Q performs computation ...;
2. if the data flow from n_i to n_{i+1} is from a process node P to a terminator node M, then A_i consists of the following activities:
   Process P performs computation ...;
   Process P sends output data $x_{i,1}$ of type $Tx_{i,1}$ to terminator M;
   Process P sends output data $x_{i,2}$ of type $Tx_{i,2}$ to terminator M;
   ...
   Process P sends output data $x_{i,k}$ of type $Tx_{i,k}$ to terminator M;
3. if the data flow from n_i to n_{i+1} is from a process node P to a data store node R, then A_i consists of the following activities:
   Process P performs computation ...;
   Process P updates data $x_{i,1}$ of type $Tx_{i,1}$ in data store R;
   Process P updates data $x_{i,2}$ of type $Tx_{i,2}$ in data store R;
   ...
   Process P updates data $x_{i,k}$ of type $Tx_{i,k}$ in data store R;
4. if the data flow from n_i to n_{i+1} is from a terminator node M to a process node P, then A_i consists of the following activities:
   Process P receives input data $x_{i,1}$ of type $Tx_{i,1}$ from terminator M;
   Process P receives input data $x_{i,2}$ of type $Tx_{i,2}$ from terminator M;
   ...
   Process P receives input data $x_{i,k}$ of type $Tx_{i,k}$ from terminator M;
   Process P performs computation ...;
5. if the data flow from n_i to n_{i+1} is from a data store node R to a process node P, then A_i consists of the following activities:
   Process P obtains data $x_{i,1}$ of type $Tx_{i,1}$ from data store R;
   Process P obtains data $x_{i,2}$ of type $Tx_{i,2}$ from data store R;
   ...

Process P obtains data $x_{i,k}$ of type $Tx_{i,k}$ from data store R;
Process P performs computation ….

For example, the path of the invalid input scenario covers the path of data flow ⟨terminator, input, validate, display⟩ in the Bank system DFD.

A path of data flow is said to be *infeasible*, if there is no path in the state transition diagram that covers the path of data flow. In other words, in the operation of the system such data transfer processes cannot happen. Therefore, such a path of data flow cannot be tested. Subsequently, the discussion of data flow testing only concerns feasible paths of data flow.

An *elementary data flow path* is a path of data flow on which there is no node that occurs more than once. A *simple data flow path* is a path of data flow on which there is no arc that occurs more than once.

Definition 5 (*Elementary data flow path coverage and simple data flow coverage criteria*). A test set t is said to be adequate according to the *elementary data flow path coverage* criterion, if for all elementary data flow paths q in the DFD, there is at least one path p in t such that p covers q. The test set t is said to be adequate according to the *simple data flow path coverage* criterion, if for all simple data flow paths q in the DFD, there is at least one path p in t such that p covers q.

Elementary data flow paths and simple data flow paths are the most simple data transfer processes. Elementary data flow paths do not contain any *cycles*, which are sub-paths whose start nodes and end nodes are the same. Although simple data flow paths do not contain any arc more than once, they may contain cycles. However, any cycle can appear only once in a simple path. The *cycle once data flow path coverage* criterion requires covering all cycles at least once.

Definition 6 (*Cycle once data flow path coverage criterion*). A test set t is said to be adequate according to the *cycle once data flow path coverage* criterion, if for all data flow paths q in which no cycle occurs more than once, there is at least one path p in t such that p covers q.

Adequacy criteria that require testing cycles repeatedly can be defined. However, given that the testing is to be performed by human analysts, such repeated cycles are inefficient and generally unnecessary. The strongest data flow testing method is the *all data flow path coverage* criterion. When the data flow diagram contains feasible cycles, it requires an adequate test set to cover a very large or infinite number of data flow paths. Therefore, adequacy according to the *all data flow path coverage* criterion may not be practically achievable.

Definition 7 (*All data flow path coverage criterion*). A test set t is said to be adequate according to the *all data flow path coverage* criterion, if for all data flow paths q in the DFD, there is at least one path p in t such that p covers q.

4.2. State transition testing

State transition testing methods consider whether a test suite adequately tests a STD. State coverage and transition coverage criteria are examples of state transition testing methods.

The *state coverage* criterion requires that a test suite should cover all states in the STD. The *transition coverage* criterion requires that the test paths should cover all state transitions in the STD. They are formally defined as follows.

Definition 8 (*State coverage criterion and transition coverage criterion*). A test set t is said to be adequate according to the *state coverage* criterion, if for all states in the STD, there is at least one path p in t such that p contains the state node. A test set t is said to be adequate according to the *transition coverage* criterion, if for all state transitions there is at least one path p in t such that p contains the state transition.

For example, the set of paths consisting of enquiry, deposit and withdrawal scenarios is adequate according to the state coverage criterion, but not adequate according to the transition coverage criterion. The set of four scenarios given in Section 3.2 is adequate according to both state coverage and transition coverage criteria.

A sequence $\langle s_1, s_2, \ldots, s_k \rangle$ of states in a STD is called a *path of control flow*, if there is an arc from state s_i to s_{i+1} for all $i = 1, 2, \ldots, k$ in the STD. The node s_1 and s_k are called the path's *start node* and *end node*, respectively. A path is called a *cycle*, if its start node and end node are the same. A path of control flow is called a *simple path*, if there is no arc occurring more than once in the path, and it starts with the *begin* node and ends with an *exit* node. A path of control flow is called an *elementary path*, if there is no node that occurs more than once in the path and it starts with the *begin* node and ends with an *exit* node. Obviously, an elementary path is a simple path. For example, the paths given in

Section 3.2 are elementary paths. A path p in the STD *covers* a path $q = \langle s_1, s_2, \ldots, s_k \rangle$ of control flow if the $\langle s_1, s_2, \ldots, s_k \rangle$ is a sub-sequence of nodes on the path p.

Definition 9 (*Elementary and simple control flow path coverage criteria*). A test set t is said to be adequate according to the *elementary control flow path coverage* criterion, if for all elementary paths q of control flow in the STD, there is at least one path p in t such that p covers q.

A set t is said to be adequate according to the *simple control flow path coverage* criterion, if for all simple paths q of control flow, there is at least one path p in t such that p covers q.

Elementary and simple paths correspond to the simplest task scenarios which do not contain loops and repeated activities. To test more complicated scenarios, a test set may be required to cover all paths in a STD, including those containing cycles of activities.

Definition 10 (*All control flow path coverage criterion*). A set t is said to be adequate according to the *all control flow path coverage* criterion, if for all paths q of a control flow that starts with the *begin* node and ends with an *exit* node, there is at least one path p in t such that p covers q.

For example, the Bank system STD only contains four paths of control flow. Therefore, the set of the paths is adequate according to all the *control flow path coverage* criterion.

Since a STD that contains cycles may have an infinite number of paths, *all control flow path coverage* testing may be too inefficient to achieve adequacy because it is impractical to cover all paths. In fact, repeating a cycle more than once in testing software requirements is sometimes unnecessary. Hence, we have the following cycle once criterion.

Definition 11 (*Cycle once control flow path coverage criterion*). A set t is said to be adequate according to the *cycle once control flow path coverage* criterion, if for each path q of control flow that starts with the *begin* node, ends with an *exit* node of the diagram and contains no cycle more than once, there is at least one path p in t such that covers q.

When a STD contains a large number of simple and elementary cycles, a great number of test cases may be required to satisfy the *cycle once control flow path coverage* criterion. A weaker criterion is given below, which is inspired in McCabe's cyclomatic test criterion for program testing (McCabe, 1983).

Definition 12 (*Complete set of independent paths*). A set t is said to be adequate according to the *complete set of independent paths* criterion, if t covers a complete set of linear independent paths of the STD.

Note that, for a state transition diagram of n nodes, a arcs and e exit nodes, a complete set of independent paths contains $a - n + e + 1$ paths. For example, there are 9 arcs, 7 states and 1 exit node in the STD of the Bank system. Hence, according to the formula, there are 4 independent control flow paths. In fact, the four paths given in Section 3.2 constitute a complete set of independent paths. Therefore, it is adequate according to the complete set of independent paths criterion.

4.3. Entity testing

Entity testing methods focus on the testing of the entity relationship model of a requirements definition. The adequacy of testing is decided by whether the entities and their attributes are checked thoroughly. An attribute is said to be covered by a path p, if the activity list of p contains data on the attribute. An entity is said to be covered by a path p, if p covers at least one of the attributes of the entity.

Definition 13 (*Attribute coverage criterion*). A set t of test cases is said to be adequate according to the *attribute coverage* criterion with respect to an ERD, if for all attributes b of each entity e in the ERD, there is at least one p in t such that p covers b.

An adequacy criterion weaker than attribute coverage is the *entity coverage* criterion.

Definition 14 (*Entity coverage criterion*). A set t is said to be adequate according to the *entity coverage* criterion with respect to an ERD, if for all entities e of the *ERD*, there is at least one path p in t such that p covers e.

For example, the path of the invalid input scenario only involves the entity customer and account and not the entity money. Hence, it is adequate according to neither the entity coverage criterion, nor the attribute coverage criterion. The path of the deposit scenario involves all the attributes of all the entities in the ERD. Hence, it is adequate according to both criteria.

5. Analysis and combinations of the testing methods

This section analyzes the relationships between the testing methods developed in the Section 4 and derives new adequacy criteria by combining existing ones.

5.1. The subsume relation

Generally, a test data adequacy criterion A subsumes criterion B, if for all testing, the adequacy of a test set according to A implies its adequacy according to B. Here, the software artefact under test is a requirements definition. The subsume relation is formally defined as follows.

Definition 15 (*Subsume relation*). Let A, B be two adequacy criteria. A is said to subsume B, if for all requirements definitions r and all test sets t for testing r, t satisfies criterion A for testing r implies that t also satisfies B for testing r. Formally, $\forall r \in RE.\forall t \in T.(A(r,t) \Rightarrow B(r,t))$.

A basic property of the subsume relation is transitivity, i.e. if A subsumes B and B subsumes C, then A also subsumes C.

The subsume relation between adequacy criteria is essentially a comparison of the strictness of testing methods. In general, while the subsume relation does not guarantee a better fault detecting ability (Frankl and Weyuker, 1993), it has been proved that under certain conditions the subsume relation does mean better fault detection. For example, when test adequacy criteria are only used to decide if a set of test cases is adequate, but not used to generate test cases (this is called the *posterior testing scenario* (Zhu, 1996)), criterion A subsumes criterion B implies that testing method A has a higher probability of detecting at least one fault and a greater expected number of errors found by testing.

The following theorem gives a subsume relation between data flow testing methods.

Theorem 1. *The data flow coverage criterion subsumes the data node coverage criterion*

Proof. By the definition of *data flow coverage* criterion, if a test set t is adequate according to the *data flow coverage* criterion, it covers all data flows in the DFD. This means it covers all arcs in the DFD. Since a DFD must be a connected graph, a node in a DFD must be on at least one arc of the diagram. Therefore, the test set t must have covered all the nodes in the diagram. Hence, t is also adequate according to the *data node coverage* criterion. □

The following theorem gives a subsume relation between entity testing methods.

Theorem 2. *The attribute coverage criterion subsumes the entity coverage criterion.*

Proof. Let a test set t be adequate according to the *attribute coverage* criterion. By Definition 13, the test set t covers all attributes in ERD because every entity must have at least one attribute in an ERD, so t must also have covered all entities in ERD. Therefore, t is also adequate according to the *entity coverage* criterion. □

The following two theorems give two subsume relations between state transition testing methods.

Theorem 3. *The cycle once control flow path coverage criterion subsumes the transition coverage criterion.*

Proof. Let test set t be adequate according to the *cycle once control flow path coverage* criterion. By Definition 11, for all paths q in STD that contain no cycle more than once there is a path p in t such that p covers q. Let x be any arc in STD. Because, in a STD, an arc x must be on a path q_x from the start node to an exit node that does not contain a cycle more than once, there must be a path p that covers the path q_x. Therefore, the set t must also cover all arcs in STD, hence t is adequate according to *transition coverage* criterion. □

Theorem 4. *The transition coverage criterion subsumes the state coverage criterion.*

Proof. Let t be adequate according to the *transition coverage* criterion. By Definition 8, for all state transitions s there is at least one path p in t such that p covers s. Let n be any state in STD. Since in a STD, a node must be on a path from the start node to an exit node, every node in a STD must be associated to at least one state transition. Let s_n be a transition that state n is associated with. The adequacy of t according to the *transition coverage* criterion implies that there is a path p in t such that p covers s_n. Therefore, by Definition 8, t is also adequate according to the *state coverage* criterion. □

Subsume relations also hold between testing methods of different types.

Theorem 5. *The transition coverage criterion subsumes the process coverage.*

Proof. Let t be adequate according to the *transition coverage* criterion. By Definition 8, for all transitions s in STD, there is a path p in t such that p covers s. Let P be a process in DFD. The consistency and completeness constraints imposed on DFDs and STDs requires that every process in a DFD should appear at least once in the STD. Let s_P be the transition in STD that contains P. The adequacy of t according to the *transition coverage* criterion implies that there is a path p in t such that p covers s_P. By Definition 1, the path p covers the process P. Therefore, the test set t is also adequate according to *process coverage* criterion. □

Theorem 6. *The data flow coverage criterion subsumes the attribute coverage criterion.*

Proof. The consistency and completeness constraints imposed on DFDs and ERDs requires that the collection of data in the DFD should be the same as the collection of data represented as the entities and their attributes in the ERD. Therefore, if a test set covers all the data flows in a DFD, it must also have covered all the attributes in the ERD. □

Theorem 7. *The all control flow path coverage criterion subsumes the all data flow path coverage criterion.*

Proof. By Definition 7, the *all data flow path coverage* criterion only requires that feasible data flow paths are covered by testing. Let q be any feasible data flow path. By definition, the feasibility of a data flow path implies that there is a control flow path p such that p covers q. If a test set t is adequate according to the *all control flow path coverage* criterion, then by Definition 10, there is at least one path in t that covers p. This means q is covered by t. Therefore, t is also adequate according to *all data flow path coverage* criterion. □

Notice that, although a simple control flow path always covers at least one elementary data flow path, a test set that satisfies the *elementary data flow path coverage* criterion does not necessarily satisfy the *simple control flow adequacy* criterion, because two different simple control flow paths may cover the same elementary data flow path. However, if in a STD any two different simple control flow paths always have different sequences of events associate to the arcs, then, a test set that satisfies the *elementary data flow path coverage* criterion must satisfy the *simple control flow paths coverage* criterion.

More subsume relations will be given in the next subsection.

5.2. Combinations of test criteria

Adequacy criteria can be obtained by combining existing ones. This section discusses such combination methods and uses them to derive new adequacy criteria to those discussed in the previous section and proves that some of the criteria can be defined as combinations of others. By doing so, more subsume relations between adequacy criteria are obtained.

5.2.1. Conjunction

One of the most useful combination methods is the conjunction of two or more adequacy criteria.

Definition 16 (*Conjunctive combination of test criteria*). Let A, B be two test adequacy criteria. Criterion C is the conjunction of A and B, if a test set t is adequate if and only if t satisfies both A and B.

For example, the *data node coverage* criterion can be defined as the conjunction of the *process coverage*, *terminator coverage* and *data store coverage* criteria. The *data flow coverage* criterion is the conjunction of the *definition coverage* criterion, *use coverage* criterion, *input coverage* criterion, *output coverage* criterion, and *parameter coverage* criterion.

The conjunction of two adequacy criteria A and B is stronger than the original criteria in the sense of the subsume relation between adequacy criteria.

Lemma 1. *Let the adequacy criterion C be the conjunction of adequacy criteria A and B. Then, C subsumes A, and C subsumes B.*

Proof. Let C be the conjunction of A and B. By Definition 16, a test set t that satisfies C must also satisfy A. This means C subsumes A. Similarly, C subsumes B. □

By Lemma 1, we have the following theorem about the subsume relations between adequacy criteria.

Theorem 8.
1. *Data node coverage criterion subsumes data store coverage, process coverage, and terminator coverage criteria;*
2. *Data flow coverage criterion subsumes definition coverage, use coverage, input coverage, output coverage, and parameter coverage.*

Proof. By the definitions of data flow coverage and data node coverage and Lemma 1. □

5.2.2. Disjunction

Another useful combination of adequacy criteria is the disjunction of criteria. Before giving a formal definition of the notion, an example is presented. Considering the *data store coverage* criterion, it is defined such that a test set is adequate according to the criterion if all data stores in a DFD are covered by the test set, where "a data store is covered if either a data flow into the data store is covered or a data flow from the data store is covered". Such a definition consists of two parts, the first defines what need to be covered and the second defines what it means by the word "cover". Such a criterion is called a coverage criterion. Formally, let criterion A be defined by the expression that "a test set t satisfies A if for all x in S_A (r), there is a test case p in t such that $P_A(x,p)$", i.e. $A(r,t) \Longleftrightarrow \forall x \in S_A(r).\exists p \in t.(P_A(x,p))$, where $P_A(x,p)$ is a predicate, which means that the test case p covers the structure x. Such a criterion is called a coverage criterion, and S_A and P_A are called the *structure to be covered* and the *covering method*. For the data store coverage criterion, the structure to be covered is the set of data stores in the DFD. The covering method is that "either a data flow into the data store is covered or a data flow from the data store is covered".

Now considering the *definition coverage* and *use coverage* criteria, obviously they are coverage criteria, too. What is more important is that there is a relationship between the *data store coverage* criterion and these two criteria. First, there is a relationship between the structures to be covered. Each data flow to be covered by the *definition coverage* criterion is associated with a data store, which is an element of the structure to be covered by the *data store coverage* criterion. A similar relationship exists between the structures to be covered by the *data store coverage* and *use coverage* criteria. Second, there is also a relationship between the covering methods. The definition of the covering method in the *data store coverage* criterion is directly based on the covering methods of the *definition coverage* and *use coverage* criteria. The covering of a data store requires the existence of an associated element in either of the structures to be covered by the *definition coverage* criterion or the *use coverage* criterion be covered in the sense of the covering methods by these two criteria. Such a relationship between *data store coverage* criterion and the *definition* and *use coverage* criteria is called a disjunction and can be formally defined as follows.

Definition 17 (*Disjunctive combination of test criteria*). Let A and B be coverage criteria, S_A and S_B are the structures to be covered for A and B, and P_A and P_B be the covering methods for A and B, respectively. An adequacy criterion C is the disjunction of A and B, if C is a structure coverage criterion with S_C as the structure to be covered and P_C as the covering method such that

1. $\forall x \in S_C.(\exists a \in S_A.F_A(a,x) \vee \exists b \in S_B.F_B(b,x))$, and
2. for all path p, $\forall x \in S_C.(P_C(x,p) \Longleftrightarrow \exists a \in S_A.F_A(a,x) \wedge P_A(a,p) \vee \exists b \in S_B.F_B(b,x) \wedge P_B(b,p))$,

where $F_A(a,x)$ and $F_B(b,x)$ are predicates which define the relationship between elements in S_C and elements in S_A and S_B, respectively.

Definition 17 states that, firstly, for all elements x to be covered according to criterion C, there is either an element a in S_A such that a is associated with x, or there is an element b in S_B such that b is associated with x. Secondly, an

element x in S_C is considered as having been covered by a test case p according to C, if either the element a in S_A associated with x is covered according to criterion A, or the element b in S_B associated with x is covered according to criterion B. Approximately, each element in S_C may have two components that constitute S_A and S_B. Criterion A requires adequate test sets covering all components in S_A, while criterion B requires covering all components in S_B. The disjunction of A and B only requires an adequate test set covering one of the components for each element in S_C, but it does not matter which component is covered.

For example, *data store coverage* is a disjunction of the *definition coverage* and *use coverage* criteria. Here, the structure to be covered by *definition coverage* contains all the data flows that flow into a data store. The structure to be covered by *use coverage* contains all the data flows that flow from a data store. The covering method for *data store coverage* is that "a data store is covered if either a data flow into the data store is covered or a data flow from the data store is covered". Similarly, *terminator coverage* is a disjunction of *input coverage* and *output coverage*.

The disjunction of two adequacy criteria is not necessarily subsumed by the original ones. Thus, the disjunction of criteria is weaker than the conjunction of the original ones.

Lemma 2. *Let adequacy criteria A and B be two coverage criterion, adequacy criterion C being a disjunction of A and B. Then the conjunction of A and B subsumes C.*

Proof. Let A and B be coverage criteria, S_A and S_B be the structures to be covered for A and B, and P_A and P_B be the covering methods for A and B, respectively. Let C' be the conjunction of A and B. By Definition 16, $C'(r,t) \Longleftrightarrow A(r,t) \wedge B(r,t)$. By the definition of coverage criteria, we have that

$$A(r,t) \Longleftrightarrow \forall x \in S_A(r).\exists p \in t.(P_A(x,p)), \quad \text{and} \quad B(r,t) \Longleftrightarrow \forall x \in S_B(r).\exists p \in t.(P_B(x,p))$$

Thus,

$$C'(r,t) \Longleftrightarrow (\forall x \in S_A(r).\exists p \in t.(P_A(x,p)) \wedge \forall x \in S_B(r).\exists p \in t.(P_B(x,p))). \tag{$*$}$$

Let $x' \in S_C$. By Definition 17, $\exists a \in S_A.F_A(a,x') \vee \exists b \in S_B.F_B(b,x')$. By (*) and Definition 17, we have that

$$\begin{aligned} C'(r,t) &\Rightarrow \forall x' \in S_C.(\exists a \in S_A.F_A(a,x') \wedge \exists p \in t.P_A(a,p) \vee \exists b \in S_B.F_B(b,x') \wedge \exists p \in t.P_B(b,p)) \\ &\Rightarrow \forall x' \in S_C.\exists p \in t.P_C(x',p) \Rightarrow C(r,t). \quad \square \end{aligned}$$

Theorem 9.
1. *The conjunction of the definition coverage criterion and the use coverage criterion subsumes the data store coverage criterion.*
2. *The conjunction of the input coverage and output coverage criteria subsumes the terminator coverage criterion.*

Proof. By Lemma 2. $\square$

5.2.3. Reduction

Reduction is a widely used method to produce a new test criterion from an existing one for structure coverage testing in the study of adequacy criteria for testing. The new criterion reduces the strength of testing by requiring adequate test sets covering fewer elements. Formally, reduction is defined as follows.

Definition 18 (*Reduction of test criterion*). Let A be a structure coverage criterion with S_A as the structure to be covered and P_A as the covering method. An adequacy criterion B is called a reduction of A, if its covering methods is the same as P_A, but its structure to be covered is a subset of S_A, i.e. $S_B \subseteq S_A$.

The reduction of a test criterion generates a criterion weaker than the original.

Lemma 3. *If B is a reduction of adequacy criterion A, than A subsumes B.*

Proof. By the definition of coverage criteria and Definition 18. $\square$

The following theorem gives the reduction relationship between data flow testing methods.

Theorem 10.
1. *The all data flow path coverage criterion subsumes the cycle once data flow path coverage criterion.*
2. *The cycle once data flow path coverage criterion subsumes the simple data flow path coverage criterion.*

3. *The simple data flow path coverage criterion subsumes the elementary data flow path coverage criterion.*
4. *The simple data flow path coverage criterion subsumes the data flow coverage criterion.*

Proof. By proving that one is a reduction of the other and then by applying Lemma 3. As discussed in Section 4.1, the set of data flows is a subset of simple data flow paths; the set of elementary data flow paths is a subset of simple data flow paths; the set of simple data flow paths is a subset of cycle once data flow paths; and the set cycle once data flow paths is a subset of all data flow paths. □

The following theorems gives the reduction relationship between state transition testing methods.

Theorem 11.
1. *The all control flow paths coverage criterion subsumes the cycle once control flow path coverage criterion.*
2. *The cycle once control flow path coverage criterion subsumes the simple control flow path coverage criterion.*
3. *The simple control flow path coverage criterion subsumes the elementary control flow path coverage criterion.*

Proof. Similar to the proof of Theorem 10. □

Theorem 12.
1. *The all control flow paths coverage criterion subsumes the complete set of independent paths criterion.*
2. *The complete set of independent paths criterion subsumes the all simple paths coverage criterion.*

Proof. It is easy to prove (1) by applying Lemma 3. The proof of statement (2) is similar to the proof of a theorem in (Jin et al., 1997). Details are omitted for the sake of space. □

A summary of the subsume relationships between adequacy criteria is given in Fig. 6, in which nodes are adequacy criteria and arrows are subsume relations.

6. Tool implementation and empirical studies

A software tool has been constructed to automatically generate a set of task activity lists according to state transition based test criteria, and to measure the adequacy of the test set according a set of data flow adequacy criteria. This requirements testing tool has been integrated into a requirements analysis support system NDRASS (Xu et al., 1995; Jin and Zhu, 1997; Zhu and Jin, 2000). Fig. 7 shows the interface of the tool as a part of the NDRASS system.

The tool shows on the screen the generated test sets, their activity lists and their adequacy measurement, see Fig. 8.

The adequacy of various generated state transition test sets are measured according to data flow test criteria and displayed as a table, see Fig. 9.

A number of case studies have been conducted with the testing tool. Table 2 gives the data collected from these case studies. It shows the numbers of the test cases generated by the tool for each sample system to satisfy various test criteria. It also gives the complexity of the requirements in terms of the number of states, transitions and elementary cycles for each sample system. The results from these case studies shows that the testing method proposed in this paper is practical in the sense that a reasonable number of activity lists are generated. Moreover, the length of these activity lists is also manageable, see Table 3.

7. Conclusion

Based on previous work on the application of activity lists to the description of software behaviour for requirements validation, this paper explores testing methods for selecting test scenarios. Three types of testing methods are investigated. Data flow testing methods select test scenario according to the data flow model of requirements definition. State transition testing methods are based on the state transition model. Entity testing methods are concerned with the entity relationship model. The adequacy criteria for these testing methods fall into an almost hierarchical structure of subsume relations, which include relationships between different types of testing methods provided that the different models satisfy a set of consistency and completeness constraints. Test adequacy criteria for these testing methods are formally defined so that different levels of test strictness can be accurately specified and objectively measured.

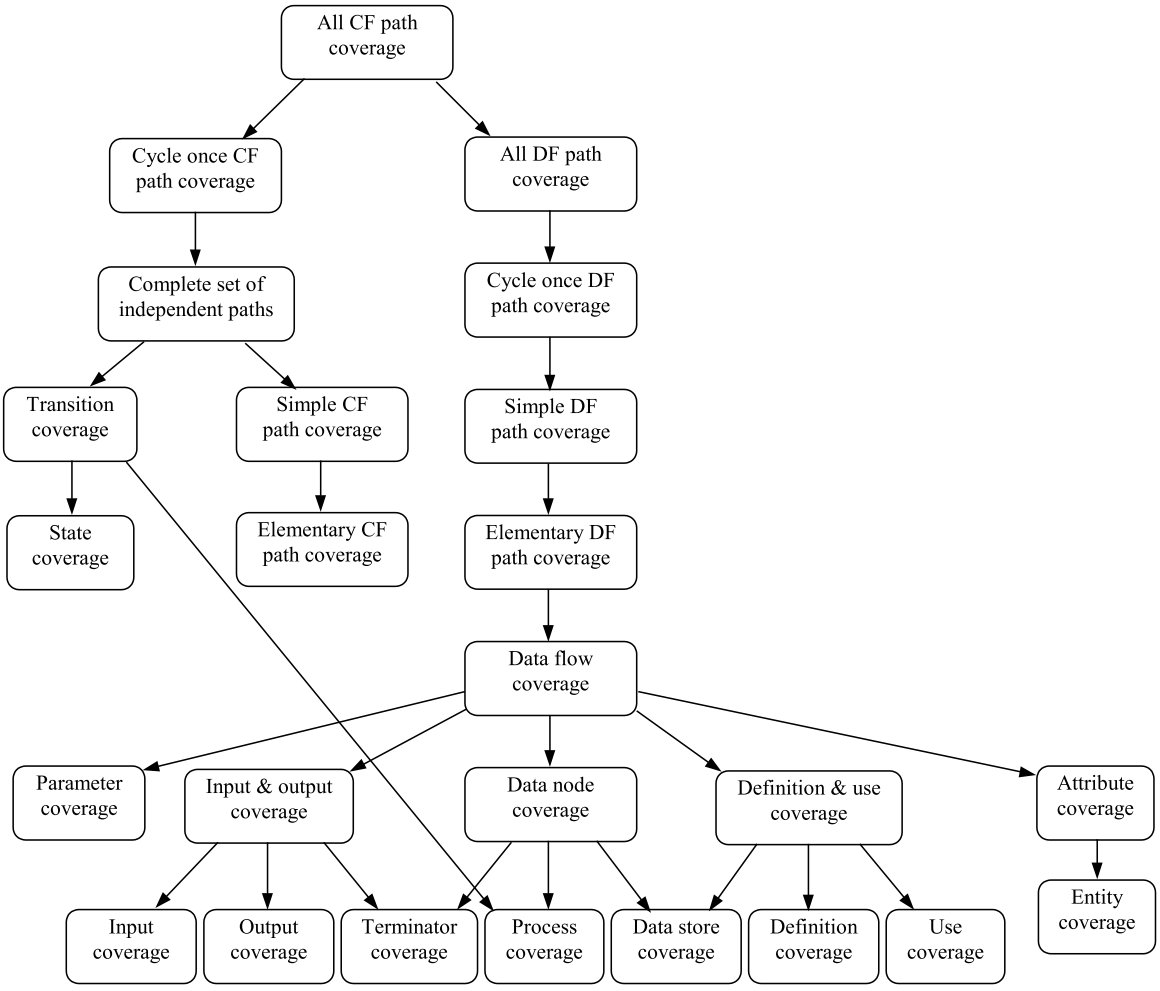

Fig. 6. The subsume hierarchy.

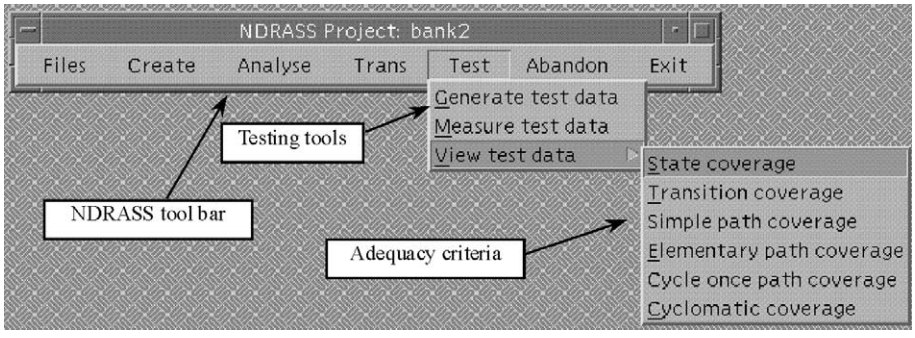

Fig. 7. Integration of the testing tools into NDRASS.

7.1. Advantages of the proposed approach

The proposed approach has the following advantages.

First, as argued in (Zhu et al., 1999a,b), descriptions of software behaviour in the form of activity lists are more readable than diagrams and their associate dictionaries. Each activity list represents a task scenario that may occur in the use of the system. Information related to the scenario originally scattered over various diagrams and dictionaries are put together in one activity list. Information irrelevant to the scenario is filtered out so that they are not presented in the activity list. Two-dimensional diagrams are cast into sequences of events in temporal order. Each event is described in structured natural language. Such a linear description provides all the details of a system's behaviour

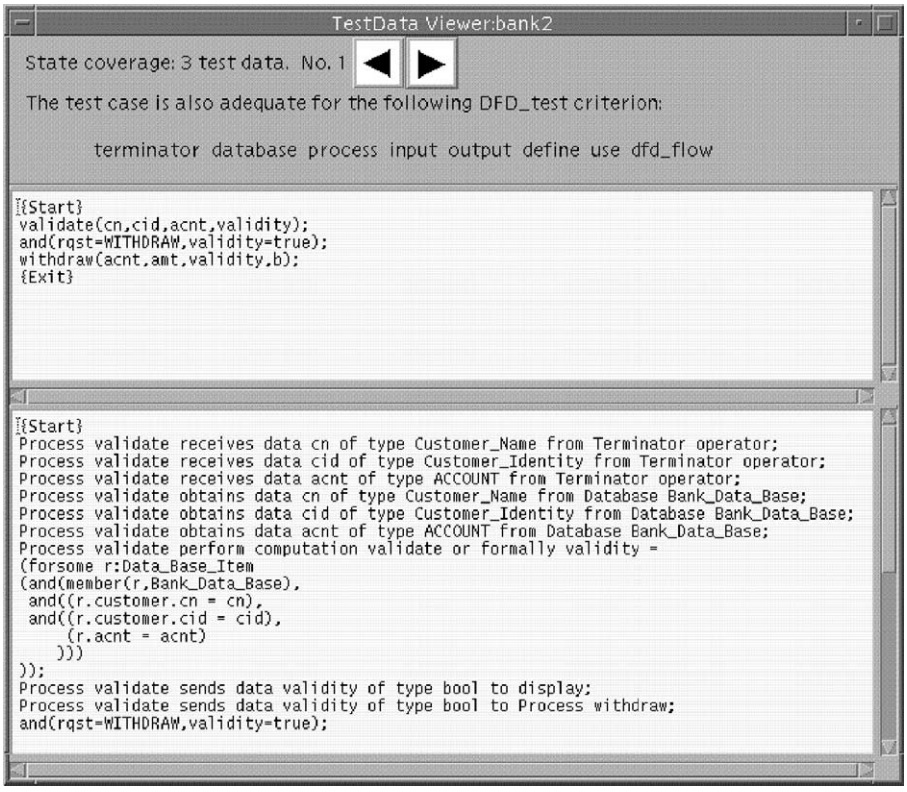

Fig. 8. The display of an activity list on screen.

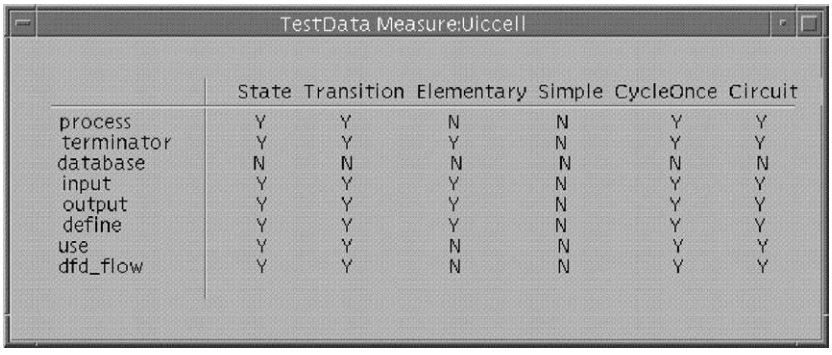

Fig. 9. Measurement of adequacy according to data flow test criteria.

Table 2
Requirements complexity and the numbers of test cases to satisfy test criteria

Sample system	Test criteria					Requirements complexity		
	State coverage	Transition coverage	Elementary path coverage	Simple path coverage	Complete set of independent path	No. of states	No. of transitions	No. of cycles
Wind tunnel	2	2	2	2	2	8	7	0
Bank management	3	4	4	4	4	6	8	0
Elevator controller	4	5	5	5	6	12	16	1
Travel agents	6	6	6	6	7	16	16	1
Book store management	1	1	1	1	4	10	12	3
Gas burner controller	4	6	2	9	14	24	29	6
Mobile phone cellar net	6	10	2	16	26	38	47	11

specified by the requirements definition concerning the information exchanged between components as well as agents in the environment, the computations performed and the uses and updates of internal states. Therefore, system's

Table 3
Average length of activity lists to satisfy test criteria

Sample system	Test criteria				
	State coverage	Transition coverage	Elementary path coverage	Simple path coverage	Complete set of independent path
Elevator controller	6.75	6	5.6	5.6	6.33
Travel agents	6	6	6	6	7
Wind tunnel	10	10	10	10	10
Bank management	17	15	15	15	15
Book store management	54	54	3	3	20
Gas burner controller	28.25	21.17	21.11	17.5	24.67
Mobile phone cellular net	47.5	47.5	18	35.75	41.62

behaviour is much more testable when described in the form of activity lists than in the form of diagrams and dictionaries.

Second, this paper shows that various methods of systematically selecting test scenarios can be formally defined as test adequacy criteria. These methods form a fairly simple, nearly hierarchical structure, and so can be selectively applied to achieve various levels of test strictness. This offers a flexible but systematic approach to testing requirements definitions because test adequacy criteria provide a means of accurately specifying test requirements and objectively measuring test achievement.

Third, software tools, such as the prototype we have developed, can be implemented to automatically generate a set of task scenarios according to user selected adequacy criterion and to generate an activity list for each task scenario. Therefore, the cost of generating activity lists is negligible in comparison with the cost of examining the behaviour of the system. Moreover, as shown by the empirical studies, the number of task scenarios and the lengths of activity lists are acceptable for the practical use of the method.

7.2. Related work and further research

Since Goodenough and Gerhart (1975) pointed out that the central problem of software testing methods is test criteria, a great number of test criteria have been proposed and investigated in the literature for program testing (Zhu et al., 1997), but few adequacy criteria have been proposed for testing software requirements. The data flow testing methods proposed in this paper are inspired in the data flow testing methods for testing programs, especially the work by Rapps and Weyuker (1985), Frankl and Weyuker (1988), Ntafos (1984), and Laski and Korel (1983). The state transition testing methods are similar to program control flow testing methods, such as statement coverage, branch coverage and various path coverage methods (Myers, 1979). Although a number of state transition testing methods can be defined analogously to control flow testing methods, such as level-i path testing (Paige, 1978) and various decision condition coverage criteria (Myers, 1979), how effective such criteria are for testing software requirements is an open issue and warrants further empirical studies and theoretical analysis. The entity testing methods seems less mature than the data flow and state transition testing methods. Further development of such testing methods might focus on how to test relationships specified in ERDs.

Although the proposed methods improves the efficiency of static testing of requirements definitions by providing more testable descriptions of software behaviour, how effective the testing methods are for detecting errors needs further study. Empirical studies of the methods in industrial environments are needed to investigate the error detecting ability of activity lists with various testing methods.

A future research topic will involve adapting or extending the proposed approach for other software design and testing purposes. There are a number of possible directions for such adaptations and extensions. For example, is the method applicable to the testing of software designs presented in other formats such as in the object-oriented models? While the syntax of activity lists and the transformation rules that generate such activity lists seem more suitable for testing functional requirements than non-functional requirements, the question is whether or not they can be modified so that other features, such as performance and real time requirements, can be effectively tested, provided that such non-functional requirements are appropriately represented in requirements definitions. Finally, although the method is inspired by HCI's approach to task analysis, the activity lists generated from the requirements definitions focuses on the internal behaviour of the system and hence existing HCI task analysis techniques cannot be applied to the analysis of the specified system. The question is whether an activity list that contains more information about the user will be closer to that of an HCI task analysis. Of course, this requires that enough information about the user's behaviour should be contained in the requirements definition.

Table 4
The data dictionary of the bank system

Data name	Description	Form	Constraint
Name	The name of a customer	**String**	name ⟨⟩ nil
Identity	The identity number of a customer	**Integer**	
Customer	A customer of a bank branch	**Record** Name: string; Identity: integer **End**	
INPUT_DATA	The structure of input data	**Record** name: string; iden: integer; acc: integer; amount: real; request: REQUEST **End**	
Account number	An account number held in the branch	**Integer**	
REQUEST	Service request, which can be an enquiry, a deposit or a withdraw	**Enum** (enquiry, deposit, withdraw)	
Balance	The amount of money left in an account	**Real**	balance $\geqslant 0$
Money	The amount of money to be deposited to or to be withdrawn from an account	**Record** amount: **Real**; currency: **Enum** ($) **End**	amount $\geqslant 0$
Database	The database used by the branch for storing the information about who is a customer, which number is an account number and who owns an account.	**Set** (**Record** Customer: customer; Account: account number; Balance: money **End**)	
rqst	The request of a customer	**REQUEST**	
input_data	The input data from the operator	**INPUT_DATA**	
acc	Account number	**Integer**	
amt	Amount	**Real**	
din	Input data from operator	**INPUT_DATA**	
cn	Customer name	**String**	cn⟨⟩nil
cid	Customer identity number	**Integer**	
valid	If the personal information is valid, the validation process outputs true as the value of valid, otherwise, it outputs false.	**Bool**	

Table 5
The relationship dictionary of the Bank system

Name	Entities	Description	Definition
Owns	Customer, account	Owns (John, 210093) means that John owns the account 210093	owns(c, ac) $\Longleftrightarrow \exists$r $\in$ database. ((ac = r.account) $\wedge$ (c = r.customer))
Balance-of	Account, money	Balance-of (210093, 20.50) means that the amount of money left in the account 210093 is $20.50	balance_of(ac, b) $\Longleftrightarrow \exists$r $\in$ database. ((ac = r.account) $\wedge$ (b = r.balance))

Appendix A. Bank system requirement definition dictionaries

See Tables 4–6.

Appendix B. Activity lists for the bank system example

Enquiry:
{begin}
Process *input* receives data *din* of type *INPUT_DATA* from terminator *operator*;
Process *input* performs computation *input*, or formally, "*and*(*cn* = *din.name*, *cid* = *din.iden*, *acn* = *din.account*, *amt* = *din*. *amount*, *rqst* = *din.request*)";
Process *input* sends data *cn* of type *string*to process *validate*;
Process *input* sends data *cid* of type *integer* to process *validate*;
Process *input* sends data *acn*of type integer to process *validate*;

Process *input* sends data *acc* of type integer to process *enquiry*.
Process *validate* receives data *cn* of type *string* from the process *input*;
Process *validate* receives data *cid* of type *integer* from the process *input*;
Process *validate* receives data *acn* of type *integer* from process *input*;
Process *validate* performs "validate if *cn* of *cid* owns the account *acn* according to the information stored in the database", or formally, "$\exists r \in database.((acn = r.account) \wedge (cid = r.customer.identity) \wedge (cn = r.customer.name)) \Rightarrow valid = true; \neg\exists r \in database.((acn = r.account) \wedge (cid = r.customer.identity) \wedge (cn = r.customer.name)) \Rightarrow valid = false$";
Process *validate* sends data *valid* of type bool to terminator *display terminal*;
In this case, *valid* = *true*, *request* = *enquiry*;
Process *enquiry* receives data *acc* of type *integer* from process *input*;
Process *enquiry* obtains data *b* of type *real* from data store *database*;
Process *enquiry* performs computation "*output the balance b of the account acc*.", or formally, "*balance_of*(*acc*,*b*)";
Process *enquiry* sends data *bal* of type real to terminator *display ternimal*;
{exit}

Withdrawal:
{begin}
Process *input* receives data *din* of type *INPUT_DATA* from terminator *operator*;
Process *input* performs computation *input*, or formally, "*and*(*cn* = *din.name*, *cid* = *din.iden*, *acn* = *din.account*, *amt* = *din. amount*, *rqst* = *din.request*)";
Process *input* sends data *cn* of type *string* to process *validate*;
Process *input* sends data *cid* of type *integer* to process *validate*;
Process *input* sends data *acn* of type *integer* to process *validate*;
Process *input* sends data *acc* of type *integer* to process *withdraw*;
Process *input* sends data *amt* of type *real* to process *withdraw*;
Process *validate* receives data *c n* of type *string* from the process *input*;
Process *validate* receives data *cid* of type *integer* from the process *input*;
Process *validate* receives data *acn* of type *integer* from process *input*;
Process *validate* performs "validate if *cn* of *cid* owns the account *acn* according to the information stored in the database", or formally, "$\exists r \in database.((acn = r.account) \wedge (cid = r.customer.identity) \wedge (cn = r.customer.name)) \Rightarrow valid = true; \neg\exists r \in database.((acn = r.account) \wedge (cid = r.customer.identity) \wedge (cn = r.customer.name)) \Rightarrow valid = false$";
Process *validate* sends data *valid* of type bool to terminator *display terminal*;
In this case, *valid* = *true*, *request* = *withdraw*;
Process *withdraw* receives data *acc* of type *integer* from process *input*;
Process *withdraw* receives data *amt* of type *real* from process *input*;
Process *withdraw* obtains data *balance* of type *real* from data store *database*;
Process *withdraw* performs computation "*withdraw the amt of money from account acc when the original balance b is greater than or equal to amt. Then, the balance is updated to be* $b' = b - amt$". *Then, the new balance bal is displayed, or formally,* "$balance\_of(acc, b) \wedge (b \geq amt) \Rightarrow (balance\_of(acc, b') \wedge (b' = b - amt))^{\wedge} bal = b'$";
Process *withdraw* updates the data *acc* of type *integer* in data store *database*;
Process *withdraw* updates the data *balance* of type *real* in data store *database*;
Process *withdraw* sends data *bal* of type real to terminator *display terminal*;
{exit}

Deposit:
{begin}
Process *input* receives data *din* of type *INPUT_DATA* from terminator *operator*;
Process *input* performs computation *input*, or formally, "*and*(*cn* = *din.name*, *cid* = *din.iden*, *acn* = *din.account*, *amt* = *din. amount*, *rqst* = *din.request*)";
Process *input* sends data *cn* of type *string* to process *validate*;
Process *input* sends data *cid* of type *integer* to process *validate*;
Process *input* sends data *acn* of type integer to process *validate*;
Process *input* sends data *acc* of type *integer* to process *deposit*;
Process *input* sends data *amt* of type *real* to process *deposit*;
Process *validate* receives data *cn* of type *string* from the process *input*;

Table 6
The process dictionary of the bank system

Name	Input	Output	Description	Definition
Input	input_data: INPUT_DATA	name: string; dentity: integer; acc: integer; amount: real	translate input_datato the name, identity of the customer, the the amount and the type of request	and(name = input_data.name,identity = input_data.iden, acc = input_data.account, amount = input_data.amount, request = input_data. request)
Validate	name: string; identity; interger; acc: integer;	valid: Bool	validate if the person *name* of identity *identity* owns the account *acc* according to the information stored in the database	$\exists$r $\in$ database.((acc = r.account) $\wedge$(identity = r.customer.identity) $\wedge$(name = r.customer.name)) $\Rightarrow$ valid = true; $\neg\exists$r $\in$ database.((acc = r.account) $\wedge$(identity = r.customer.identity) $\wedge$(name = r.customer.name)) $\Rightarrow$ valid = false;
Enquiry	acc: integer	bal: real	output the balance *bal* of the account *acc*.	balance_of(acc, bal)
Deposit	acc: integer; amount: real;	bal: balance	deposit the *amount* of money into account *acc* so that the original balance b stored in database is updated to $b' = b + amount$, the new balance *bal* is displayed	balance_of(acn, b) $\Rightarrow$ (balance_of(acn, b′) $\wedge$(b′ = amt + b))$^\wedge$bal = b′
Withdraw	acn: integer; amount: real	bal: balance	withdraw the *amount* of money from account *acc* when the original balance b is greater than or equal to amount. Then, the balance is updated to be $b' = b - amount$. Then, the new balance *bal* is displayed	balance_of(acn, b) $\wedge$ (b $\geq$ amt) $\Rightarrow$ (balance_of(acn, b′) $\wedge$ (b′ = b − amt))$^\wedge$bal = b′

Process *validate* receives data *cid* of type *integer* from the process *input*;
Process *validate* receives data *acn* of type *integer* from process *input*;
Process *validate* performs "validate if *cn* of *cid* owns the account *acn* according to the information stored in the database", or formally, "$\exists r \in database.((acn = r.account) \wedge (cid = r.customer.identity) \wedge (cn = r.customer.name)) \Rightarrow valid = true; \neg\exists r \in database.((acn = r.account) \wedge (cid = r.customer.identity) \wedge (cn = r.customer.name)) \Rightarrow valid = false$";
Process *validate* sends data *valid* of type bool to terminator *display terminal*;
In this case, *valid* = *true*, *request* = *deposit*;
Process *deposit* receives data *acc* of type *integer* from process *input*;
Process *deposit* receives data *amt* of type *real* from process *input*;
Process *deposit* obtains data *balance* of type *real* from data store *database*;
Process *deposit* performs computation "*deposit the amount of money into account acc so that the original balance b stored in database is updated to b'* = *b* + *amount*, *the new balance bal is displayed*", or formally, "$balance\_of(acn, b) \Rightarrow (balance\_of(acn, b') \wedge (b' = amount + b))^\wedge bal = b'$";
Process *deposit* update the data *acc* of type *integer* in data store *database*;
Process *deposit* update the data *balance* of type *real* in data store *database*;
Process *deposit* sends data *bal* of type real to terminator *display ternimal*;
{exit}

References

Abdurazik, A., Offutt, J., 2000. Using UML collaboration diagrams for static checking and test generation. In: Proceedings of the Conference on Unified Modelling Language, pp. 383–395.

Afifi, F.H., White, L.J., Zeil, S.J., 1992. Testing for linear errors in non-linear computer programs. In: Proceedings of the International Conference on Software Engineering, pp. 81–91.

Amla, N., Ammann, P., 1992. Using Z specifications in category partition testing. In: Proceedings of the Conference on Computer Assurance, pp. 3–10.

Ammann, P., Offutt, J., 1994. Using formal methods to derive test frames in category-partition testing. In: Proceedings of the Conference on Computer Assurance, pp. 69–79.

Anderson, J.S., Durley, B., 1993. Using scenarios in deficiency-driven requirements engineering. In: Proceedings of RE'93, pp. 134–141.

Annett, J., Duncan, K.D., 1967. Task analysis and training design. Occupational Psychology 41, 211–221.

Annett, J., Stanton N.A. (Eds.), 1998. Special issue: task analysis. Ergonomics, 1529–1737.

Basili, V.R., Selby, R.W., 1987. Comparing the effectiveness of software testing. IEEE Transactions on Software Engineering 13 (12), 1278–1296.
Beront, G., Gaudel, M.C., Marre, B., 1991. Software testing based on formal specifications: a theory and a tool. Journal of Software Engineering, 387–405.
Bertolino, A., Corradini, F., Inverardi, P., Muccini, H., 2000. Deriving test plans from architectural descriptions. In: Proceedings of ICSE'2000, pp. 220–229.
Boehm, B.W., 1981. Software engineering economics. Advances in Computing Science and Technology. Prentice-Hall, Englewood Cliffs, NJ.
Bouge, L., Choquet, N., Fribourg, L., Gaudel, M.C., 1986. Test set generation from algebraic specifications using logic programming. Journal of System and Software 6, 343–360.
Budd, T.A., 1980. Mutation analysis: ideas, examples, problems and prospects. In: Chandrasekaran, Radicchi (Eds.), Computer Program Testing. North Holland, Amsterdam, pp. 129–149.
Carroll, J.M., 2000. Making use: Scenario-based Design of Human–Computer Interactions. MIT Press, Cambridge, MI.
Chen, H.Y., Tse, T.H., Chen, T.Y., 2001. TACCLE: a methodology for object-oriented software testing at the class and cluster levels. ACM Transactions on Software Engineering and Methodology 10 (1), 56–109.
Chen, H.Y., Tse, T.H., Deng, Y.T., 2000. ROCS: an object-oriented class-level testing system based on the relevant observable contexts technique. Information and Software Technology 42 (10), 677–686.
Clarke, L.A., Podguski, A., Richardson, D.J., 1982. A close look at domain testing. IEEE Transactions on Software Engineering 8 (11), 1318–1332.
DeMillo, R.A., Lipton, R.J., Sayward, F.G., 1978. Hints on test data selection: help for the practising programmer. Computer 11 (April), 34–41.
Diaper, D. (Ed.), 1989. Task Analysis for Human–Computer Interaction. Ellis Horwood, Chichester, UK.
Diaper, D., McKearney, S., Hurne, J., 1998. Human–computer interaction and software engineering: integrating task and data flow analyses using the pentanalysis technique. Ergonomics 41 (11), 1553–1582.
Frankl, P.G., Weyuker, J.E., 1988. An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering SE-14 (10), 1483–1498.
Frankl, P.G., Weyuker, J.E., 1993. A formal analysis of the fault-detecting ability of testing methods. IEEE Transactions on Software Engineering 19 (3), 202–213.
Fujiwara, S., Bockmann, G., Khendek, F., Amalou, M., Ghedamsi, A., 1991. Test selection based on finite state models. IEEE Transactions on Software Engineering 17 (6), 591–603.
Gild, T., Gramham, D., 1993. Software Inspection. Addison-Wesley, Reading, MA.
Goodenough, J.B., Gerhart, S.L., 1975. Toward a theory of test data selection. IEEE Transaction on Software Engineering SE-3, 156–173.
Haumer, P., Pohl, K., Weidenhaupt, K., 1998. Requirements elicitation and validation with real world scenes. IEEE Transactions on Software Engineering 24 (12), 1036–1054.
Hetzel, W., 1984. The Complete Guide to Software Testing. Collins.
Howden, W.E., 1976. Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering SE-2 (9), 208–215.
Howden, W.E., 1987a. Functional Program Testing and Analysis. McGraw-Hill, New York.
Howden, W.E., 1987b. Weak mutation testing and completeness of test sets. IEEE Transactions on Software Engineering 8 (4), 371–379.
Hsia, P., Samuel, J., Gao, J., Kung, D., Toyoshima, Y., Chen, C., 1994. Formal approach to scenario analysis. IEEE Software, 33–41.
Jacobson, I. et al., 1992. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, Reading, MA.
Jin, L., Zhu, H., 1997. Automatic generation of formal specification from requirements definition. In: Proceedings of the Conference on Formal Engineering Method, pp. 243–251.
Jin, L., Zhu, H., 1998. Description and analysis of use scenarios in requirements engineering. In: Proceedings of SEKE'98, pp. 18–25.
Jin, L., Zhu, H., Hall, P., 1997. Adequate testing of hypertext applications. Journal of Information and Software Technology 39 (4), 225–234.
Johnson, P., Diaper, D., Long, J., 1984. Tasks, skills and knowledge: task analysis for knowledge based descriptions. In: Proceedings of the Conference on Human–Computer Interaction, vol. 1, pp. 23–27.
Kamsties, E., Hormann, K., Schlich, M., 1998. Requirements engineering in small and medium enterprises. Journal of Requirements Engineering 3 (2), 84–90.
Laski, J., Korel, B., 1983. A data flow oriented program testing strategy. IEEE Transaction on Software Engineering SE-9, 33–43.
Long, J., 1989. Cognitive ergonomics and human–computer interaction: an introduction. In: Long, J., Whitefield, A. (Eds.), Cognitive Ergonomics and Human–Computer Interaction. Cambridge University Press, Cambridge, pp. 4–34.
Long, J., 1997. Research and the design of human–computer interactions or What happened to validation? In: Proceedings of HCI'97, pp. 223–243.
Luqi, Chang, C.K., Zhu, H., 1998. Specifications in software prototyping. Journal of Systems and Software 42 (2), 125–140.
Macdonald, F., Miller, J., Brooks, A., Roper, M., Wood, M., 1995. A review of tool support for software inspection. In: Proceedings of the International Workshop on Computer Aided Software Engineering, pp. 340–349.
Macdonald, F., Miller, J., Brooks, A., Roper, M., Wood, M., 1996. Automating the software inspection process. Automated Software Engineering 3, 193–218.
McCabe, T.J. (Ed.), 1983. Structured Testing. IEEE Computer Society Press, Silver Spring, MD.
Morasca, S., Pezze, M., 1990. Using high-level Petri nets for testing concurrent and real-time systems. In: Proceedings of the Conference on Real-Time Systems: Theory and Applications, pp. 119–131.
Myers, G.J., 1979. The Art of Software Testing. Wiley, New York.
Ntafos, S.C., 1984. On required element testing. IEEE Transaction on Software Engineering SE-10 (6), 795–803.
Nuseibeh, B., Kramer, J., Finkelstein, A., 1994. A framework for expressing the relationships between multiple views in requirements specification. IEEE Transactions on Software Engineering 20 (10), 760–773.
O'Neill, E., Johnson, P., Johnson, H., 1999. Representations and user–developer interaction in cooperative analysis and design. Human–Computer Interaction 14, 43–91.
Offutt, J., Abdurazik, A., 1999. Generating tests from UML specifications. In: Proceedings of the UML'99: Beyond the Standards. Lecture Notes in Computer Science, vol. 1723, pp. 416–429.
Ould, M.A., Unwin, C. (Eds.), 1986. Testing in Software Development. Cambridge University Press, Cambridge.
Paige, M.R., 1978. An analytical approach to software testing. In: Proceedings of COMPSAC'78, pp. 527–532.

Parrish, A.S., Zweben, S.H., 1993. Clarifying some fundamental concepts in software testing. IEEE Transactions on Software Engineering 19 (7), 742–746.

Pezze, M., Young, M., 1996. Generation of multiformalism state-space analysis tools. In: Proceedings of the International Symposium on Software Testing and Analysis, pp. 172–179.

Podgurski, A., Clarke, L.A., 1989. The implications of program dependences for software testing, debugging and maintenance. In: Proceedings of the Symposium on Software Testing, Analysis, and Verification, pp. 168–178.

Podgurski, A., Clarke, L.A., 1990. A formal model of program dependences and its implications for software testing, debugging and maintenance. IEEE Transactions on Software Engineering 16 (9), 965–979.

Rapps, S., Weyuker, E.J., 1985. Selecting software test data using data flow information. IEEE Transactions on Software Engineering SE-11 (4), 367–375.

Richardson, D., Wolf, A., 1996. Software testing at the architectural level. In: Proceedings of the Software Architecture Workshop, pp. 68–71.

Richardson, D.J., Clarke, L.A., 1985. Partition analysis: a method combining testing and verification. IEEE Transactions on Software Engineering 11 (12), 1477–1490.

Richardson, D.J., Thompson, M.C., 1993. An analysis of test data selection criteria using the RELAY model of fault detection. IEEE Transactions on Software Engineering 19 (6), 533–553.

Rosenblum, D.S., 1997. Adequate testing of component-based software. Technical Report UCS-ICS-97-34, Department of Computer Science, University of California, Irvine.

Stocks, P.A., Carrington, D.A., 1993. Test templates: a specification-based testing framework. In: Proceedings of International Conference on Software Engineering, pp. 405–414.

Stokes, D.A., 1991. Requirements analysis. In: Software Engineer's Reference Book. Butterworth-Heinemann, London.

Weyuker, E.J., 1986. Axiomatizing software test data adequacy. IEEE Transactions on Software Engineering SE-12 (12), 1128–1138.

Wheele, D.A. (Ed.), 1996. Software Inspection: An Industry Best Practice. IEEE Computer Society Press, Los Alamitos.

Xu, J., Zhu, H., et al., 1995. From requirements definition to formal functional specification – A transformational approach. Science in China 38 (Supp.), 28–43.

Yourdon, E., 1989a. Modern Structured Analysis. Prentice-Hall, Englewood Cliffs, NJ.

Yourdon, E., 1989b. Structured Walkthroughs, fourth ed. Prentice-Hall International, Englewood Cliffs, London.

Zeil, S.J., 1983. Testing for perturbations of program statements. IEEE Transactions on Software Engineering 9 (3), 335–346.

Zeil, S.J., 1989. Perturbation techniques for detecting domain errors. IEEE Transactions on Software Engineering 15 (6), 737–746.

Zhu, H., 1996. A formal analysis of the subsume relation between software test adequacy criteria. IEEE Transactions on Software Engineering 22 (4), 248–255.

Zhu, H., Hall, P., 1993. Test data adequacy measurement. Software Engineering Journal 8 (1), 21–30.

Zhu, H., Jin, L., 2000. Scenario analysis in an automated tool for requirements engineering. Journal of Requirements Engineering 5 (1), 2–22.

Zhu, H., Hall, P., May, J., 1997. Software unit test coverage and adequacy. ACM Computing Surveys 29 (4), 366–427.

Zhu, H., Jin, L., Diaper, D., 1999a. Application of task analysis to the validation of software requirements. In: Proceedings of SEKE'99, pp. 239–245.

Zhu, H., Jin, L., Diaper, D., 1999b. Activity lists as a description of software behaviour for requirements validation. Technical Report CMS-TR-99-01, School of Computing and Mathematical Sciences, Oxford Brookes University.

Dr. Hong Zhu is a senior lecturer in computing at the School of Computing and Mathematical Sciences of Oxford Brookes University, UK. He received his B.Sc., M.Sc. and Ph.D. degrees in computer science from Nanjing University, China, in 1982, 1984 and 1987, respectively. From 1987 to 1990, he worked at the Institute of Computer Software and Department of Computer Science at Nanjing University as a lecturer and later as an associate professor in computer science. From 1990 to 1994, he was a research fellow at Brunel University and at the Open University, UK. He returned to Nanjing University in December 1994. Since 1996 he was a professor of computer science at Nanjing University before he joined Oxford Brookes University in 1998. He has published over 60 papers in journals and conference proceedings. He received the NSF of China's Premier's Award of Distinguished Young Scientists in China in 1997. His current research interests include software testing, requirements engineering, software agent technology, and web engineering.

Dr. Lingzi Jin received her B.Sc., M.Sc. and Ph.D. degrees in computer science from Nanjing University, China, in 1982, 1984 and 1987, respectively. From 1987 to 1998, she worked at the Institute of Computer Software and Department of Computer Science at Nanjing University as a lecturer and later as an associate professor in computer science. From 1992 to 1994, she was on leave from Nanjing University and visited the Open University, UK, as a research fellow. Since 1999, she has been working in the IT industry as a software test manager at Cherwell Scientific Ltd., Oxford, UK, recently renamed as Family Genetix. She has recently joined AIT plc., Henley-on-Thames, Oxfordshire, as senior software tester. She has published widely in the area of software engineering, including software testing and quality management, software internationalization and localization, reverse engineering, software requirements engineering, program transformation, etc.

Prof. Dan Diaper joined Bournemouth University in 1996 to take up the post of Head of the Department of Computing in the School of Design, Engineering and Computing. Since the University's academic reorganisation in 1998 he has primarily had a research role as Professor of Systems Science and Engineering. Diaper gained his doctorate from the University of Cambridge in 1982 for his work in the Department of Experimental Psychology on the methodology underlying the experimental evidence of visual subliminal perception. He worked at the Ergonomics Unit, University College London as a Research Assistant and then Research Fellow on various Human–Computer Interaction (HCI) related projects. In 1986 he took the post of Senior Lecturer in the Department of Psychology at Liverpool Polytechnic (now Liverpool John Moores University). In 1989 he made a career change, taking a lectureship at the Department of Computer Science at the University of Liverpool. Diaper has published more than 60 research papers in journals and conference proceedings on a wide range of HCI and software engineering topics and he has edited seven books. Until 1999 he was for a decade the General Editor of the international journal *Interacting with Computers*: *The Interdisciplinary Journal of Human–Computer Interaction*. He continues to co-edit the Springer-Verlag book series on Computer-Supported Co-operative Work (CSCW). Diaper is a corporate member of the British Computer Society and has served on many of its committees, including Council. During the 1990s he was Chair of the Department of Trade and Industry's CSCW Special Interest Group.

Mr. Ganhong Bai received his M.Sc. degree in computer science from Nanjing University, China, in 1999. He is working for Bell Labs China, Lucent Technologies. His working area is telecommunication software development.