Automatic Generation of Formal Specification from Requirements Definition

Lingzi Jin and Hong Zhu
Computer Software Institute, Nanjing University,
Nanjing, 210093, China, Tel.: +86 (25)3594683, Email: (ljin | hzhu)@netra.nju.edu.cn

Abstract
Developing software on the bases of formal functional specifications in first order logic and algebraic languages has been the prime endeavour of research on formal methods. But, the industrial development community has been slow to adopt formal methods. Instead, researchers and software engineers in the area of software requirements engineering have advanced various notations for defining user's requirements and systematic methods of developing requirements definitions. There is a wide gap between the current practice of software requirements engineering and the research on formal specification and software formal development. This paper presents the system NDRASS, which support requirements engineering with the methods of current state of practice and to link such practice to formal methods by automatically generating formal functional specifications in Z.

1. Introduction
Deriving programs from formal functional specifications in first order logic and algebraic languages and proving program correctness with respect to such formal specifications have been the prime endeavour of research on formal methods. However, it is difficult to develop such formal functional specifications. The industrial development community has been slow to adopt formal methods except some acceptance and practice. There is growing realisation that training and education may not be the answer. In contrast, researchers and software engineers in the area of software requirements engineering have advanced various notations for defining user's requirements and systematic methods of developing requirements definitions.

Requirements analysis and specification are concerned with eliciting, clarifying and documenting user's requirements of a computation system and producing the corresponding functional specification [17]. Many studies have shown that errors made at this stage are very costly (even impossible) to rectify. Neglected or only partially completed requirements analysis tends to lead to problems later in development. It is perceived as an area of growing importance. [14],[15],[17]

In the analysis and specification of user's requirements, software engineers are often confronted with difficulties due to the complexity of the problem, the communication barriers between people of diverse backgrounds, the inconsistency and incompleteness of information and frequent changes of user’s requirements. They must talk to a wide range of people, with diverse background, interests, and personal goals. The communication is not only in the direction of information elicitation, but also in the direction of the validation of elicited information. As the main result of information elicitation and analysis, a requirements definition should be a complete and precise statement of user's requirements on the functions, the constraints on the development and the target environments and the goals of the system. This contractual document must be readable by the client for validation, by the managers for decision making and by the software engineers for development. A great amount of effort of research on requirements definition have been spent in the search for appropriate representation format of requirements so that it can be read by these people. It is recognised that people of diverse background and different viewpoints need different representations. It is desirable that multiple representations of various views are used in defining software requirements and automatic transformation from one view to another is supported by software tools [12]. Among various representations, informal representations such as statements in natural languages are of obvious advantages with respect to overcoming the barriers of communication with people of diverse backgrounds. Its disadvantages are also obvious, such as ambiguity. Formal representations provide a solution to the ambiguity problem. They also have advantages of offering possibilities of the formal proof and analysis of properties of requirements such as formal verification of consistency and completeness. Moreover, formal functional specification in first order logic and algebraic formal specification languages is the base of software formal development and formal verification of implementation correctness. One of the main disadvantages of formal specification methods is that they are difficult to develop and understand for people having no background in formal methods and mathematics. As a trade-off between the preciseness of formal representations and the ease of communication of informal representations, the use of semi-formal representations is the state of practice. A semi-formal representation, such as diagrams, uses a well-defined notation but may contain informal components. The coexistence of formal and informal
representations in requirements definitions has been more and more widely accepted by researchers and practitioners. However, the mixture of informal and formal representations does not facilitate software formal development and verification of implementation correctness. There is a wide gap between the current practice of software requirements engineering and the research on formal specification and software formal development. How to bridge the gap is the subject of this paper.

This paper reports a research that aims at developing software tools to support requirements engineering with the methods of current state of practice and to link such practice to the formal methods by automatically generating formal functional specifications in well established specification languages such as Z [16]. We present a system NDRASS, which supports automatic generation of formal functional specifications in Z from requirements definitions in a requirements definition language called NDRDL [4]. The language is based on the classic method of structured analysis [22], but slightly extended by very limited uses of first order logic as formal representation. It provides facilities supporting requirements definition by diagrams and dictionaries.

The paper is organised as follows. Section 2 is an overview of the language NDRDL and system NDRASS. Section 3 focuses on the automatic generation of formal functional specifications in Z. Section 4 is the conclusion of the paper.

2. Overview of the NDRDL language and NDRASS system

We identify two kinds of supports to requirements engineering: (a) language supports, and (b) tool supports [19]–[21]. Language supports provide language facilities in which user’s requirements are represented, expressed and communicated. They can help software engineers to cope with difficulties due to the complexity of the problem and communication barriers. Tool supports use software tools to perform or help to fulfil various tasks involved in requirements analysis and specification, such as the storage, management and retrieval of specified user’s requirements, the analysis of expressed requirements, and the transformation of one representation into another. They can help to deal with incompleteness and inconsistency and frequent changes of user’s requirements.

2.1 The NDRDL language

The NDRDL language [4] is based on the classic methods of structured analysis [22]. A requirements definition in NDRDL has a quite standard hierarchical structure. In addition to informal description of user’s requirements in natural language, a requirements definition in NDRDL contains three diagrams and three dictionaries. The diagrams include an entity-relationship diagram (ERD), a data flow diagram (DFD) and a control flow diagram (CFD). The set of dictionaries includes a data dictionary, a relationship dictionary and an operation dictionary. The following illustrates the forms and usage of the diagrams and dictionaries by an example of Personal Bank Account Management System (PAMS).

(A) Entity-relationship diagram.

An entity-relationship diagram provides a semi-formal data model of the required system by specifying the entities and their relationships involved in the system. Figure 2 is the entity-relationship diagram of PAMS. When entities and the relationships represent the data and their relationships in the real world, the diagram describes a part of the subject world. Details of the data and their relationships are further described in data dictionary and relationship dictionary. They are a part of the domain knowledge related to the particular requirements. When an entity is to be generated by the system and when a relationship is to be implemented or maintained by the software, the diagram also describes a part of the system world. This nature of entity-relationship diagrams is well illustrated by the PAMS, in which the customer, bank account and capital belong to the subject world. Although the owns and balance_of relationships belong to the subject world, they are to be managed by the system.

(B) Data flow diagram.

A data flow diagram describes the source of data, the process of the data within the system and the final receivers of the processed data. The data flow diagram of the PAMS is given in Figure 4. The process of the data describes the...
functionality of the system, hence belongs to the system world. The dataflows from the sources and among the process nodes embed the functionality into the subject world. Obviously, the part of the subject world described in data flow diagrams overlaps with that contained in the entity-relationship diagram. Therefore, inconsistency may occur. NDRDL language explicitly imposes consistency constraints on the two diagrams. Readers are referred to [21] for details of the consistency constraints.

![Dataflow Diagram Elements](image1)

**Figure 3. Elements of dataflow diagram**

![Dataflow Diagram of PAMS](image2)

**Figure 4. Data flow diagram of PAMS.**

(C) **State transition diagram.**

A state transition diagram is a flow graph, which is a directed graph with a start node and at least one exit node such that any node in the graph is on a path from the start node to an exit node. In a state transition diagram, 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 use of the software. Only such sequences of events are allowed to happen in the use of the software. State transition diagrams provide, therefore, a means of describing task oriented usage of the software. The state transition diagram of the PAMS is given in Figure 5. From this diagram, we can see that in the use of the system, the validity of a customer's personal information must be checked before any transactions can proceed.

The events associated with the arcs in a state transition diagram can either be an invoke of a process or a predicate describing an event in the outside world. These information must be consistent with those specified in the corresponding data flow diagram and entity-relationship diagram. Therefore, consistency constraints are also imposed by NDRDL language.

![State Transition Diagram of PAMS](image3)

**Figure 5. State transition diagram of PAMS**

(D) **Data dictionary.**

The data dictionary defines the form and usage of data in the required system. An entry in a data dictionary consists of four fields. The first is the name of the data to be defined. The second is the informal description of the data in natural language. The third is a formal description of the data. It gives the type, hence the structure, of the data. The type of the data is represented similar to a data type in programming languages. The fourth is the constraint on its value. The constraint is in the form of a first order predicate. Figure 6 gives the data dictionary of the PAMS.

![Data Dictionary](image4)

**Figure 6. Data dictionary of PAMS**

(E) **Relationship dictionary.**

The relationship dictionary defines the relationship between entities. An entry in a relationship dictionary contains four fields. The first field is the name of the relationship. The second field gives the entities of the relationship. The third field is an informal description of the relationship in natural language. The fourth field is a formal definition of the relationship written as a predicate. The relationship dictionary of PAMS is given in Figure 7.

![Relationship Dictionary](image5)

**Figure 7. Relationship dictionary of PAMS**

(F) **Operation dictionary.**

An entry in an operation dictionary defines an operation on data. It consists of five fields. The first field is the name of the operation to be defined. The second and the third are the input and output of the operation, respectively. The fourth is an informal description of the function of the operation. The fifth is the formal definition of the operation in the form of a predicate that relates the output to the input. Figure 8 gives the operation dictionary of the PAMS.

![Operation Dictionary](image6)

**Figure 8. Operation dictionary of PAMS**
The dictionaries provide links connecting the informal, semi-formal and formal representations. Each entry in a dictionary contains an informal description of the defined terminology and a formal definition. Due to the redundancy among the diagrams and dictionaries, inconsistency and incompleteness may occur. The definitions of the data, relationships or operations in the dictionaries may also be inconsistent with their uses in the diagrams. Therefore, some constraints on them are imposed to obtain complete and consistent requirements definitions; see [21] for details of the consistency constraints.

### 2.2 The NDRASS system

There is a wide gap between informal description of requirements definition and formal specification. User’s initial requirements statement must be in informal or semi-formal representations such as in natural languages.
and diagrams, whilst any decent analysis of the requirements has to be based on a formal representation. Given the current state of the art of natural language understanding, it seems impossible to build a practical tool to bridge the gap between informal and formal descriptions. Therefore, we take a progressive and orderly transition approach to requirements engineering. As shown in Figure 10, we consider the process of requirements elicitation, analysis, documentation and specification as a sequence of interacting and iterating phases. It starts with an informal and vague requirements statement, which is then gradually transformed into a formal and consistent functional specification. This approach is taken by the NDRASS system.

As shown in Figure 9, NDRASS system provides the following facilities to support this approach:

- A **text editor** for the edition and modification of texts in natural language;
- Graphic editors for the edition and modification of various diagrams;
- Managers of dictionaries for the edition, modification, browse and management of dictionaries;
- An automatic checker of the consistency and completeness among dictionaries and diagrams;
- Two automatic generators: one for the generation of frameworks of dictionaries; the other for the generation of formal functional specifications in Z.

As illustrated in Figure 10, a typical requirements analysis process that NDRASS supports starts with production of an informal description of user’s requirements. This is supported by a text editor. The second step is the construction of semi-formal models of the required system. This is supported by the graphic editors of NDRASS. Once consistency between the diagrams is checked, the dictionary generator can be invoked to generate frameworks of the dictionaries. A framework of data dictionary consists of all the data names used in the diagrams, their data structure according to the ERD. The fields of informal description and constraints are left to be filled by requirements engineer manually. A framework of relationship dictionary consists of the names of the relationships appeared in ERD, and the entities involved in the relationship. The fields of informal description of the relationship and formal definition of the relationship are left to be manually filled in. The framework of operation dictionary consists of the names of the processes appeared in the data flow diagram, and the input, output of the
process. The fields of informal description and formal definition of the process are left to be filled in manually. The completion of the dictionaries are supported by the dictionary manager. Once the dictionaries have been completed, the consistence between the diagrams and dictionaries can be checked, and then a formal functional specification in Z can be automatically generated.


This section will focus on the automatic generation of functional specifications in Z from consistent and complete requirements definitions in NDRDL. The Z language provides schemas to modular descriptions of the state space and the operations and functions of a software system. Readers are referred to [16] for the Z notations.

3.1 The Generation of State Space Descriptions

Two types of schemas, entity schema and relationship schemas, are generated according to the information contained in ERD, data dictionary and relationship dictionary. For each entity in the ERD, an entity schema is generated according to the following rules, see Figure 11.

A. The name of the schema is the entity name;
B. For each attribute attr of type T, a declaration attr:T is included in the declaration part of the schema;
C. If the entry of the entity in the data dictionary contains a predicate to describe the constraints on its values, the predicate is copied into the predicate part of the schema with some syntactical transformations.

Entity schemas are used as types, i.e. to give (a) the types of state variables, (b) the types of the parameters, input and output of functions and operators, and (c) the types of attributes in the definition of other entity schemas.

The system’s state space is determined by the data storages contained in the data flow diagram. For each data storage DataStore in the data flow diagram, the following schema is generated:

```
DataStore
Var_DataStore : Type_of_storage
Constraint_predicate
```

where Type_of_storage is the schema name which defines the type of the data storage. It can be an entity name if the schema has been generated as described above. Otherwise, it will be a schema name that is generated according to the type specified in the data dictionary. When structured data types are used, intermediate schemas will be generated. For example, the following schemas will be generated for the database storage in the data flow diagram of the PAMS.

```
Type_X1
Customer: customer
Account: account
Balance: money
Balance.amount >= 0
```

A relationship schema defines a relationship among entities. For each relationship R in the entity-relationship diagram, a relationship schema is generated according to the following rules, as illustrated in Figure 12.

A. The schema gives a global definition of relation R, which is defined as a Cartesian product of the entities involved in the relationship;
B. If an entity Y participates in the relationship as a multiple relation, a predicate in the following form is included into the predicate part of the definition:

```
\forall a:Entity_A, \exists! c:Entity_C.R(a, \ldots, c)
```

C. If an entity Y participates in the relationship as a unique relation, a predicate in the following form is included into the predicate part of the definition:

```
\forall c:Entity_C, \exists a:Entity_A.R(a, \ldots, c)
```

Figure 11. Generation of an entity schema from NDRDL.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Form</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity A</td>
<td>…</td>
<td>Record</td>
<td>P(Attr a, …, Attr c)</td>
</tr>
</tbody>
</table>
```

Figure 12. Generation of relationship schema.
D. If the entry of the relationship in the relationship dictionary contains a predicate as a formal definition, the predicate is also copied into the predicate part of the schema with some syntactical transformation.

3.2 The generation of function / operation definitions

The definitions of functions and operations are generated according to the information contained in the DFD and operation dictionary. For each process in DFD, an operation schema is generated according to the following rules.

A. The name of the schema is the name of the process;
B. For each inward dataflow that does not come out of a data store, "X? : TX" is included in the declaration part for the data X of type TX associated with the flow;
C. For each outward data flow that does not go into a data store, "Y! : TY" is included in the declaration part for the data Y of type TY associated with the flow;
D. If a data store DS has data flowing into the process node, the DS schema is included into the operation schema; if there are data flowing from the process node into a data store DS, the DS schema is included with decoration D;
E. The predicate P in the operation dictionary is transformed into P' in Z notation and included in the predicate part. In addition to the syntactical transformations, variables in P must also be systematically decorated according to the following rules:
   (a) for each input variable x, if it is associated with an inward data flow coming out from a data store, it is unchanged. Otherwise, x is replaced with x?;
   (b) for each output variable y, if it is associated with an outward data flow going to a data store, it is replaced with y'. Otherwise, it is replaced with y!.

For example, the following is the schema for the deposit operation of PAMS.

```
<table>
<thead>
<tr>
<th>Name</th>
<th>Input</th>
<th>Output</th>
<th>Description</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Op</td>
<td>x1:Tx1,y1:Ty1</td>
<td></td>
<td>P(x1,x2,y1,y2)</td>
<td></td>
</tr>
</tbody>
</table>
```

```
deposite
bank_database
Δ bank_database
an? : account_number
m? : money
b! : Real

balance_of(an?, x) => (balance_of(an?, x') ^ (x'=x+m?)^(b!=x'))
```

3.3 The generation of system operation structure

In Z language, a software system is described as a function on the state space. This function will be generated according to the CFD.

The generation of system control function is based on Fenton et al.'s theory of structured programming [6] to improve the readability of generated Z code. According to the theory, every flow graph can be uniquely decomposed into a set of prime graphs so that it is the composition of the prime graphs by the nest and composition operations. Figure 14 gives some examples of prime graph which correspond to control structures.

![Figure 14. Examples of prime flow graphs.](image1)

The generation process consists of the following steps.

A. A flow graph is normalised so that it contains only one start node and one exit node and every node has at most two outward arcs;
B. The flow graph is decomposed into prime graphs such that the flow graph is represented as a decomposition tree. Given a flow graph, the decomposition starts with finding prime flow graphs. A prime flow graph is then reduced to an arc from the start node to the exit node of the prime sub-graph. Such a reduction process is recorded and a decomposition tree is constructed;
C. The Z description of the flow graph is generated according to the decomposition tree.

NDRASS selects a set of prime flow graphs as well structured CFD. Such prime flow graphs have well
structured and readable Z descriptions as shown in Figure 14. Prime flow graphs not in the set are considered as not well structured control structures. Once such a prime flow graph is detected, the user is warned and asked if any modification will be made. If no any, a recursive Z description of the prime flow graph will be generated. The distinction of well structured from not well structured sub-graphs enables us to control software complexity at requirements engineering stage and helps quality control. Figure 15 gives the process of normalisation and decomposition of the state transition diagram.

4. Conclusion

Addressing the difficulties of requirements engineering, the literature has advanced a number of proposals, such as:

• integrating multiple views and representations (formal and informal) to soothe the communications of people [7], [12];
• modelling requirements engineering processes in various paradigms to provide guidelines for the development of requirements definition and the transition from the informal to the formal [7], [8], [11];
• developing methods and software tools to support resolving conflict requirements, coping with incompleteness [1], automating the transition from the informal to the formal [8], etc.;
• modelling software systems and their environments and employing abstract domain knowledge and object-oriented methodology to manage requirements changes [2], [13];

This list of research work is far from complete. The most closely related work is perhaps the work of Fraser et al. They reported a system which can semi-automatically generate formal functional specifications in VDM from data flow diagram [8]. There is not a single approach that has solved all the problems in requirements engineering. A number of research programmes, such as the NATURE project of ESPRIT III programme [10], have been launched to integrate some existing theories and techniques. Our approach to requirements engineering is a progressive and orderly transition approach, which is characterised by a step by step transition from informal to semi-formal, and finally, into formal descriptions. This approach is supported by the NDRDL language and the NDRASS system. Once a complete and consistent requirements definition is obtained, a formal functional specification in Z can be automatically generated by NDRASS system. NDRASS system has been implemented on Sun Workstation Sparc 490 at the Institute of Computer Software at Nanjing University.

We are further extending the expressiveness of NDRDL language and the power of NDRASS system to support the automatic generation of diagrams and dictionaries.

Acknowledgement

The authors would like to thank Prof. J. Xu and many colleagues and postgraduate students at The Institute of Computer Software, Nanjing University, who participated in the research project reported in this paper.

5. References

Theoretical Computer Science, 36, 145-171.


