Report Number: WUCS-82-7

1982-02-01

A Rigorous Approach to Building Formal System Requirements

Gruia-Catalin Roman

This paper reports the author's experience in the use of formal specifications and presents a step by step approach to developing functional requirements for computer-based systems. A simple model of system requirements is introduced first. A systematic approach to developing requirements by starting with the general model and adapting it to the needs to the problem at hand is described and illustrated by means of the simple but realistic example. A basic knowledge of predicate calculus and set theory is assumed on the part of the reader. The presentation is tutorial in nature.

A RIGOROUS APPROACH TO
BUILDING FORMAL SYSTEM REQUIREMENTS

Gruia-Catalin Roman

WUCS-82-7

February 1982

Department of Computer Science
Washington University
St. Louis, Missouri 63130

As appeared in Proceedings of the Computer Software and Applications Conference, November 1982, pp. 417-423.

ABSTRACT

This paper reports the author's experience in the use of formal specifications and presents a step by step approach to developing functional requirements for computer-based systems. A simple model of system requirements is introduced first. A systematic approach to developing requirements by starting with the general model and adapting it to the needs of the problem at hand is described and illustrated by means of a simple but realistic example. A basic knowledge of predicate calculus and set theory is assumed on the part of the reader. The presentation is tutorial in nature.

Acknowledgement: This work was partially supported by Rome Air Development Center and by Defense Mapping Agency under contract F30602-80-C-0284.

Keywords: formal specifications, functional requirements.

INTRODUCTION

Formal definition of functional requirements for computer-based systems has received considerable attention in recent years. A measure of the researchers' concern with this topic is the great variety of specification language proposals that have been put forth. They differ in the degree of formality, power, and the nature of the formalisms being used. Some approaches are based on the use of finite-state machines [HENI79] and, thus, they offer simplicity but also reduced power. Others emphasize dataflow (SADT [ROSS77], PSL/PSA [TEIC77]). They are concerned with the functions to be performed by the system and the data being passed between them. Yet another approach is used in RSL [BELL77] which defines the requirements in terms of stimulus-response paths. Attempts have also been made to capture the behavior characteristics of the systems in algebraic specifications, e.g., [RIDD78], and as partial orders [GREI77]. Data-oriented modeling of the requirements has been stimulated by efforts in the database area [TSIC82] while applicative languages have been advocated as a means to achieve executability of the requirements [ZAVE81].

The dominant concerns of those involved in the development of requirements specification languages have been the ease of use and the potential for automation of their respective proposals. There are, however, a number of other important issues demanding careful investigation. They relate to the pragmatics of using formal specifications. How one choses an appropriate specification language, how one develops the specifications, how one gets started, are questions often formulated by designers that feel the need for improvements in the quality of the system requirements but have no experience with the use of formal specifications. These questions also explain why formal specifications are rarely used in practice despite the great need to accumulate experience in this area and despite the benefits they promise.

This paper addresses several of these issues. It reports on the author's experience in the use of formal specifications and presents a step by step approach to developing formal requirements. The tutorial is intended to give assistance and confidence to the novice and to share with other practitioners some observations about the nature of system requirements. A basic knowledge of predicate calculus and set theory is assumed on the part of the reader. While no exposure to any requirements definition language is required, some appreciation for the role the requirements play in the development of the system is necessary.

The remainder of the presentation is separated into four sections. The first one introduces a formal model of system requirements. A systematic approach to developing formal requirements by starting with the general model and by adapting it to the needs of the problem at hand is described and illustrated by means of a simple but realistic example in the section that follows. A discussion of several topics related to the development of formal requirements, including unresolved issues and current concerns, precedes the conclusions.

FORMAL MODEL OF SYSTEM REQUIREMENTS

The system requirements are generated in the problem definition stage and prior to any attempt at system design. They consist of a conceptual model and a set of constraints which together define the acceptability criterion for any proposed system realization. A system is said to meet its requirements if and only if it carries out the functionality described by the conceptual model and satisfies all relevant constraints, also called non-functional requirements.

The role of the conceptual model is to capture in finite and precise terms the functional aspects of the interaction between the needed system and its environment. The constraints, on the other hand, limit the design space by imposing restrictions over the class of systems the designer might consider. Actually, the degree of complexity of a system is measured not by size alone but also by the severity of the constraints it must satisfy.

Before continuing the discussion of the conceptual model which is the main concern of this paper, it ought to be pointed out that recent increases in the ability to define formally the desired functionality have not been accompanied by commensurable advances in the definition of system constraints. There are four important reasons contributing to this state of affairs. First, there is a great diversity of types of constraints (e.g., response time, space, reliability, cost, schedule, weight, power, etc.). Second, some of them are related to possible design solutions which are not formally stated at the time the system requirements are conceived. Furthermore, their relevance differs at different points in the design. Third, many constraints are not formalizable given current state-of-the-art. Finally, not all constraints are explicit. For instance, the designer is expected to follow generally accepted rules of the trade in designing a system without having them explicitly stated.

In general, there is considerable agreement among authors with regard to the nature of the conceptual model. The conceptual model must have the ability to describe all pertinent environmental states, an abstraction of the system states, and the way in which both environmental and system states change. The conceptual model is denoted CM and is defined as follows:

CM = (E, EO, S, SO, F) where

E is the set of environmental states,
EO is the set of possible initial environmental states,
S is the set of system states,
SO is the set of possible initial system states,
F is the set of state transition rules, i.e.,
F SUBSET.OF ((E x S) x (E x S)).

(See the Notation Summary for conventions used in this paper.)

Although significantly more complex models have been developed for the study of systems in general (e.g., [WYMO67]), this simple model appears to be adequate for many types of computer-based systems.

The definition above makes clear two important facts. First, because both the environmental and the system states may be infinite in number, the model may not be reduced, in general, to a finite-state machine. Second, in the general case, the state transition rules define a relation between pairs of states because nondeterminism is present in most systems and their environments.

The approach to describing the states and the state transition rules varies from one specification language to another. The notation used in the next section, for instance, is borrowed from set theory (for describing the environmental and the system states) and predicate calculus (for defining the state transition rules). Furthermore, some languages make implicit assumptions about either or both the nature of the states and of the state transition rules; the loss in generality is justified by increased specificity in the handling of a particular application area. As an example, a system that responds to stimuli from the environment in a manner which is independent of the history of previous stimuli and responses may be easily described in a language which equates the state of the environment with the current stimulus, which has no ability to describe system states, and which is able to define a mapping from the set of stimuli to the set of responses. Another example could be used to illustrate the fact that there is also great variability in the way state transitions may be described: in a biomedical simulation system a new state is generated as a result of the integration of a set of differential equations.

If the conceptual model is structured in a hierarchical manner, e.g., CM" = (CM, CM'), then one needs the notion of decomposition defined below. CM' is said to be a decomposition of CM, i.e., CM' REFINES CM, if and only if

given   CM = (E, EO, S, SO, F) and CM' = (E', EO', S', SO', F')

there exists a function PHI such that

```
     a. PHI : (E' x S') --> (E x S)

     b. PHI is onto (E x S)

     c. for-all e0',s0' there-exist e0,s0: PHI(e0',s0') = (e0,s0)

     d. IF   ( ((e1',s1'), (e2',s2')) MEMBER.OF F') AND
               PHI(e1',s1') = (e1,s1)    AND
               PHI(e2',s2') = (e2,s2)    AND
               NOT( (e1,s1) = (e2,s2) )
        THEN ((e1,s1), (e2,s2)) MEMBER.OF F
```

In the case of large systems, where top-down specification of the conceptual model becomes a necessity, this definition establishes a fundamental criterion for checking the self-consistency of the system requirements.

THE APPROACH

This section introduces the reader to a systematic approach to developing formal functional requirements. The conceptual model introduced in the previous section and an informal description of the application are the starting point for this approach. The application considered here is a simplified version of a banking operation:

> EGBANK is a small bank having several branch offices in the city. Tellers from each branch office are authorized to create new accounts, to check the balance of some account, to make deposits and withdrawals from one account at a time, and to transfer money from one account to another. All replies must be complete, i.e., they must include the account number, customer name, and current deposit value for every account accessed by the teller. All queries resulting in successful banking transactions are logged for auditing purposes. The log entries always include the teller identification.

For illustration purposes, these functional requirements are assumed to represent a complete definition of the customer's needs. The way in which they are converted to a conceptual model is outlined below.

Preliminary tailoring of the formal model.

Most applications do not require the full power of the requirements definition model. This explains why in some cases even finite-state machines proved adequate [HENI79]. Early identification of the complexity of the state transition rules may bring about significant savings in the effort involved in generating the requirements. This statement is strongly supported by past experience and may be explained by the fact that, by understanding the exact nature of the transition rules one is better prepared to avoid two opposite but equally time wasting pitfalls: (1) the use of formalisms which are not powerful enough to do the job and (2) the generation of specifications which are unnecessarily complex and whose simplification often turns out to be more expensive than starting from scratch.

Fortunately, a priori determination of the nature of the state transition rules appears to be possible. By analyzing the informal problem definition one may be able to establish that the state transition rule, F, is indeed a function. (In general, nondeterministic behavior suggests the use of a relation rather then a function, i.e., given the current system/environment state there are several possible next states.)

In EGBANK, the state of the environment is given by the nature of the currently pending teller queries. The state of the system is represented by the composite of all bank accounts. State changes in the environment occur due to arrival of new customers which trigger new queries in their behalf and due to arrival of replies to pending queries. State changes in the system take place due to processing of queries which may change the amounts present in various accounts and may create new accounts.

It appears, therefore, that F is not a function and that both E and S are non-finite. While this degree of complexity seems unavoidable, there are still some opportunities for simplifications and they should be investigated. For instance, F may be decomposable into simpler relations or functions.

The suggestion has been made earlier that the state of the environment is determined by the pending queries. An argument could be made, however, that the system is affected not by the pending queries, but by the processing of each new query. Furthermore, the answer to a query is determined by the nature of the respective query and by the state of the system at the time the query is processed (not at arrival time). Consequently, the system actions are captured by the function FS defined below:

```
FS : (Q x S) --> (R x S)

where

Q is the set of possible queries
R is the set of possible replies
S is the set of system states (as before).
```

Each query may be considered as if it were alone in the system because this is exactly the tellers' perception of the system.

As far as the environment is concerned, one may define a relation FE which captures the changes that occur at each teller: the issuing of some query from Q, the return of some answer from R, and the lack of activity which is denoted by "nil". Furthermore, FE indicates that concurrent access to the system is required (without suggesting that concurrent processing, a design issue, is needed) and that queries are not necessarily processed in the order of arrival.

```
FE SUBSET.OF (E x E)

where

  n is the number of tellers

  E = (Q UNION R UNION {nil})**n

  for-all x,y:
        [ ((...,x,...), (...,y,...)) MEMBER.OF FE
          IFF
          ( (x MEMBER.OF Q) AND (y MEMBER.OF R) AND id(x)=id(y) )
          OR ( (x MEMBER.OF R) AND (y = nil) )
          OR ( (x = nil) AND (y MEMBER.OF Q) ) ]
```

The function "id" appearing above will be formally defined later. It was introduced here, however, in order to state that the answer (i.e., y) received by some teller bears the same identification as the original query (i.e., x) sent by the same teller.

At last F may be defined by using FS and FE. (Note: FS, S, Q, and R will be fully defined later.)

```
( (e1, s1), (e2,s2) ) MEMBER.OF F

IFF

(e1, e2) MEMBER.OF FE
AND
IF ( ( e1=(...,x,...)  AND e2=(...,y,...) AND
       (x MEMBER.OF Q) AND (y MEMBER.OF R) )
               THEN ( ((x, s1), (y,s2)) MEMBER.OF FS )
               ELSE s1=s2
```

While it is true that F could have been defined directly in terms of S, Q, R, and n, there are certain advantages to the strategy being presented. In many cases, the attempt to look for a decomposition of F results in much simplified formalizations--not so in our example. More importantly, however, it often leads to a degree of separation of concerns useful in the understanding and analysis of the requirements. In our example, for instance, FS deals with the query processing aspect while FE relates to behavioral aspects of the environment indicating such things as the fact that a reply must succeed the respective query, that queries are not necessarily processed in the order of arrival, etc.

Definition of the environmental states.

The informal requirements specify the nature of the queries (Q) and, indirectly, the nature of the replies (R) that have been used in the high level definition of the environmental states. There are four types of queries: account creation, reading of the account data, updating of the account, and fund transfer between two accounts. Furthermore, each account is generally characterized by the owner's name, by the amount on deposit, and by some account number. By taking advantage of this knowledge and by the fact that each query must have a teller identifier, the set Q may be now defined

```
Q = tellerids x
        ( ({create} x customers x deposits)
          UNION
          ({read} x accounts)
          UNION
          ({update} x accounts x deposits)
          UNION
          ({trans} x accounts x accounts x deposits) )
```

The sets tellerids, accounts, customers, and deposits are left undefined in this paper because their definitions are trivial to construct, not from the earlier informal specification of the problem but by soliciting the missing information. This illustrates one important advantage of the formal specifications. They frequently uncover many important details that were left out in the informal requirements definition.

By refering back to the informal requirements the set of replies, R, is defined as follows:

```
R = tellerids x
        ( {error}
          UNION
          (accounts x customers x deposits)
          UNION
          ( (accounts x customers x deposits)
              x
            (accounts x customers x deposits) )  )
```

At this point, it is easy to deduce the definition of the id function introduced earlier. It simply returns the first element of any n-tuple supplied as its argument.

Definition of system states.

The state of the system is characterized, at any point in time, by the status of all the accounts and by the transaction log. Therefore, the system state might have to be defined in terms of a set that captures the state of the accounts (call it "b" from bank records) and by a sequence that models the log (call it "l").

```
s = (b, l)
```

Next, one could propose b to be described by

```
b SUBSET.OF (accounts x (customers UNION {nil}) x deposits)
```

This definition, however, would permit the undesirable situation where the same account appears twice in the system. Both (1234, Smith, $350) and (1234, Brown, $250) could be part of S. The single account number occurrence condition forces b to be a function from accounts to customers cross deposits:

```
b : accounts --> (customers x deposits)
```

Notation-wise, however, later use of b will be permitted to take both the form of a set (e.g., (a,c,d) MEMBER.OF b) and that of a function (e.g., b(a)), depending on which one is more convenient at the time.

Given the nature of the log which is only a sequence of queries, i.e.,

```
l MEMBER.OF Q*,
```

the set of system states, S, becomes

```
S SUBSET.OF {b | b : accounts --> (customers x deposits)} x Q*
```

This completes the definition of the system states.

Definition of state transition rules.

Because part of the definition was already given earlier in this section, all that remains to be done is to complete the definition of FS which has been established to be of the form:

```
FS : (Q x S) --> (R x S)
```

For the sake of clarity, separate definitions are provided for each of the four types of queries.

The creation of a new account requires that account to be unassigned to any customer. Furthermore, at creation time, both the customer name and some value for the initial deposit must be provided. The account number is supplied by the system.

```
FS((id,create,c,d), (b,l))
   <== if  (c MEMBER.OF customers) AND
           (d MEMBER.OF deposits)  AND d>0 AND
           (there-exist a: ( (a,nil,0) MEMBER.OF b ))
           then
             ((id,a,c,d),
                   ((b MINUS {(a,nil,0)} UNION {(a,c,d)}),
                                   ((id,create,a,c,d).l)))
           else
             ((id,error), (b,l))
```

It should be noted that in case the query is improperly specified, the reply being returned is an error message. Moreover, in accordance with the stated requirements, errors are not logged.

In order to find out information about an account, its number has to be supplied.

```
FS((id,read,a), (b,l))
   <== if  (a MEMBER.OF accounts)  AND
           (there-exist c,d: ((a,c,d) MEMBER.OF b ) AND
                                          NOT( c=nil ))
           then
             ((id,a,c,d), (b,((id,read,a).l)))
           else
             ((id,error), (b,l))
```

While testing the fact that "a" is a valid account is mathematically redundant, it is kept in the definition for clarity purposes. (This issue often causes long discussions during requirements reviews but it is the author's strong conviction that clarity must come before brevity.)

Both deposits and withdrawals are accomplished via the update query. It depends upon the sign of the amount being supplied by the teller. An additional condition for the success of the query is that the amount left in the account must be strictly positive.

```
FS((id,update,a,d), (b,l))
   <== if  (a MEMBER.OF accounts)  AND
           (d MEMBER.OF deposits)  AND
           (there-exist c,d0: ( (a,c,d0) MEMBER.OF b ) AND
                              NOT( c=nil )  AND  (d0+d > 0) )
           then
             ((id,a,c,d0+d),
                   ((b MINUS {(a,c,d0)} UNION {(a,c,d0+d)}),
                         ((id,update,a,d).l)))
           else
             ((id,error), (b,l))
```

The transfer query, called trans, allows one to transfer funds between two accounts. Its meaning is analog to that of removing a positive amount from the first account followed by a deposit in the second account.

```
FS((id,trans,a1,a2,d), (b,l))
   <== if  (a1 MEMBER.OF accounts)  AND
           (a2 MEMBER.OF accounts)  AND
           (d MEMBER.OF deposits)   AND (d>0) AND
           (there-exist c1,d1: ( (a1,c1,d1) MEMBER.OF b ) AND
                                NOT( c1=nil ) AND (d1-d > 0) ) AND
           (there-exist c2,d2: ( (a2,c2,d2) MEMBER.OF b ) AND
                                               NOT( c2=nil ) )
           then
             ((id,a1,c1,d1-d,a2,c2,d2+d),
                   ((b MINUS {(a1,c1,d1)} UNION {(a1,c1,d1-d)}
                      MINUS {(a2,c2,d2)} UNION {(a2,c2,d2+d)}),
                          ((id,trans,a1,a2,d).l)))
           else
             ((id,error), (b,l))
```

The entire specification is complete. Its adequacy still needs to be established through reviews involving the intended user or customer.

Because increases in the complexity of the items being described result in more complex definitions, the introduction of more powerful notation than the one employed in this paper becomes a necessity. At a minimum, one needs to formulate the definitions in terms of primitive functions which are separately defined at some later point. Ultimately, the designer is led naturally, by the need for clarity and simplicity, to developing hierarchical specifications.

DISCUSSION

The approach described in this paper has grown out of the experience gained in the last four years during which formal requirements have been defined for a large variety of relatively small problems. These include: the semantic definition of numerous toy languages, the specification, at several levels, of the message handling subsystem for a local communication network, the definition of the communication primitives for a proposed message-based version of Pascal, and some data processing applications comparable to the example used in the previous section. Involvement in the development of requirements for some real production systems using a partially-formalized technique described in [ROMA79] also resulted in better understanding of what is pragmatically feasible with regard to the role in production of formal requirements.

It is the contention of this paper that the approach is ready for use in small to medium size data processing and real-time systems. Nevertheless, special consideration must be given to the nature of the problem being considered, the background of the personnel involved, the formalism being contemplated, and to the balance between the formal and the informal components of the requirements to be produced.

The introduction of formal requirements into an organization can be neither sudden nor complete. A wise first step is to employ formal requirements only for the hard-to-define aspects of the system requirements in conjunction with some other less formal but already familiar technique. Thus, the few designers who happen to have the appropriate formal background may be utilized effectively and the initial learning curve has a minimal effect on the overall project performance.

Furthermore, the combined use of formal and informal specifications appears to be not just a transient solution meant to bring about the widespread use of formal specifications but a highly desirable property of requirements definitions in general. Experience has shown that, even when the requirements are completely formalized and the people involved have above average mathematical skills, the absence of an accompanying informal narrative greatly increases the review time and reduces their effectiveness. Finally, one other important factor affecting the success of a formal specification is the use of standard notation. Future use of computer-aided design tools will make this issue obsolete but, until then, lack of standardization may result in misunderstandings and confusion. (Our policy has been to stay with basic mathematical notation. However, the use of a standard keyboard character set has resulted in some compromises.)

One issue that has been ignored throughout most of this paper is the formal definition of the system constraints, i.e., non-functional requirements. So far, the formal requirements we developed centered on rendering the system functionality (the conceptual model) and allowed the constraints to be formulated through the use of natural language. However, attempts to specify formally some of the constraints have been made by others in conjunction with work on specification languages [ALFO79, ZAVE81] and in the area of system modeling and theory [WYMO67]. Making some of these results accessible to the general practitioner is a task for the future.

More work is also required to establish techniques for checking the self-consistency, completeness and accuracy of the requirements. Ad-hoc mathematical proofs and group reviews proved adequate for small scale problems but are not expected to be cost effective and accurate enough for large projects where the use of computer-aided design tools able to carry out some of these checks and proofs becomes a necessity.

CONCLUSIONS

A rigorous approach to the development of formal system requirements definitions has been presented in a tutorial fashion and illustrated on a simple example of a banking system. The approach reflects the author's several years experience developing formal requirements for a variety of small scale problems. The notation used in the paper is based on set theory and predicate calculus both of which are generally considered essential in the education of today's computer scientist and are familiar to many system designers. The paper contends that, based on the experience accumulated with the use of both formal and semi-formal specifications, the development of formal requirements for small to medium size systems is feasible and can be cost effective.

REFERENCES

[ALFO79] Alford, M., "Requirements for Distributed Data Processing Design," *Proc. 1'st Int. Conf. on Distributed Computing Systems*, pp. 1-14, October 1979.

[BELL77] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, pp. 49-60, January 1977.

[GREI77] Greif, I., "A Language for Formal Problem Specification," *CACM* 20, No. 12, pp. 931-935, December 1977.

[HENI79] Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," *Proc. Conf. on Specifications of Reliable Software*, April 1979.

[RIDD78] Riddle, W. E., et al, "Behavior Modeling During Software Design," *IEEE Trans. on Soft. Eng.* SE-4, No. 4, pp. 671-678, July 1978.

[ROMA79] Roman, G.-C., "Verification Procedures Supporting Software Systems Development," *Proc. of 1979 NCC*, pp. 947-956, June 1979.

[ROSS77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, pp. 16-34, January 1977.

[TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. on Soft. Eng.* SE-3, No. 1, pp. 41-48, January 1977.

[TSIC82] Tsichritzis, D. C. and Lochovsky, F. H., *Data Models*, Prentice-Hall, Inc. 1982.

[WYMO67] Wymore, A. W., *A Mathematical Theory of Systems Engineering--The Elements*, John Wiley and Sons, Inc., 1967.

[ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems," *Proc. 5'th Int. Conf. on Soft. Eng.*, pp. 295-304, March 1981.

NOTATION SUMMARY

n-tuple	(a, b, ...)

set definition	{a, b, ...}
	{x ¦ predicate(x)}

function definition	fname: domain --> range
	fname(arguments)
	<== if predicate then value1
	else value2

quantifiers	
existential	(there-exist x: predicate)
universal	(for-all x: predicate)
cross product	set1 x set2
n'power cross product	set**n
logical implication	IF predicate1 THEN predicate2
logical operators	AND, OR, NOT
subset test	set1 SUBSET.OF set2
set membership test	element MEMBER.OF set
set operations	UNION, INTERSECTION, MINUS
set of all sequences over some set	set*
concatenation	sequence1.sequence2
special constant	nil