

# An Examination of Requirements Specification Languages\*

T.H. Tse and L. Pong\*\*  
Department of Computer Science  
The University of Hong Kong  
Pokfulam Road  
Hong Kong

## ABSTRACT

We examine the features which are most desirable in requirements specification languages, and then use the framework to review six established languages: PSL, SADT, EDDA, SAMM, HOS and RSL.

**Keywords and phrases:** Requirements specification, specification languages, systems development

**CR Categories:** D.2.1, F.3.1, K.6.3

## 1. INTRODUCTION

A requirements specification for an information system is important for several reasons: It serves as a means of communication between the user and the systems developer; it represents in a systematic fashion the current state of the real world, its problems and its future requirements; it enables the systems developer to turn real world problems into other forms which are more manageable in terms of size, complexity, human understanding and computer processability; it serves as the basis for the design, implementation, testing and maintenance of the target system. In order that all the objectives of a requirements specification be met, we need a powerful specification language. Quite a number of authors (such as [9, 21, 13, 36, 38]) have proposed independently the desirable features of requirements specification languages. In Section 2 of this paper, we consolidate these features and present them in the context of an engineering process. Then, in Section 3, we use the proposed features as the basis of an examination of some of the established research in specification languages.

## 2. DESIRABLE FEATURES OF A REQUIREMENTS SPECIFICATION LANGUAGE

Information systems development can be conceived of as an engineering process. A graphical representation is shown in Figure 1. We must first of all build a model, which is a small-scaled abstract representation of the real world. All unnecessary details in the physical world which are irrelevant to the engineering process are removed from the model, or in other words, ignored during

---

\* Part of this research was done at the London School of Economics, University of London under a Commonwealth Academic Staff Scholarship. The research was also supported in part by a Hong Kong and China Gas Research Grant and a University of Hong Kong Research and Conference Grant.

\*\* Currently with IBM, Canada.

the analysis stage. When a bridge or tunnel is planned between two nations, for instance, the political issues should best be dealt with by politicians and not form part of the engineer's requirements specification.

If the resulting model is still too complex, further abstractions may be necessary, until the problem is reduced to a manageable size. The model is then analysed and manipulated until a feasible solution is found. In engineering, diagrams and mathematics are often used because they have been found to be more suitable for manipulation than verbal descriptions. One representation may have to be transformed into another so that the most appropriate model for a given analysis can be used. Diagrams, for instance, may have to be converted into equations. Finally, if the abstract solution is accepted by the customer, a construction phase turns it into a real system.

In order for a requirements specification to be useful in systems development, seen as an engineering process, the specification language must exhibit various features, each being relevant to one of the stages. These features will be highlighted in this section.

We recognise that there are authors who may object to having an engineer's view of information systems development. Examples are Checkland, Land and Mumford [5, 18, 23], who regard systems development as a human activity process. They propose that emphasis should be made on such issues as understanding the impacts of change, people-oriented design and user participation throughout the development process. It would be interesting to study how our proposal for the desirable features of a specification language fits into this alternative framework. It is, however, beyond the scope of the present paper to do so.

## **2.1 ABSTRACTION OF THE REAL WORLD**

A requirements specification language is the means by which users can make a model of the real world and specify its problems and requirements. It is the bridge between a development environment and the users, including systems analysts, designers and end users. We must ensure that this interface is understandable to all concerned. The usual marketing phrase "user-friendliness" is a bit too vague to act as a useful guide. Instead, we consider it essential for the language to have the following properties:

### **2.1.1 USER FAMILIARITY OF THE SPECIFICATION LANGUAGE**

It would be difficult for users to employ an entirely new requirements specification language because of several reasons.

- (a) There is an inertial effect from the point of view of users. They are not willing to try a new method with which they are not familiar.
- (b) From the management point of view, a well-tested methodology which has proven popularity tends to be more acceptable than a newly proposed technique. It is more easy to recruit staff members who are trained and experienced in an established method. It would be easier to maintain standards if the same methodology were used throughout the company. Managers in general find it safer to be old-fashioned than to try the latest innovation and regret it afterwards.

As a result, practitioners are rather hesitant to use new tools which involve an unfamiliar formal language [10, 20, 21]. When we propose a new systems development environment, therefore, we should not be inventing an entirely new language, with the hope that it will turn out to be the best in the world. Instead, we should try to use a currently available specification language which has most of the desirable features and, more importantly, has proven popularity among practitioners.

### 2.1.2 LANGUAGE STYLE

#### (a) Textual Language

When we consider the use of a textual language for requirements specifications, two possibilities arise: We may like either a natural language or a more formal programming-like language. There is little doubt that natural languages give a better persuasive power and more freedom of expression, especially in the initial phases of the system life cycle when a degree of uncertainty is involved. It is also more natural to the average end user and hence improves the user-understanding of a new situation. Research workers in artificial intelligence, however, are still trying very hard to make natural language understood by computer systems, and hence it is impossible currently to find a development environment which supports requirements specifications based on natural language.

On the other hand, requirements specified in a natural language may cause ambiguities, since they may be interpreted differently by different people. As pointed out by DeMarco [11] and others, standard English prose is not suitable even for specifications which are processed manually. Languages that have a better defined syntax and slightly more restrictive semantics would therefore be preferred. These languages are more formal in nature and resemble a programming language or a mathematical language.

#### (b) Graphical Language

It is generally agreed that graphical representation of complex material is much more comprehensible than its textual counterpart. The reasons can be summarized as follows:

- Graphics is in two dimensions while text is in one dimension. The former gives an additional degree of freedom in presentation.
- Graphics is more useful in showing the hierarchical structure of complex systems and more natural in describing parallelism.
- A person reading graphics can do so selectively, depending on the level of details required. If he reads text, he has to do so linearly.
- There is a limit to the number of concepts which can reasonably be held in the short-term memory of human mind [22]. A person reading graphics can start off generally and go down to detail after some degree of familiarization. If one is reading text, then one has to start off with the detail and abstract the skeleton concepts afterwards.

It should be noted, however, that graphical languages with too many symbols are not necessarily comprehensible to end users. The graphics-based language must consist only of a relatively small number of easily understood symbols.

### (c) Hybrid Languages

It is often impractical to define a requirements specification only in terms of a single graphical or textual language. For example, although graphical languages are better than textual languages in presenting an overview of a complex situation, textual languages are regarded as better tools for detailed description. Although formal languages have precise semantics, their meanings are often explained through natural languages. We need, therefore, a language which exists in more than one format. A reasonable subset of the specification, at least at the higher levels of abstraction, must exist both as formal and informal versions. The specification must be convertible from one form to another so that users, analysts, designers, implementors and managers can communicate effectively. A person needs only to review the version most appropriate to his understanding and his needs. A formal one-to-one correspondence must be maintained among the various syntaxes of the specification language, so that there will not be any error or dispute.

## 2.1.3 COMPLEXITY

As suggested in [39], complexity is the main barrier to the understanding of system problems. A requirements specification language should therefore provide a means to improving the conceptual clarity of the problems. There are two ways of attaining this goal.

### (a) Separation of Logical and Physical Characteristics

When we analyse the requirements of a system, we should distinguish between logical and physical characteristics. By logical characteristics we mean the essential features of the system which must be satisfied in order to meet the users' requirements, irrespective of the actual procedure or mechanism. They are also known as essential characteristics. By physical characteristics we mean the way in which the system actually functions in the real world. A typical example of the latter is a process which employs some specific tool or material, such as photocopiers or microfilms. In another example, we may encounter two independent tasks with no data passing from one to the other, and yet they are performed in a definite sequence. Such an order of events is more likely to be due to political or historical issues not relevant to the system.

A requirements specification language must allow us to model logical and physical characteristics separately, in order to differentiate important issues from non-essentials. For example, when we analyse a control system for operating a lift, we may like to exclude such physical features as colour, decor and lighting from our logical model. In this way, the logical model for the lift control system may be found to be equivalent to that of a disk control system, so that any expertise in one design can be applied to the other.

### (b) Multi-Level Abstraction

A hierarchical framework should be provided by the specification language to enable users to visualize a target system more easily. It allows users to start conceptualizing the system at the highest level of abstraction, and then to continue steadily downwards for further details. It allows users to distinguish those parts of the target system relevant to a given context from those which are not. Such a top-down approach frees users from embarking on unnecessary detail at a time when they should be concerned only with an overall view of the target system. On the other hand, it allows certain parts of a system to be investigated in detail when other parts are not yet defined.

To enable users to relate a requirements specification to real life problems, the specification language should help them to refine the target system in a natural way. Target subsystems must be as self-contained as possible. The interface between any two subsystems should be kept to a minimum but defined explicitly. This will reduce complexity, avoid misunderstanding, and make it possible for two subsystems to be analysed further by two different people. System modules created in this way can be reusable common components which are independent of the parent modules that call them. Furthermore, sufficient mechanism should be build into the development environment to help users to check whether each refinement step is logically valid, so that each group of child modules exactly replace the relevant parent module.

### 2.1.4 MODIFIABILITY

The requirements specification must be structured in such a way that the parts of a target system can be modified easily to cater for new user needs, technological updates and/or other changes in the environment. It should cater not only for amendments by the original specifier, but also allow modifications to be made by anyone who is assigned the task.

## 2.2 MANIPULATION OF REPRESENTATIONS

### 2.2.1 TOOLS FOR MANIPULATION

#### (a) Formalism

In order to eliminate the problems of ambiguity during the construction and implementation of a target system, the requirements specification should be expressible in a precise notation with a unique interpretation. A formal framework must, therefore, be present. It helps to reduce the probability of misunderstanding between different designers. At the same time, automated tools based on the formal framework can be used to validate the consistency and completeness of the specification. Furthermore, given the complexity and scope of present day systems, manual development and maintenance methods are highly ineffective. The implementation and maintenance phases can be made “computer-aided” more easily if a formal framework is present.

**(b) Rigour**

Systems development, as it is being practised at the present, lacks any theoretical background and is seen by many as a black art [7]. In other engineering disciplines, the engineers can provide users with a guaranteed degree of confidence by supporting practice with theory, hence ensuring that the product is error free. It is, on the other hand, impossible for a systems developer to do so [26]. To solve the problem, the systems development process must be supported by theory, so that the correctness of implementation can be proved and verified against the user requirements. A variety of theoretical techniques are already available in computer science for the verification of software correctness. We must ensure that the requirements specification language is supported by a mathematical foundation so that it can be mapped on to the appropriate theories.

**2.2.2 TRANSFORMATION****(a) Support of Different Development Situations**

It has been found [8, 19, 31] that different models are needed for different development situations depending on the environment, emphasis and stage of development. For example, a hierarchical chart may be useful as an overview of the target system. Another model showing algorithmic details may be more appropriate for implementation purposes. A third model in a mathematical form may be used for proving the correctness of the implementation. Thus the requirements specifications may be transformed from one style or notation into another. This raises the problem of determining whether the models are in fact equivalent in semantics, and is another reason why the models must be supported by rigour.

**(b) Transparency of Formalism**

In order to support the manipulation and construction phases of the engineering process, a requirements specification usually consists of a substantial amount of formalism or jargon not fully understood by end users. The formal and mathematical aspects of the specification language must be transparent to users because most people who are not mathematically trained feel infuriated if they are presented with a list of Greek and Hebrew symbols. A unified mathematical framework must, however, be present in the language so that it is possible to guarantee a one-to-one correspondence between the formal side and the user-friendly side of the same language. Otherwise it is impossible to ensure, for example, that a hierarchical chart presented to the end user is the same as the algorithmic description read by the implementor.

**2.2.3 INDEPENDENCE OF DESIGN AND IMPLEMENTATION**

A requirements specification should be independent of the design and implementation of the target system. The supporting language must be behaviour-oriented and non-procedural. In other words, it should help us to spell out “what to do” rather than “how to do it”. The resulting specification must be open-ended in terms of implementation, rather than imposing a specific design choice such as the algorithms, files or databases to be used. It should not depend on the proposed hardware or other resources that are subject to change. It must not cause any obstacle to the introduction of new technologies.

## 2.3 CONSTRUCTION OF REAL SYSTEM

A requirements specification can be seen as a contract between the user and the systems developer. During the construction of the target system, we must have an independent means of deciding whether the developer has fulfilled the contract. Elements in the specification should be traceable to the final design and implementation. In other words, suitable cross-referencing mechanism must be provided in the specification language to help us to verify the consistency between the specified system and the implemented version. This enables us to verify that all the functions and constraints specified are actually addressed by the developed system. It also ensures that operations in the final system are attributed to requirements made in the original specification. Furthermore, in the case where implementation is achieved through the help of automatic code generators, such as the Information Engineering Facility, the cross-referencing mechanism must also be automated.

## 3. REVIEW OF REQUIREMENTS SPECIFICATION LANGUAGES

In this section we review six requirements specification languages, namely PSL, SADT, EDDA, SAMM, HOS and RSL, which have been selected for study for the following reasons:

- (a) They were pioneers in requirements specification languages with supporting environments meant to cover the entire systems life cycle, and have stood the test of time.
- (b) They are better known and published, so that interested readers can obtain further information easily.
- (c) They are still active in development and recent enhancements have been reported.
- (d) The languages examined cover a wide spectrum of characteristics. Some of the languages were developed originally as manual tools, but others were meant to be supported by automatic systems from the very beginning. For some languages, a system can be specified in a hierarchical manner, but not for others. Some languages use graphics as the specification medium while others are purely textual.
- (e) The final reason is a pragmatic one. We have included some of the languages which are more familiar to the present authors.

We shall use a common example to highlight the properties of these languages. It must be a fairly simple one to help us to restrict the length of the current paper. The example chosen is the specification of a customer order system, with which most readers are familiar, so that no further explanation is required. Readers interested in more complex examples may consult the references suggested in the respective subsections.

### 3.1 PROBLEM STATEMENT LANGUAGE (PSL)

PSL [34, 33] was developed by the ISDOS project at the University of Michigan and is currently a commercial product marketed by ISDOS Inc. It was part of the first major project in defining a requirements specification language formally and analysing it automatically. PSL statements cover system structure, data flows, data structures, system behaviour and project management. Their syntaxes and semantics are formally based on the entity-relationship approach [6]. (This appears, however, to be an afterthought since the entity-relationship model was not available when PSL was first introduced.) Descriptive comments in natural English can also be included to enhance readability. An example of PSL is shown in Figure 2.

PSL supports multi-level refinement, so that systems can be specified in a hierarchical manner. The logical characteristics of systems are separable from the physical ones. The specifications are independent of design and implementation, and can be traced to the target systems. Statements can be modified without major amendments to the rest of a specification. Although PSL can only be input as a textual language, a set of graphical reports can be generated automatically by the corresponding development environment for user verification. Unfortunately, users must go back to source PSL statements for any modification. Furthermore, there is no guarantee of a one-to-one correspondence between PSL statements and the graphical counterpart.

PSL was not designed for any particular systems development methodology. When attempts were made to use PSL as the front-end language in System Optimization and Design Algorithm [24, 25], it had to be modified to suit the environment. A META/GA system [34, 37] was subsequently developed to solve the problem. Instead of using one standardized PSL for all applications, the formal description of a tailor-made PSL may be defined for a given development methodology and input to the META system. Systems descriptions can then be formulated in that particular PSL. This approach has been tested on structured methodologies [11, 16, 40], and the results are encouraging. The main user-interface, however, is still formal.

### 3.2 STRUCTURED ANALYSIS AND DESIGN TECHNIQUE (SADT)

SADT [12, 30] has been developed by SofTech Inc. A specification is made up of a hierarchy of SADT diagrams, each of which is a network of boxes representing activities, as shown in Figure 3. The arrows at the four sides of each box represent Input, Output, Control and Mechanism for the activity involved. The activities and input/output data can be decomposed in a top-down fashion according to strict syntactical and semantic rules, so that a multi-level specification can be defined. An indexing scheme is provided so that the relationships of boxes and arrows at different levels can be traced easily.

A natural language or an artificial language for a particular application can be embedded into this graphical framework. The two languages together constitute the specification medium for that application. In this way, the embedded language cannot be used in an arbitrary fashion but must follow definite guidelines, hence reducing the effects of potential errors.

SADT provides a graphical means of refining problems and expressing solutions. A specification is modifiable and can be traced to the target system, but is quite dependent on the process design of the final system. Although SADT provides systems analysts with a useful visual aid, the large number (about 40) of primitive constructs are definite obstacles to user-understanding. Moreover, the concept of Mechanism may mislead analysts to deal prematurely with physical implementation issues.

In spite of the complex nature of its graphical notations, SADT was only designed for manual systems development. There is no underlying formalism. Any language can be used as the embedded language. Rules to analyse the consistency and completeness of a specification are not provided. “Obvious” requirements are allowed to be omitted. As it stands, there is no guarantee of a smooth transition to automatic development environments.

### 3.3 EDDA

EDDA [35] is an attempt to enhance SADT to include a mathematical formalism, so that specifications can be analysed automatically for their static and behavioural properties. EDDA exists in two forms: the graphical form G-EDDA is for human understanding and the symbolic form S-EDDA is for computer processing. The former is in fact identical to SADT. An example of the latter is shown in Figure 4. Since the two forms have corresponding syntaxes and identical semantics, one can easily be transformed into the other. An extended Petri net [28, 29] is used as the mathematical model for the formal semantic definition. Transitions in Petri nets correspond to activities in SADT, and places in Petri nets correspond to data items. To support the complex structure of SADT, EDDA extends the Petri net concept to include predicates and coloured tokens at transitions and places. Delay times and probabilities of activation are also included to help with analysing the behavioural properties of target systems.

EDDA is an attempt to add a formalism to an existing graphical language. It has all the positive characteristics of SADT. In addition, it has a textual representation which is in a one-to-one correspondence with a graphical representation. The language is formal with a mathematical model transparent to users. Specifications are modifiable and can be traced to the target systems, but like SADT, they are quite dependent on the process design of the final systems. EDDA has three major drawbacks:

- (a) It is restricted to SADT and cannot be linked to other structured methodologies.
- (b) Not every one of the 40 features of SADT has a Petri net counterpart. Even where such a correspondence exists, the large number of concepts and notations hinders user-understanding.
- (c) Even where a user is familiar with SADT, he must learn an entirely new textual language S-EDDA because the latter serves as the input medium for the development environment.

### 3.4 SYSTEMATIC ACTIVITY MODELLING METHOD (SAMM)

SAMM [17, 27, 32] has been developed by Boeing Computer Services Co. The language construct is a combination of graphics and graph-theoretical notions, with the aim that the resulting language can be machine-processed. A specification consists of a context tree, activity diagrams and condition charts. A sample context tree is shown in Figure 5(a). As the name suggests, it functions as a table of contents for activity diagrams, expressed in a hierarchical form. A sample activity diagram is shown in Figure 5(b). It specifies the relationships between activities and data flows. Although similar to SADT diagrams in appearance, the activity diagram does not show Controls or Mechanisms. Details of activities and data are further specified through data tables and condition charts (Figures 5(b) and (c)). The former spells out the descriptions and structures the input/output data involved. The latter describes the input and activity requirements for the production of the outputs.

Each element of SAMM can be formulated mathematically. The mathematical models behind context trees and activity diagrams are labelled trees and directed graphs, respectively. A total systems description is thus a tree structure whose nodes are flowgraphs. In this way, the theory of trees and graphs can be applied to analyse a specification. The consistency of the specification, for instance, can be verified by checking whether each activity diagram is a connected graph and whether each output state is reachable from some input state through a finite number of paths.

SAMM supports graphical representation and multi-level refinement. The logical characteristics of target systems can be separated from physical characteristics. The specifications are independent of design and implementation, can be modified easily, and can be traced to the target systems. Not much consideration, however, has been given to users who would like some form of textual input, especially for specifying the lowest level requirements of a system.

### 3.5 HIGHER ORDER SOFTWARE (HOS)

HOS [14, 15] is a requirements specification language developed by Higher Order Software Inc. to support the automation of the HOS methodology, also known as the functional life cycle model. The language was known originally as AXES but the name HOS has now been popularized by James Martin [20, 21].

HOS is designed to support the entire systems development process and to generate a provably correct systems design. It is based on a formal model constructed on a set of mathematical axioms. It represents a system by a binary tree called a control map, as shown in Figure 6. Each module in the system is a mathematical function, and is shown as a node in the binary tree. Input and output of the modules are represented by the domains and co-domains of the functions. The tree also describes how functions are decomposed into sub-functions. Methods of decomposition are defined mathematically. A development environment [1, 15] has been implemented to handle the decomposition and the code generated can be proved mathematically to be correct. In this way, multi-level specification is formally supported. The tree structure can be further transformed into a dynamics graph, which is useful for understanding the behavioural properties of the target system.

HOS exists in two equivalent forms — graphical and textual. There is a one-to-one correspondence between the two representations, thereby allowing one representation to be converted into the other or traced against the other. The logical characteristics of systems are separable from physical ones. The specifications are independent of design and implementation, and easily modifiable. The language is formal and mathematically based. Unfortunately, although it attempts to hide the mathematics from users, the result does not appear natural. For example, when the input and output of a process are specified, the standard convention of mathematical functions is followed. As a result, the input definition appears on the right hand side of a box and the output definition appears on the left hand side, so that misunderstanding may arise.

### 3.6 REQUIREMENTS STATEMENT LANGUAGE (RSL)

RSL [2, 3, 4] has been developed by TRW Defence and Space Systems Group. It specifies software requirements in terms of processing paths, each of which represents a sequence of processing steps connecting the arrival of an input message (or stimulus) to the generation of the appropriate output message (or response). Each processing step is known as an Alpha.

The processing paths and steps are represented in a graphical form known as requirements nets or simply R-nets, as shown in Figure 7. In order to support stepwise refinement in systems development, the description of any Alpha in an R-net can be replaced by a number of lower level Alphas. Unlike SADT, however, the designers of RSL prefer the final documentation to appear in only one layer rather than as a hierarchy of R-nets. Intermediate steps of decomposition will not be documented. Only a flat network will be shown in the final form.

RSL can also be represented in a textual form. The statements consists of four basic constructs, namely elements, relationships, attributes and structures. The first three constructs specify the non-procedural aspects of users' requirements in a style similar to the entity-relationship approach. The structures are used to define the behavioural requirements of the system. The graphical and textual representations of RSL are equivalent. The structure statements are in fact the result of projecting R-nets on to a one-dimensional space. In this way, R-nets can be specified explicitly through an interactive graphical tool, or implicitly through the structure statements.

The mathematical formalism behind RSL is transparent to users. The logical characteristics of target systems can be separated from physical characteristics. The specifications are independent of design and implementation, are modifiable and can be traced to the target systems. The major drawback of RSL is that, although it is said to support stepwise refinement, it is not reflected in the resulting documentation. The final specification in its flat form may be incomprehensible to users who have not been involved with the development process.

#### 4. CONCLUSION

We have examined the desirable features of requirements specification languages and used them as the basis for reviewing six established languages. A summary is shown in Table 1. In most of the languages examined, research workers have seen the need to propose a formal and/or mathematical basis for information systems development. They have also proposed the use of graphics, to be supplemented (except in the case of SAMM) by textual languages for defining details. The complexity problem of target systems is also recognized. A hierarchical framework is provided in most of the specifications (except in the case of RSL), and the logical characteristics of target systems are separable from physical ones.

Many of these requirements specification languages are created as a result of studies in formalism, and cause a psychological barrier to end users. Such a deficiency has been recognised in the case of PSL, and hence the supporting development environment has been enhanced to allow for interface with more popular tools such as structured methodologies. The main user-interface, however, is still formal.

Only one project discussed in this paper uses the concept of employing an existing language as the starting point for a mathematical framework. Namely, SADT is chosen as the graphical language for EDDA. Unfortunately, EDDA may, in fact, be hindered by SADT, since the latter is not as popular as other structured tools because of the extreme complexity of its graphical notations. Besides, EDDA accepts only textual input and generates graphics afterwards, and hence the formalism is not transparent to users.

## ACKNOWLEDGEMENTS

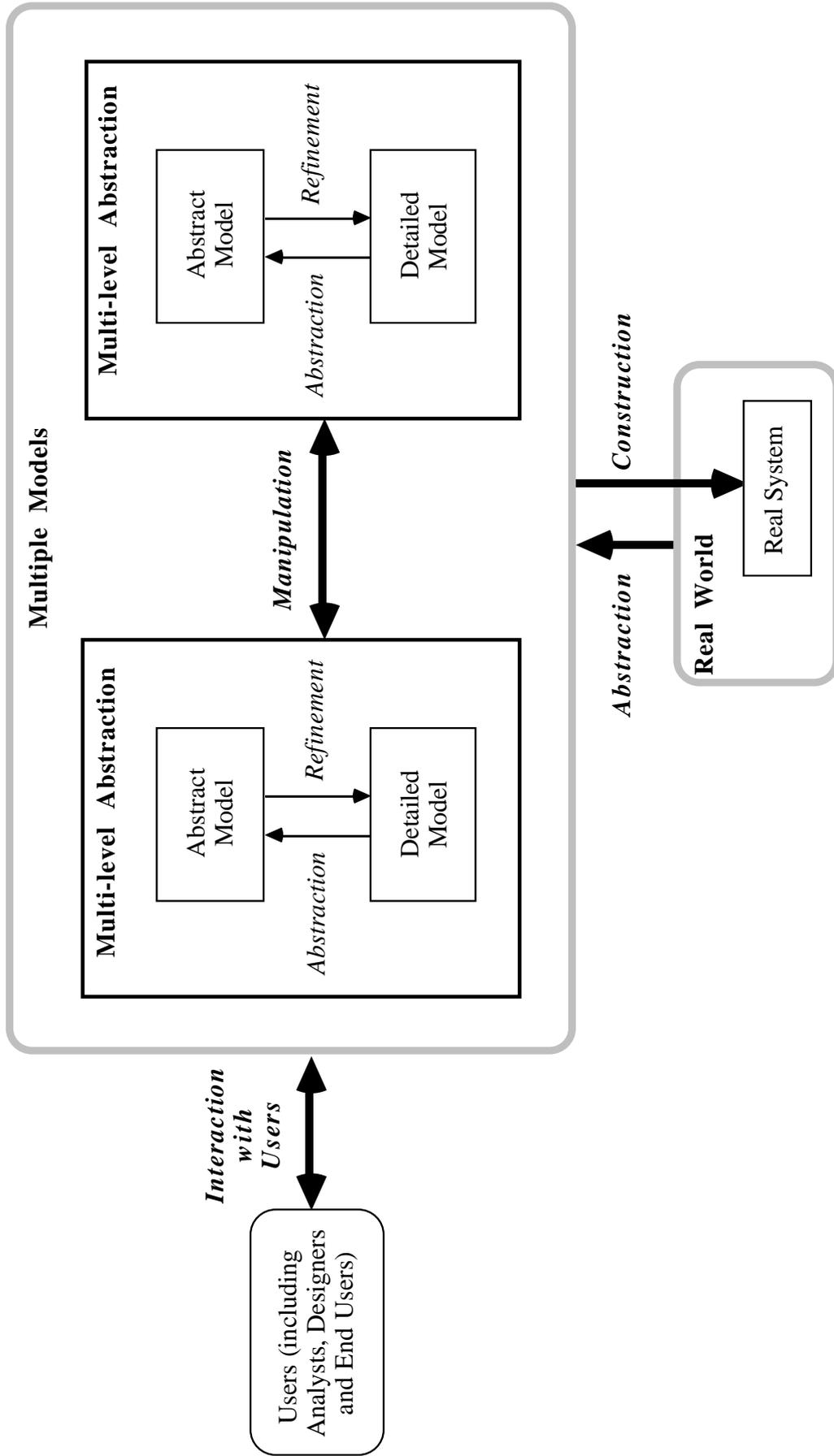
The authors are indebted to Professor Ian Angell, Professor Bernie Cohen, Professor John Campbell and Professor Ronald Stamper for their encouraging comments and suggestions.

## REFERENCES

- [1] *USE.IT Reference Manual*, Higher Order Software, Inc., Cambridge, Massachusetts (1982).
- [2] M.W. Alford, “The software requirements methodology (SREM) at the age of four”, in *Proceedings of the 4th Annual International Computer Software and Applications Conference (COMPSAC '80)*, IEEE Computer Society Press, New York, pp. 866–874 (1980).
- [3] M.W. Alford, “Software requirements engineering methodology (SREM) at the age of two”, in *Advanced System Development / Feasibility Techniques*, J.D. Couger, M.A. Colter, and R.W. Knapp (eds.), Wiley, New York, pp. 385–393 (1982).
- [4] M.W. Alford, “SREM at the age of eight: the distributed computing design system”, *IEEE Computer* **18** (4): 36–46 (1985).
- [5] P.B. Checkland, *System Thinking, System Practice*, Wiley, New York (1981).
- [6] P.P. Chen, “The entity-relationship model: towards a unified view of data”, *ACM Transactions on Database Systems* **1** (1): 9–36 (1976).
- [7] B. Cohen, W.T. Harwood, and M.I. Jackson, *The Specification of Complex Systems*, Addison Wesley, Wokingham, UK (1986).
- [8] M.A. Colter, “Evolution of the structured methodologies”, in *Advanced System Development / Feasibility Techniques*, J.D. Couger, M.A. Colter, and R.W. Knapp (eds.), Wiley, New York, pp. 73–96 (1982).
- [9] M.A. Colter, “A comparative examination of systems analysis techniques”, *MIS Quarterly* **10** (1): 51–66 (1984).
- [10] A.M. Davis, “The design of a family of application-oriented requirements languages”, *IEEE Computer* **15** (5): 21–28 (1982).
- [11] T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey (1979).
- [12] M.E. Dickover, C.L. McGowan, and D.T. Ross, “Software design using SADT”, in *Structured Analysis and Design*, State of the Art Report, J. Hosier (ed.), vol. 2, Infotech, Maidenhead, UK, pp. 99–114 (1978).
- [13] R. Rock-Evans (ed.), *Analyst Workbenches*, State of the Art Report, Infotech Pergamon, Maidenhead, UK (1987).
- [14] M. Hamilton and S. Zeldin, “Higher order software: a methodology for defining software”, *IEEE Transactions on Software Engineering* **SE-2** (1): 9–36 (1976).
- [15] M. Hamilton and S. Zeldin, “The functional life cycle model and its automation: USE.IT”, *Journal of Systems and Software* **3** (3): 25–62 (1983).
- [16] M.A. Jackson, *System Development*, Prentice Hall International Series in Computer Science, Prentice Hall, London (1983).

- [17] S.S. Lamb, V.G. Leck, L.J. Peters, and G.L. Smith, “SAMM: a modeling tool for requirements and design specification”, in *Proceedings of the 2nd Annual International Computer Software and Applications Conference (COMPSAC '78)*, IEEE Computer Society Press, New York, pp. 48–53 (1978).
- [18] F.F. Land and R.A. Hirschheim, “Participative systems design: rationale, tools, and techniques”, *Journal of Applied Systems Analysis* **10**: 91–107 (1983).
- [19] R.J. Lauber, “Development support systems”, *IEEE Computer* **15** (5): 36–46 (1982).
- [20] J. Martin, *Program Design which is Provably Correct*, Savant Institute, Carnforth, Lancashire, UK (1983).
- [21] J. Martin, *An Information Systems Manifesto*, Prentice Hall, Englewood Cliffs, New Jersey (1984).
- [22] G.A. Miller, “The magic number seven, plus or minus two: some limits on our capacity for processing information”, *Psychological Review* **63**: 81–97 (1956).
- [23] E. Mumford, *Designing Human Activity Systems*, Manchester Business School, Manchester (1983).
- [24] J.F. Nunamaker, Jr., “A methodology for the design and optimization of information processing systems”, in *System Analysis Techniques*, J.D. Couger and R.W. Knapp (eds.), Wiley, New York, pp. 359–376 (1974).
- [25] J.F. Nunamaker, Jr., B.R. Konsynski, Jr., T. Ho, and C. Singer, “Computer-aided analysis and design of information system”, *Communications of the ACM* **19** (12): 674–687 (1976).
- [26] D.L. Parnas, “Software aspects of strategic defense systems”, *American Scientist* **73**: 432–440 (1985).
- [27] L.J. Peters and L.L. Tripp, “A model of software engineering”, in *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*, IEEE Computer Society Press, New York, pp. 63–70 (1978).
- [28] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, New Jersey (1981).
- [29] W. Reisig, *Petri Nets: an Introduction*, EATCS Monographs on Theoretical Computer Science, vol. 4, Springer, Berlin (1985).
- [30] D.T. Ross and K.E. Schoman, “Structured analysis for requirements definition”, in *Classics in Software Engineering*, E. Yourdon (ed.), Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey, pp. 365–385 (1979).
- [31] O. Shigo, K. Iwamoto, and S. Fujibayashi, “A software design system based on a unified design methodology”, *Journal of Information Processing* **3** (3): 186–196 (1980).
- [32] S.A. Stephens and L.L. Tripp, “Requirements expression and verification aid”, in *Proceedings of the 3rd International Conference on Software Engineering (ICSE '78)*, IEEE Computer Society Press, New York, pp. 101–108 (1978).
- [33] D. Teichroew and E.A. Hershey III, “PSL/PSA: a computer-aided technique for structured documentation and analysis of information processing systems”, in *Advanced System Development / Feasibility Techniques*, J.D. Couger, M.A. Colter, and R.W. Knapp (eds.), Wiley, New York, pp. 315–329 (1982).

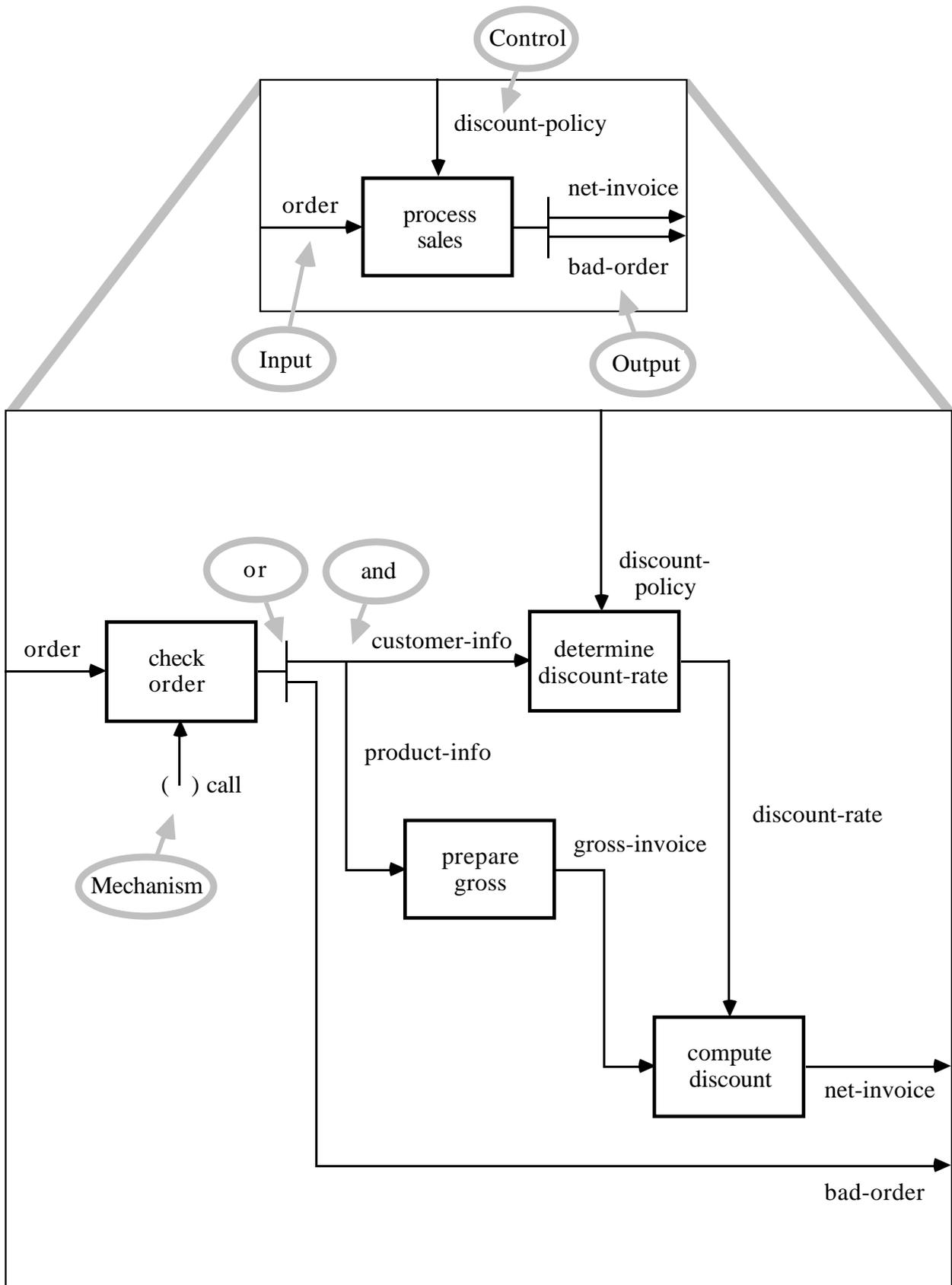
- [34] D. Teichroew, P. Macasovic, E.A. Hershey III, and Y. Yamamoto, “Application of the entity-relationship approach to information processing systems modelling”, in *Entity-Relationship Approach to Systems Analysis and Design: Proceedings of the 1st Conference on Entity-Relationship Approach*, P.P. Chen (ed.), North-Holland, Amsterdam, pp. 15–39 (1980).
- [35] W. Trattnig and H. Kerner, “EDDA: a very-high-level programming and specification language in the style of SADT”, in *Proceedings of the 4th Annual International Computer Software and Applications Conference (COMPSAC '80)*, IEEE Computer Society Press, New York, pp. 436–443 (1980).
- [36] A.I. Wasserman, P. Freeman, and M. Porcella, “Characteristics of software development methodologies”, in *Information Systems Design Methodologies, a Feature Analysis: Proceedings of the IFIP WG 8.1 Working Conference*, T.W. Olle, H.G. Sol, and C.J. Tully (eds.), North-Holland, Amsterdam (1983).
- [37] Y. Yamamoto, *An Approach to the Generation of Software Life Cycle Support Systems*, Ph.D. Thesis, The University of Michigan, Michigan (1981).
- [38] S.S. Yau and J.J.-P. Tsai, “A survey of software design techniques”, *IEEE Transactions on Software Engineering* **SE-12** (6): 713–721 (1986).
- [39] R.T. Yeh and P. Zave, “Specifying software requirements”, *Proceedings of the IEEE* **68** (9): 1077–1085 (1980).
- [40] E. Yourdon, *Modern Structured Analysis*, Yourdon Press Computing Series, Prentice Hall, Englewood Cliffs, New Jersey (1989).



**Figure 1 Schematic Concept of an Engineering Process**

PROCESS: process-order  
 /\* authors T.H. Tse and L. Pong \*/  
 DESCRIPTION:  
 this process captures the details of a valid order, calculates the amounts, discount and net-amount, and prepares an invoice;  
 GENERATES: net-invoice;  
 RECEIVES: valid-order, discount-rate;  
 SUBPARTS ARE: prepare-gross, compute-discount;  
 PART OF: process-sales;  
 DERIVES: amount  
 USING: price, quantity-ordered;  
 DERIVES: total-amount  
 USING: amount;  
 DERIVES: net-amount  
 USING: total-amount, discount-rate;  
 PROCEDURE:  
 1. multiply price and quantity-ordered to obtain amount;  
 2. update stock record accordingly;  
 3. add amounts to obtain total-amount;  
 4. multiply total-amount by discount-rate to obtain net-amount;  
 5. update customer record accordingly;  
 6. generate invoice for net-amount;  
 HAPPENS: 1 TIMES-PER valid-order;  
 TRIGGERED BY: valid-order-event;  
 TERMINATION CAUSES: invoice-event;  
 SECURITY IS: account-clerk-only;

**Figure 2 Example of PSL Specification**



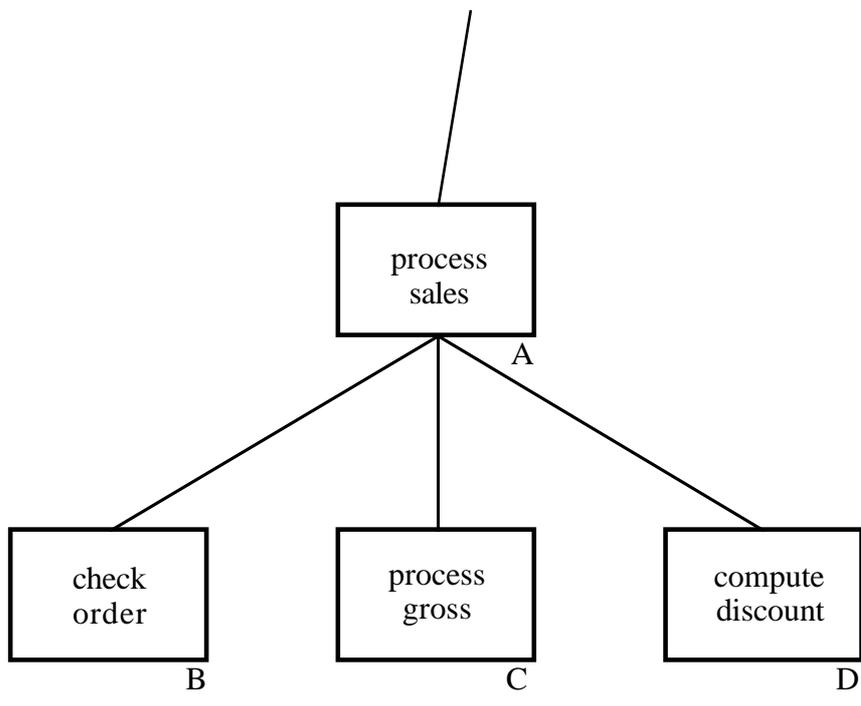
**Figure 3 Example of SADT Specification**

```

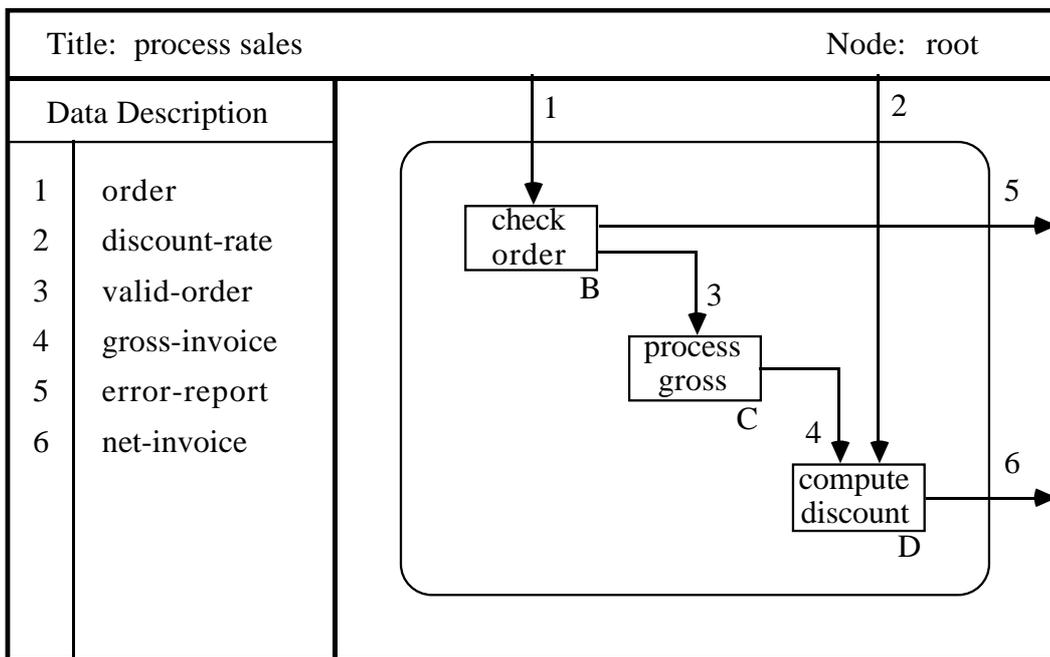
process process-sales (interfaces
  input I1 = order: orders
  control C1 = discount-policy: rules
  output O1 = net-Invoice: invoices);
type
  orders = customer-info sequ product-info
internal data
  product-info: product-array
  customer-info: customer-records
  valid-order: orders
  bad-order: orders
  gross-invoice: invoices
  discount-rate: rates
structure
  xfork order = validOrder + bad-order
subprocesses
  process check-order (interfaces
    input I1 = order: orders
    output O1 = checked-order: orders);
    forward
  process determine-discount-rate (interfaces
    input I1 = customer-info: customer-records
    control C1 = discount-policy: rules
    output O1 = discount-rate: rates);
    forward
  process prepare-gross (interfaces
    input I1 = product-info: product-array
    output O1 = gross-invoice: invoices);
    forward
  process compute-discount (interfaces
    input I1 = discount-rate: rates
      I2 = gross-invoice: invoices
    output O1 = net-invoice: invoices);
    forward
begin
process check-order (I1 = order, O1 = checked-order);
process determine-discount-rate (I1 = customer-info,
  C1 = discount-policy, O1 = discount-rate) <
  process prepare-gross (I1 = product-info,
    O1 = gross-invoice);
  process compute-discount (I1 = discount-rate,
    I2 = gross-invoice, O1 = net-invoice);
end

```

**Figure 4 Example of S-EDDA Specification**



**Figure 5(a) Example of SAMM Context Tree**



**Figure 5(b) Example of SAMM Activity Diagram**

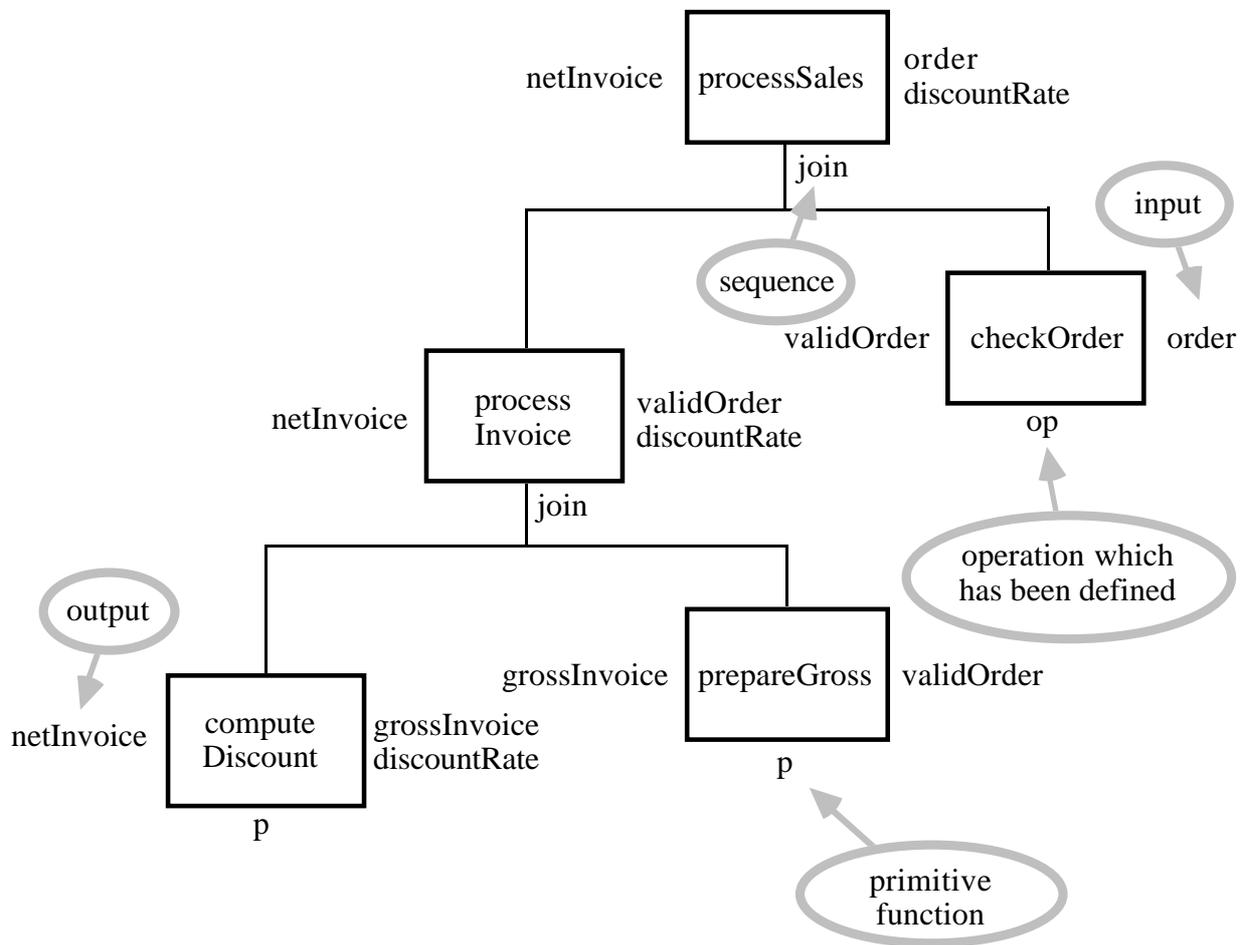
| Title: process sales |                |      | Node: root            |                               |
|----------------------|----------------|------|-----------------------|-------------------------------|
| Output               | Input Required | Cond | Condition Description |                               |
| 3                    | 1              | 1    | 1                     | Process sucessfully completed |
| 4                    | 3              | 1    | 2                     | Error encountered             |
| 5                    | 1              | 2    |                       |                               |
| 6                    | 2, 4           | 1    |                       |                               |

**Figure 5(c) Example of SAMM Condition Chart**

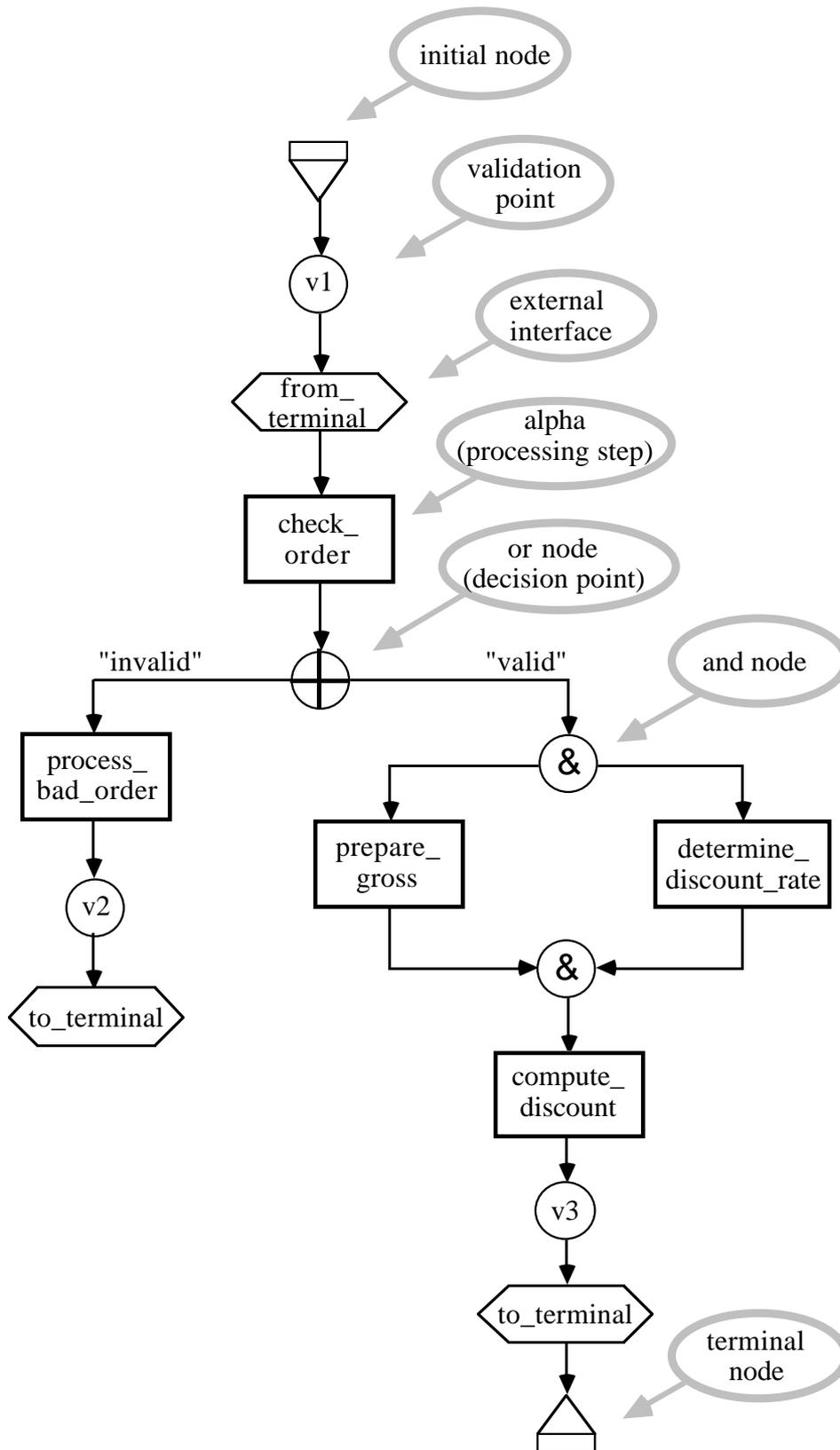
```

.model/processSales
netInvoice = processSales (order, discountRate) [j]
  validOrder = checkOrder (order) [op]
  netInvoice = processInvoice (validOrder, discountRate) [j]
    grossInvoice = prepareGross (validOrder) [p]
    netInvoice = computeDiscount (grossInvoice, discountRate) [p]
.end

```



**Figure 6 Example of HOS Specification**



**Figure 7 Example of R-Net**

|  | PSL | SADT | EDDA | SAMM | HOS | RSL |
|--|-----|------|------|------|-----|-----|
| <b>ABSTRACTION OF THE REAL WORLD</b>                         |     |      |      |      |     |     |
| <b>User Familiarity of the Specification Language</b>        | ×   | √×   | √×   | ×    | ×   | ×   |
| <b>Language Style</b>  |     |      |      |      |     |     |
| • Textual language   | √   | ×    | √    | ×    | √   | √   |
| • Graphical language   | √×  | √    | √    | √    | √   | √   |
| • Hybrid languages   | √   | ×    | √    | √×   | √   | √   |
| <b>Complexity</b>  |     |      |      |      |     |     |
| • Separation of logical and physical characteristics         | √   | √×   | √×   | √    | √   | √   |
| • Multi-level abstraction                                    | √   | √    | √    | √    | √   | √×  |
| <b>Modifiability</b>   | √   | √    | √    | √    | √   | √   |
| <b>MANIPULATION OF REPRESENTATIONS</b>                       |     |      |      |      |     |     |
| <b>Tools for Manipulation</b>                                |     |      |      |      |     |     |
| • Formalism  | √   | √×   | √    | √    | √   | √   |
| • Rigour   | ×   | ×    | √    | √    | √   | √   |
| <b>Transformation</b>  |     |      |      |      |     |     |
| • Support of different development situations                | √   | √×   | √    | √    | √   | √   |
| • Transparency of formalism                                  | ×   | √×   | ×    | √    | √×  | √   |
| <b>Independence of Design and Implementation</b>             | √   | √×   | √×   | √    | √   | √   |
| <b>CONSTRUCTION OF REAL SYSTEM</b>                           |     |      |      |      |     |     |
| <b>Traceability between Specification and Target Systems</b> | √   | √    | √    | √    | √   | √   |

|                |    |                     |
|----------------|----|---------------------|
| <b>LEGION:</b> | √  | Supported           |
|                | √× | Partially supported |
|                | ×  | Not supported       |

**Table 1 A Summary of the Desirable Features of Requirements Specification Languages**