Formal Methods For The Re-Engineering of Computing Systems[*]

X. Liu, H. Yang and H. Zedan
Software Technology Research Laboratory
De Montfort University,
England

Email: {xdl, hjy, hzedan}@dmu.ac.uk

Abstract

We present a short review of formal methods and their use in the re-engineering of computing systems. The paper considers five classes of formal notations and theories, namely *state/model-based, logic-based, algebraic-based, process algebra and net-based* formalisms together with combined formalisms.

Keywords : formal methods, re-engineering, wide spectrum language, real-time systems, refinement, reverse engineering, logic.

Contents

[*]A version of this report appears in the proceedings of the IEEE International Annual Conference on Computer Software and Application (Compsac'97), August 1997.

1 INTRODUCTION

The debate about the use and relevance of formal methods in the development of computing systems has always attracted a considerable attention and is continually doing so. One school of thought (the protagonists) claims that formal techniques offer a complete solution to the problems of system development. Another school claims that formal methods have little, or no, use in the development process (at least due to the cost involved). There is a third view point, that we share, which states that formal methods is *both over-sold and under-used.*

Nonetheless, whatever school of thought one prescribes to, it is important to realise that as the complexity of building computing systems is continually growing, makes a disciplined, systematic and rigorous methodology essential for attaining a "reasonable" level of dependability and trust in these systems. The need for such a methodology increases as "fatal" accidents are attributable to software errors.

In response to this, an intense research activity has developed resulting in the production of formal development techniques together with their associated verification tools that have been successfully applied in forward engineering such systems. For example, assertional methods, temporal logic, process algebra and automata, have all been used with some degree of success.

In the area of reverse engineering, formal methods have also been put forward as a means to

1. formally specify and verify existing systems in particular those already operating in safety-critical applications;
2. introduce new functionalities and/or
3. take advantage of the improvement in systems design techniques.

In this paper we attempt to review a large class of formal methods that have suggested in the re-engineering process of computing systems. We shall also discuss some of their benefits and limitations. But first, it is necessary to lay some terminological groundwork and to consider current practices.

Formal Methods The term *formal methods* is used to refer to methods with sound basis in mathematics. These should be distinguished from *structured methods* which are well defined but do not have sound mathematical basis to describe system functionalities. Formal methods allows system functionalities to be precisely specified whilst structured methods permit the precise specification of systems structure. However, recently, there has been a substantial research activities to

- integrate formal and structured methods, for example the formal specification language Z [1] [60]has been integrated with the structured method known as SSADM and
- extend some formal methods allowing the treatment of non-functional requirements such as timing and probability [10,11,27,46,47,58].

We take the view that a formal method should consist of some essential components: a semantic model, a specification language (notation), a verification system/refinement calculus, development guidelines and supporting tools:

1. The *semantic model* is a sound mathematical/logical structure within which all terms, formulas and rules used have a precise meaning. The semantic model should reflect the underlying computational model of the intended application.

2. *The Specification language* is a set of notations which are used to describe the intended behaviour of the system. This language must have a proper semantics within the semantic model.

3. *Verification system/refinement calculi* are sound rules that allow the verification of properties and/or the refinement between specifications.

4. *Development Guidelines* are steps showing the use of the method.

5. *Supporting tools* involving proof assistant, syntax and type checker, animator, and prototyper.

Formal methods can be applied in two different ways.

1. The production of specifications which are then the basis for a conventional system development. In this case, specifications are used as a precise documentation medium which has the advantages of manipulability, abstraction and conciseness. Consistency checks and automatic generation of prototypes could be performed at this stage with the aid of the associated supporting tools.

2. The production of formal specification, as above, can then be used as a basis against which the correctness of the system is verified or as a basis to derive the verified system through correctness preserving refinement rules. This will give the developed system a degree of certainty and trustworthiness.

Re-engineering The process of *re-engineering* computing systems involves three main steps: *restructuring, reverse engineering* and *forward engineering.* In the present survey, we take the following view:

- *Restructuring.* It is the process of creating a logically equivalent system from the given one. This process is performed at the same level of abstraction and does not involve semantic understanding of the original system.

- *Reverse Engineering.* It is the process of analysing a system in order to obtain and identify major system components and their inter-relationships and behaviours. It involves the extraction of higher level specifications from the original system.

- *Forward engineering.* The process of developing a system starting from the requirement specification and moving down towards implementation and deployment.

In essence the re-engineering model takes the following form:
Re-engineering = *Restructuring* + *Reverse engineering* + *Forward engineering.*
Our objective in this paper is to discuss, analyse and review what role of formal methods can play within this model.

The paper is organised as follows. Section 2 deals with major classification of existing formal methods. In each class, some example formalisms are chosen as candidates. Each formalism is briefly described, together with its major references and pointers in the literature. Section 3 summarises the criteria of the review and produces our results in form of a comparative tables. Section 4 concludes our review.

2 CLASSIFICATION OF FORMAL METHODS

Formal methods can be classified into the following five classes or types, i.e., *Model-based, Logic-based, Algebraic, Process Algebra* and *Net-based (Graphical)* methods. In the following subsections we will briefly discuss each of these approaches.

2.1 Model-based Approach

General A system is modelled by explicitly giving definition of states and operations that transform the system from a state to another. In this approach, there is no explicit representation of concurrency. Non-functional requirements (such as temporal requirement) could be expressed in some cases.

Examples

- **Z** [1] [60]. With the first version proposed in 1979, the Z notion is based on predicate calculus and Zermelo Fraenkel set theory. A Z specification is written in terms of "schemas", each of which contains a signature part which declares items of interest and a predicate part which places a logical constraint on them.

- **VDM** [35] [9] [34]. VDM (the Vienna Development Method) is a formal method for *rigorous* computing system development. It is similar to Z in most aspects, although not as popular as Z. VDM support model composition and decomposition, which facilitate both the forward and reverse engineering a lot.

  Although the semantics and proofs in predicate calculus are complete and rather complete in set theory, the functions, operations, compositions and decompositions in VDM makes its semantic and proof system much more complicated to be "accomplished". Therefore, similar problems happen with Z and VDM: a complete formal semantics does not exist yet, and as the consequence, the automated support tools, such automated prover, do not exist yet. Moreover, lacking of formal semantics will also limit the potentials for automation in the re-engineering approach which adopts Z or VDM as its formal foundation.

  Time is not a part of VDM notation. When trying to apply VDM to real-time domain, novel features have to be added to VDM. VDM also keeps developing: VDM^{++}, as a new version of VDM integrated with object-oriented idea, is a rather mature product now.

- **B-Method** [36] [37] [62]. The B-method uses the Abstract Machine Notation to support the description of the target systems. The most eminent success of B method is that it already has a strong and quite mature tool B Toolkit, to support and automate the development of application systems. The B-Method is "complete" in the sense that it provides abstract machine specification and their proofs, refinements and their proofs, and composition and their proofs. The development method of B matches the typical top-down forward engineering well. A complete development may be performed and recorded. Changes may be accommodated using the replay tools. Refinement, implementation and composition steps have precise notions of correctness and mechanical generation of proof obligations. By animator, test may be performed. The final implementation step may be mechanised for common languages (e.g. C and Ada) and for some specification constructs.

  In B-Method, no guidance is provided regarding (i) design decisions or their recording, (ii) testing or inspection methodology, (iii) presentation of specifications. B toolkit is still evolving, not 'very' mature now. B method has no time feature. Novel feature has to be added when using B for real-time systems. The main users of B are found in UK.

Sample Description Z is described as a sample here.

- **Syntax and Semantics**. The conceptual basis of Z is typed set theory, and the methods is oriented to constructing models. Text and graphical representation are used.

  The basic elements of Z are types, sets, tuples and bindings. There is no universal set to which all elements belong, but a universe of disjoint sets called types (which contain basic types and composite types). A set in Z is an unordered collection of different elements of the same type, and there is a concept of infinite sets supported in Z. A tuple is an ordered collection of elements that are not necessarily the same type. A binding is a finite mapping from names to elements, not necessarily of the same type.

  The main representational form is the "schema" which is a set of bindings depicted in a special "axiomatic box" syntactical form including a signature(or Schema Name) and a property (made up of two parts—the declaration and axiomatic constraints).

  The **semantics** of Z is based on a version of Zermelo-Fraenkel set theory that does not include the replacement and choice axioms.

- **System Specification**. Operations can be specified in several ways in Z. One way is through the use of "axiomatic descriptions", which are unnamed schemas that introduce one or more global variables, and constraints on those variables. Theses specifications are called "loose specifications" by Z practitioners, who stress the use of schemas to specify. A specifier uses these to indicate a function or constant has certain properties without giving it a value.

  A Z specification is basically composed as ordered collections of schema definitions and axiomatic descriptions. There are complex scoping and naming rules, but the most important specification structuring mechanism is called "schema inclusion". The name of a defined schema may be referred to in any other schema of axiomatic description after its definition, but entities within that schema may be referred to only if the schema is included in the signature of the following schema or axiomatic description.

  Analysis of Z specifications usually means performing consistency and completeness checks, which validate the specification for accuracy and completeness, style, feasibility (sometimes called viability, seeing if a system state exists which satisfies the constraints specified in the initial condition), and expected properties. This analysis is performed by review by other specifiers who perform a "walk through" much like a code "walk through". Proofs are also used to analyse a Z specification, which is primarily done by hand, as there is no reliable automated prover for Z because a formal semantics does not exist yet.

- **Assessment**. Z is good at identifying errors that result from misconceptions in the model of a system. Z supports designing through the use of constructing models, and Z does support a refinement approach to developing systems. It is good at determining and specifying relationships between different levels of specification and design. There is also the ability to re-use Z schemas, especially those that are generic. Since the principles and stages of refinement approach in forward engineering are correspondent to those of abstraction approach in reverse engineering, Z can be also competent in being a good formal foundation of a re-engineering approach.

As mentioned before, although the semantics of Z is based on a version of Zermelo-Fraenkel set theory, it is not complete or sufficient for the whole Z notations when including schemas, tuples, binding, etc. So, a formal semantics of Z does not exist yet. As a consequence, the automated support tools, such automated prover, do not exist yet. Moreover, lacking of formal semantics will also limit the potentials for automation in re-engineering which adopts Z as formal foundation.

Time is not a part of Z notation. When trying to apply Z to real-time domain, novel features have to be added to Z. However, because of the rich expressibility of Z, Z has been used in a number of real-time applications, such as timed Z [40].

The main users of Z are found in UK and other European countries. Generally speaking, Z has been applied to a large amount of applications, some of which are rather large-scaled. It is one of the few formal methods that has been proved successful in industrial applications.

In recent years, some forms of improved Z with new technology such as object-orientation has been developed, for example, Z^{++} and Object-Z.

2.2 Logic-based Approach

General In this approach logics are used to describe system desired properties, including low-level specification, temporal and probabilistic behaviours. The validaty of these properties is achieved using the associated axiom system of the used logic. In some cases, a subset of the logic can be executed, for example the Tempura system [46]. The executable specification can then be used for simulation and rapid prototyping purposes.

Logic can be augmented with some concrete programming constructs to obtain what is known as wide-spectrum formalism. The development of systems in this case is achieved by a set of correctness preserving *refinement steps*. Examples of these form are TAM [58] and the Refinement Calculus [57].

Examples

- **ITL** [11] [47] [46] [10]. ITL (Interval Temporal Logic) has been developed in [11] [48]. This kind of logic is based on intervals of time, thought of as representing finite chunks of system behaviour. An interval may be divided into two contiguous subintervals, thus leading to *chop* operator.

- **Duration Calculus** [12] [13]. Duration Calculus was introduced in [12] as a logic to specify and reason about requirements for real-time systems. It is an extension of Interval Temporal Logic where one can reason about integrated constraints over time-dependent and Boolean valued states without explicit mention of absolute time. Several rather large-scale case studies have shown that Duration Calculus provides a high level of abstraction for both expressing and reasoning about about specifications.

- **Hoare Logic** [19]. Hoare Logic has a long history, it may be viewed as an extension of First-order Predicate Calculus [19] that includes inference rules for reasoning about programming language constructs.

  Hoare Logic provides a means of demonstrating that a program is consistent with its specification. Hoare Logic is not capable of specifying a system at high levels, however, it has distinct advantages in the low level specifications. These two features make Hoare Logic a suitable means in the first stage of reverse engineering, i.e. from source code program to an abstraction at very low level. Some research has been done in this area, such as the development of the reverse engineering tool AutoSpec [14] [24].

There is no real-time feature in Hoare Logic. Some extension can be added to make Hoare Logic more suitable for real-time domain. A Real-time Hoare Logic has been proposed [30].

Hoare Logic is one of the mathematical pillars for program verification and formal methods. Hoare Logic and its variants are used in numerous formal methods tools.

- **WP-Calculus** [18]. Weakest Precondition Calculus was first proposed by E. W. Dijkstra in 1976. A *precondition* describes the initial state of a program, and a *postcondition* describes the final state. By using the semantics of predicate logic and other suitable formal logics, WP-Calculus has been proven to be a formalism suitable for reverse engineering of source code, especially at the low abstraction levels.

- **Modal Logic** [42] [15]. Modal logic is the study of context-dependent properties such as necessity and possibility. In modal logic, the meaning of expressions depends on an implicit context, abstracted away from the object language. Temporal logic can be regarded as an instance of modal logic where the collection of contexts models a collection of moments in time. A modal logic is equipped with modal operators through which elements from different contexts can be combined. There are several approaches to the semantics of modal logic, such as 'neighbourhood' semantics. Until now, there is no application of modal logic in software reverse engineering area.

- **Temporal Logic** [56]. Temporal logic has its origins in philosophy, where it was used to analyse the structure or topology of time. In recent years, it has found a good value in real-time application.

  In physics and mathematics, time has traditionally been represented as just another variable. First order predicate calculus is used to reason about expressions containing the time variable, and there is thus apparently no need for a special *temporal* logic.

  However, philosophers found it useful to introduce special temporal operators, such as $\Box$ (henceforth) and $\Diamond$ (eventually), for the analysis of temporal connectives in languages. The new formalism was soon seen as a potentially valuable tool for analysing the topology of time. Various types of semantics can be given to the temporal operators depending on whether time is linear, parallel or branching. Another aspect is whether time is discrete or continuous [41].

  Temporal logic is *state-based.* A structure of states is the key concept that makes temporal logic suitable for system specification. Mainly, the types of temporal semantics include *interval semantics*, *point semantics*, *linear semantics*, *branching semantics* and *partial order semantics* [41].

  The various temporal logics can be used to reason about *qualitative* temporal properties. Safety properties that can be specified include mutual exclusion and absence of deadlock. Liveness properties include termination and responsiveness. Fairness properties include scheduling a given process infinitely often, or requiring that a continuously enabled transition ultimately fire.

  Various proof systems and decision procedures for finite state systems can be used to check the correctness of a program or system.

  In real-time temporal logics, *quantitative* properties can also be expressed such as periodicity, real-time response (deadline), and delays. Early approaches to real-time temporal logics were reported in [51] [5]. Since then, real-time logics have been explored in great detail.

- **RTTL** [49] [50]. RTTL (Real-Time Temporal Logic) uses a distinguished temporal domain, the ESM (Extended State Machine) state variables, and the set of ESM transitions to form temporal formula. These are then proven using an axiomatisation of the system's ESM trajectories.

  RTTL has a complex and non-compositional proof system. All of the ESMs have to be designed before any theorems that may be proved about them. There is a decision procedure for finite ESMs but due to the undecidability of predicate logic, a procedure for infinite ESMs can never be found. There is a method for RTTL, but it is basic and contains informal steps. Time is global and there is no maximum parallelism model.

  Perhaps more importantly, RTTL has a very "expressive" syntax, the user can choose either temporal domain expressions or operators. This flexibility may result in "cleaner" specifications.

  No special development method is proposed in RTTL or required by RTTL. If applied to reverse engineering area, RTTL has a flexibility to fit different methodologies.

- **RTL** [33]. RTL [33] is a real-time logic with four basic concepts: *actions* which may be composite or primitive, *state predicates* which provide assertions regarding the physical system state, *events* which are markers on the (sparse) time line, and *timing constraints* which provide assertions about the timing of events.

  Work is presently being carried out on finding an efficient general decision procedure for RTL formulas, presently it is a time consuming exercise to verify safety and liveness assertions using standard deductive proofs. Also, a design method is mentioned which may provide an environment for the engineering of large real-time systems, Jahanian and Mok suggest that RTL may form a unified basis for a theory of decomposition.

  RTL's event occurrence function allows for a rich expression of periodic and non-periodic real-time properties. However, unstricted RTL is undecidable. It does not treat data structures or infinite state systems. RTL formulas impose a partial order on computational actions which is useful for representing high level timing requirements.

  RTL has been used with some success in industrial applications and it is also being used in a major IBM project called "ORE" which is integrating RTL with a real-time programming language. There is a feeling of confidence with RTL due to its pragmatic nature.

- **TPCTL** [27]. Timed Probabilistic Computation Tree Logic (TPCTL deals with real-time constraints and reliability. Formulas of TPCTL are interpreted over a discrete time extension of Milner's Calculus of Communication Systems called TPCCS. Probabilities are introduced by allowing two types of transitions, one labeled with actions and the other labeled with probabilities.

  The semantics of TPCTL is defined over the reactive transitions of TPCCS processes. TPCTL is a logic essentially extending the branching time modalities of CTL [17] with time and probabilities. Since formulas are interpreted over TPCCS processes, which are observed through actions that label transitions, the semantics of TPCTL is defined in terms of transitions rather than states.

  TPCTL is one of the few logics that can express both hard and soft real-time deadlines, and it is possible to represent levels of criticality in TPCTL.

Because of the action based nature of TPCTL, it is difficult to specify state-based properties such as "henceforth, if the train is at the crossing then the gate must be down". Propositions such as "the gate is down" must be encoded indirectly through actions that change the state of the model, in which case the specification becomes unnecessarily complicated.

TPCTL has no special development method. However, no practice has been carried out that using TPCTL as a independent tool to specify real-time systems.

Sample Description ITL is used as a sample here.

- **Syntax and Semantics**. The syntax of ITL is defined as following, where i is a constant, a is a static variable (does not change within an interval), A is a state variable (can change within an interval), v a static or state variable, g is a function symbol, p is a predicate symbol.

  Expressions:

  $$exp ::= i \mid a \mid A \mid g(exp_1, ..., exp_n) \mid \imath a : f$$

  Formulae:

  $$f ::= p(exp_1, ..., exp_n) \mid exp_1 = exp_2 \mid exp_1 < exp_2 \mid \neg f \mid f_1 \wedge f_2 \mid \forall\, v \bullet f \mid skip \mid f_1 \,;f_2 \mid f^*$$

  The informal semantics of the most interesting constructs are as following:

  - $\imath a : f$: the value of a such that f holds.
  - $\forall\, v \bullet f$: for all v such that $f$holds.
  - skip: unit interval(length 1).
  - $f_1 \,; f_2$: holds if the interval can be decomposed("chopped") into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix.
  - f^*: holds if the interval is decomposable into a finite number of intervals such that for each of them f holds.

  The formal semantics is as followings: Let X be a choice function which maps any nonempty set to some element in the set. We write $\sigma \sim_v \sigma'$ if the intervals σ and σ' are identical with the possible exception of their mapping for the variable v.

  - $\mathcal{M}_\sigma[\![v]\!] = \sigma_0(v)$.
  - $\mathcal{M}_\sigma[\![g(exp_1, ...exp_n)]\!] = \hat{g}\ (\mathcal{M}_\sigma[\![exp_1]\!], ..., \mathcal{M}_\sigma[\![exp_n]\!])$.
  - $\mathcal{M}_\sigma[\![\imath a : f]\!] = \begin{cases} \mathcal{X}(u) \ \ if \ \ u \neq \{\} \\ \mathcal{X}(a) \ \ otherwise \end{cases}$
    where $u = \sigma'(a) \mid \sigma \sim_a \sigma' \wedge \mathcal{M}_\sigma[\![f]\!] = tt$
  - $\mathcal{M}_\sigma[\![p(exp_1, ..., exp_n)]\!] = tt$ iff $\hat{p}\ (\mathcal{M}_\sigma[\![exp1]\!])$.
  - $\mathcal{M}_\sigma[\![\neg f]\!] = tt$ iff $\mathcal{M}_\sigma[\![f]\!] = ff$.
  - $\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!] = tt$ iff $\mathcal{M}_\sigma[\![f_1]\!] = tt \ and \ \mathcal{M}_\sigma[\![f_2]\!] = tt$.
  - $\mathcal{M}_\sigma[\![\forall\, v \bullet f]\!] = tt$ iff for all $\sigma' s.t. \sigma \sim_v \sigma', \mathcal{M}'_\sigma[\![f]\!] = tt$.
  - $\mathcal{M}_\sigma[\![skip]\!] = tt$ iff $\mid \sigma \mid = 1$.

– $\mathcal{M}_\sigma[\![f_1 \,;f_2]\!] = tt$ iff
  (exists a k,$s.t.\mathcal{M}_{\sigma_0 \ldots \sigma_k}[\![f_1]\!] = tt$ and
    $((\sigma\ is\ infinite\ and\ \mathcal{M}_{\sigma_k \ldots}[\![f_2]\!] = tt)$ or
    $(\sigma\ is\ finite\ and\ k \leq \mid \sigma \mid\ \ and\ \mathcal{M}_{\sigma_k \ldots \sigma_{|\sigma|}}[\![f_2]\!] = tt)))$
  or $(\sigma\ is\ infinite\ and\ \mathcal{M}_\sigma[\![f_1]\!])$.

– $\mathcal{M}_\sigma[\![f^*]\!] = tt$ iff
  if σ is infinite then
    (exist $l_0, \ldots, l_n, s.t. l_0 = 0$ and
      $\mathcal{M}_{\sigma_{l_n} \ldots}[\![f]\!] = tt$ and
      for all $0 \leq i < n, l_i <= l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, \ldots, \sigma_{l_{i+1}}}[\![f]\!] = tt.$)
    or
    (exists an infinite number of $l_i s.t. l_0$ and
      for all $0 \leq i, l_i <= l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, \ldots, \sigma_{l_{i+1}}}[\![f]\!] = tt.$)
  else
    (exist $l_0, \ldots, l_n s.t. l_0 = 0$ and $l_n = \mid \sigma \mid$ and
      for all $0 \leq i, l_i <= l_{i+1}$ and $\mathcal{M}_{\sigma_{l_i}, \ldots, \sigma_{l_{i+1}}}[\![f]\!] = tt.$)

- **Assessment**. ITL was first proposed by Moszkowski [47]. ITL avoids the proliferation of time variables in specifications, as do all temporal logics. ITL is sufficiently general to express any discrete computation. An executable subset of ITL, called Tempura [46], is well developed. Zedan and Cau proposed a set of new refinement techniques of ITL [10], which gives ITL a strong ability to describe all the popular possible features of real-time systems. Since ITL has an executable subset Tempura, its verification and simulation can be largely facilitated. The development method of ITL fits popular re-engineering methodologies well. Generally speaking, ITL is a formal logic with enough expressibility of real-time systems and suitable for re-engineering methodologies.

2.3 Algebraic Approach

General In this approach, an explicit definition of operations is given by relating the behaviour of different operations without defining states. Similar to the model-vased approach, no explicit representation of concurrency.

Examples

- **OBJ**[25]. OBJ is a wide spectrum first-order functional language that is rigorously based on equational logic. This semantics basis supports a declarative, specificational style, facilitates program verification, and allows OBJ to be used as a theorem prover.

- **LARCH** [26]. The Larch family of algebraic specification languages was developed at MIT and Xerox PARC to support the productive use of formal specifications in programming. One of its goals is to support a variety of different programming, including imperative languages, while at the same time localising programming language dependencies as much as possible. Each Larch language is composed of two components: the *interface* language which is specific to the particular programming language under consideration and the *shared language* which is common to all programming languages. The interface language is used to specify program modules using predicate logic with equality and constructs to deal with side effects, exception handling and other aspects of the given programming language. The shared language includes specification-building operations inspired by those in CLEAR, although these are viewed as purely syntactic operations on lists of axioms rather than as semantically non-trivial as in CLEAR.

Sample Description OBJ is used as a sample here.

OBJ [25] is a broad spectrum algebraic specification/programming language based on order sorted equational logic. It is a specification language in which an algebra is defined using *objects*. Objects are carrier sets along with operations, and equational theories which are treated by the OBJ interpreter as re-write axioms. Each object is built from primitive *sorts* and enrichments of existing objects.

Proofs of equivalence are achieved automatically in OBJ by rewriting processes into their normal forms and testing for syntactic equivalence.

There are now a number of enhanced OBJ interpreters, including OBJ1, OBJ2 and OBJ3. Here we prefer using the most up-to-date one: OBJ3.

OBJ3 is based on *order sorted equational logic*, which provides a notion of *subsort* that rigorously supports multiple inheritance, exception handling and overloading. OBJ3 also provideds *parameterised programming*, which gives powerful support for design, verification, reuse, and maintenance. This approach uses two kinds of module: *objects* to encapsulate executable code, and in particular to define abstract data types by initial algebra semantics; and *theories* to specify both syntactic structure and semantic properties for modules and module interfaces. Each kind of module can be parameterised, where actual parameters are modules. For parameter instantiation, a *view* binds the formal entities in an interface theory to actual entities in a module, and also asserts that the target module satisfies the semantic requirements of the interface theory.

2.4 Process Algebra Approach

General In this approach, explicit representation of concurrent processes is allowed. System behaviour is represented by constraints on all allowable observable communication between processes.

Examples

- **CSP** [29] [28]. The Communicating Sequential Processes (CSP) formal specification notation for concurrent systems was first introduced in [29]. Since this original proposal did not include a proof method, a complete version of CSP was proposed in [28].

- **CCS** [45]. Calculus of Communicating Systems (CCS) was proposed by Milner in 1989. It is a formalism similar to CSP. CCS is also suitable for distributed and concurrent systems. At present, several variations of CCS has been developed, which forms a CCS family. CCS family includes CCS, CCS+, CCS*, SCCS, TCCS and TPCCS [22].

  Two underlying concepts of CCS are *agents* and *actions*. A CCS model consists of a set of communicating processes (agents in CCS terminology). CCS adopts operational semantics.

  CCS is a successful formalism to build system models with respect to concurrency and distribution. Compared with CSP, the emphasis of CCS is on defining a series of equivalences (bisimulations), each equivalence defining a different model of concurrency. Thus certain processes that might be considered identical in CSP, would be different in CCS. CCS has a form of modal logic to specify the observable behaviours of processes. CSP has a richer set of laws than CCS allowing for optimising design and implementations. CCS concentrates on a minimal set of operators needed for the full expression of non-deterministic concurrency and its resulting equivalences.

CCS is not a real-time formalism either. Some extensions of CCS with real-time feature have been developed, such TCCS, SCCS, and TPCCS.

- **ACP** [6] [3]. Algebra of Communicating Processes (ACP) was proposed by J.A. Bergstra in 1984. Until now, a rather large variety of ACP has been proposed, such as Real Time ACP(ACP_ρ), Discrete Time ACP. ACP is also an action-based process algebra, which may be viewed as a modification of CCS. However, ACP is an executable formalism. ACP is equipped with a process graph semantics, and adopts bisimulation proof system. ACP allows a variety of communication paradigms, including ternary communication, through the choice of the communication function.

- **LOTOS** [32] [39]. LOTOS (Language Of Temporal Ordering Specification) was developed to define implementation-independent formal standards of OSI services and protocols. LOTOS has two very clearly separated parts. The first part provides a behavioural model derived from process algebra, principally from CCS but also from CSP. The second part of LOTOS allows specifiers to describe abstract data types and values, and is based on the abstract data type language ACT ONE.

  By combining the two formalism, CCS/CSP and ACT ONE integrately, LOTOS has a strong ability to describe both the "data" and "control" of the systems, i.e., ACT ONE for the data part and CCS/CSP-based language for the control part. LOTOS is able to capture a relatively complex temporal pattern of events, involving non=determinism, concurrency and synchronisation, by means of small algebraic expression built by using few conceptually simple operators [61].

  LOTOS has formally defined syntax, static semantics and dynamic semantics. The static semantics are defined by an attributed grammar [32] and the dynamic semantics are described operationally in terms of inference rules.

  Since LOTOS has an operational semantics, it is possible to implement these semantics in an interpreter. LOTOS has "a number of" various support tools, which are although not mature or narrow-aspected, do have some successful points [20].

  LOTOS does not support real-time specifications. Although a Timed LOTOS has been proposed, it is not proven a suitable formalism for real-time systems.

  LOTOS has problems in specifying distributed systems - it does not support dynamic reconfiguration which is an important and interesting characteristics in those systems. Its model of concurrency is based in the known "interleaved semantics" in which an observer can see one event a time and concurrency is represented sequentially. Many models based on "true concurrency semantics" have been proposed although it seems that none of them will be present in the next version of LOTOS. This is a weak point in represent distributed processing where in many situations things happen simultaneously and no ordering between events can be established. Also, LOTOS has weak data specification mechanisms and cannot express time explicitly.

- **TCSP** [54]. Timed CSP [54] is an extension of Hoare's CSP, with a dense temporal model providing a global clock. A delay operator is included along with some extended parallel operators. There is an assumption of a minimum delay between any two dependent action occurrences, but no minimum delay on any two independent actions. The semantics of

TCSP is given by timed traces, and a specification relation <u>*sat*</u> is provided for verifying predicates over traces.

Processes in Timed CSP are built from sequences of communication actions. The semantic model of TCSP is based on observation and refusal *timed traces*.

It is important in specification language for real-time systems that the temporal relationships between actions is maintained through the manipulation of the specification. In a non-real-time process algebra, the concurrency operators are usually *conservative*, i.e., they degenerate into non-deterministic interleaves of the constituent processes' actions. In TCSP, the concurrency operators are non-conservative, they do not degenerate. Instead, the algebraic rules for concurrent processes define the effect of composition on the temporal domain. For example, in the process where there is concurrent composition of two *WAIT* processes, the result is a process that waits for the maximum of two delays.

There exist no tools for the manipulation of specifications written in TCSP.

- **TPCCS** [27]. Timed Probabilistic Calculus of Communicating Systems (TPCCS) [27] is essentially an extension of Milner's CCS with discrete time and probabilities. To increase the description ability, a logic named Timed Probabilistic Computation Tree Logic (TPCTL) is proposed to describe the logic of and between TPCCS processes. Therefore TPCCS, together with TPCTL, forms a framework for specification and verification of real-time and reliability in distributed systems. TPCCS, as a process algebra, is used for modeling the operational behaviour of distributed real-time systems; and TPCTL, as a logic, is used for expressing properties of the systems. A verification method for automatically proving that a system described in TPCCS satisfies properties formulated in TPCTL, is also well defined [27].

  The main advantage of TPCCS is that it has a powerful description ability for real-time distributed systems. TPCCS can reason about both time and probabilities in distributed systems. In particular, TPCCS

  - extend CCS with probabilities by adding a probabilistic choice operator and by introducing a probabilistic transition relation,
  - add discrete time to the extended CCS where the timing model is based on a minimal delay assumption, i.e., communications must occur as soon as possible,
  - define a strong bisimulation equivalence for which a sound and complete axiomatisation is given.

  TPCCS has a very formally defined syntax and semantics, which bring lots of convenience in the automation of specification and verification. However, the calculation of probabilities is not mentioned in TPCCS and TPCTL. A tool named Timing and Probability Workbench (TPWB) has been developed. TPWB Partially supports automatic verification of TPCCS.

Sample Description CSP is used as a sample here.

- **Syntax and Semantics of CSP** A CSP specification is a hierarchy of processes. A complete specification can be viewed as a single process which is composed of sub-processes, each of which is decomposed into component processes.

The CSP notation has three primitive processes for input, output and assignment:

$A!e$	Output the value e to A;
$B?x$	From channel B input to x;
$x ::= e$	Assign x the value e.

A number of operators exist for combining processes, for example:

$P \| Q$	Processes P and Q operate in parallel;
$P \sqcap Q$	Either P or process Q operates. The choice is non-deterministic;
$P\ ;\ Q$	Process P operates followed by Q.

The basic concept in CSP considers a process as a mathematical abstraction of interactions between the system and its environment. Recursion is used to describe long lasting processes. A second feature is the use traces to record the sequence of actions a process has carried out. The abstract description is then given a more concrete explanation using algebraic law, and the last step is the implementation.

The notation for CSP uses *first order logic* symbols plus some additional symbols for *traces, functions*, etc.

There is a family of increasingly sophisticated models for providing CSP specification semantics. These computational models include the *counter model*, the *trace model*, the *divergences model*, the *readiness model* and the *failure model.*

- **Assessment of CSP**

The main contribution of CSP is as a programming language for parallel processing, principally in the area of synchronising communications.

CSP supports an event model that enables the description of entities that have properties and relationships that vary over time. It allows us to model a dynamic reality, to specify systems that perform various actions in particular orderings, and to express timing constraints between these actions and on the synchronisation of various system components.

CSP specifications may be couched at any level of abstraction, viewed quite simply as a system of processes executing independently, communicating over unbuffered unidirectional channels, and synchronising on particular events.

Specifications may be manipulated through the application of a number of algebraic laws, and combined by means of a small number of operators which are known to be sound. Various semantic models allow proposed properties to be proven, and to demonstrate that particular requirements have been satisfied.

These features make CSP a suitable formalism in the area of concurrency. However, like many other methods/languages, timing constraints associated with *real-time* operations cannot be handled, or have to be in a clumsy and inefficient way. CSP is not good at handling asynchronous events, such as interrupts.

There is no tools for CSP in strict sense. However, the Occam Transformation System developed by Oxford University's Programming Research Group is an automated tool to assist in carrying out algebraic transformation. Since Occam follows the main principles of Hoare's CSP, this tool may bring some convenience to CSP, too.

2.5 Net-Based Approach

General Graphical notations are popular notations for specifying systems as they are easier to comprehend and, hence, more accessible to non-specialists. In this approach, graphical languages with a formal semantics are used, which bring special advantages in system development and re-engineering.

Examples

- **Petri Net** [55] [52]. Petri Net theory is one of the first formalisms to deal with concurrency, nondeterminism and causal connections between events. According to [44] it was the first unified theory, with levels of abstraction, in which to describe and analyse all aspects of computer in the context of its environment.

  Petri nets provide a graphic representation with formal semantics of system behaviour. Until now, a large amount of vanities of Petri Net Theory has been proposed. Generally, petri nets can be classified into ordinary (classic) petri nets and timed petri nets.

- **Timed Petri Net** [43] [21] [7] [8] [38] [53]. Petri Net theory was the first concurrent formalisms to deal with real-time. Two basic timed versions of Petri nets have been introduced: Timed petri Nets [124] and Time Petri Nets [43]. Both have been used in recent work [21] [7] [8] [38] [53]. There are two questions that arise when time is introduced to net theory: (i) the location of the time delays(at places or transitions) and (ii) the type of delay (fixed delays, intervals or stochastic delays).

  Timed Petri Nets are derived from classical Petri nets by associating a firing finite duration (a delay) with each transition of the net. The transition is disabled from occurrence for the delay period, but is fired immediately after becoming enabled. These nets are used mainly in performance evaluation.

  Time Petri Nets (TPNs) are more general than Timed Petri Nets. A Timed Petri Net can be simulated by a TPN, but not vice versa. Both a lower and an upper bound are associated with each transition in a TPN. A state in the reachability graph is a tuple consisting of a marking, and a vector of possible firing intervals of enabled transitions in that marking.

- **Statecharts** [31]. Statecharts [31] provides an abstraction mechanism based on finite state machine. It represents an improved version of the structured methods. A graphic tool called "Statemate" [2] exists to implement the formalism. Methods similar to that of Statecharts may be found in [23].

  Statecharts have been proved to be at least as expressive as state machines, and the succinct justification for them is provided by the following "equation":

  Statecharts = *state-transitions* + *depth* + *orthogonality* + *broadcast communication*.

  Statecharts provides an abstraction mechanism based on finite state machine. It represents an improved version of the structured methods. Charts denote composition of state machine into super-machines which may execute concurrently. The state machines contain transitions which are marked by enabling and output events. It is assumed that events are instantaneous, and a global discrete clock is used to trigger sets of concurrent events. Statecharts are hierarchical, and may be composed into complex charts. The semantics of Statecharts is given by maximal computation histories. An axiomatic system is presented.

  Statecharts support typical structural top-down system development method. It does not fit the procedures of reverse engineering, which abstracts specifications from source code. Statecharts have no real-time features, although certain efforts are undergoing.

Sample Description Petri Net is use as a sample here.

- **Syntax and Semantics of Petri Nets**

  The classic Petri Nets model is a 5-tuple (P, T, I, O, M). P is a finite set of places (often drawn as circles), representing conditions. T is a finite set of transitions (often drawn as bars), representing events. I and O are sets of input and output functions mapping transitions to bags of places(the incidence functions). M is the set of initial markings.

  Places may contain zero or more tokens (often drawn as black circles). A marking (or state) of the Petri nets is the distribution of tokens at a moment in time, i.e. $M : P \to N$ where N is the non-negative integers. Tokens in Petri nets model dynamic behaviour of systems. Markings change during execution of the Petri nets as the tokens "travel" through the net.

  The execution of the Petri nets is controlled by the number and distribution of the tokens (the state). A transition is enabled if each of its input places contains at least as many tokens as there exists arcs from that place to the transition. When a transition is enabled it may fire. When a transition fires, all enabling tokens are removed from its input places, and a token is deposited in each of its output places.

  Given an initial state (distribution of tokens), the reachability set is the set of all states that result from executing the Petri net. Properties such as boundness, liveness, safety and freedom from deadlock can be checked by analysing the reachability graph. The reachability graph is usually constructed using an interleaving operational semantics.

  In Petri nets causal dependencies and independencies in some set of events are explicitly represented. It is therefore easy to provide a non-restrictive partial order semantics. Events which are independent of each other are not projected onto a linear timescale. Instead a non-interleaving partial order relation of concurrency is introduced.

- **Assessment of Petri Nets**

  The advantages of using Petri nets are numerous. They are easy to comprehend due to their graphical form, they can be used to model hardware, software and human behaviour, and they allow formal reasoning of system behaviour. Some experts suggest using both Petri nets and formal logic for developing systems and the former to model the system, the latter to verify it.

  Ordinary Petri nets have been criticised for not being able to deal with fairness and data structures, e.g. the data in a measure header, although the number of tokens at a particular place in the net can simulate a local program variable. Structuring mechanisms such as composition operators are not inherently part of the theory, and there is no calculus to transform a net into a real-time programming language. Unlike state machine, a "place" in a Petri net cannot easily be identified with a place in the corresponding program code. A further problem is that the reachability graph suffers from state explosion as Petri nets become larger, thus impacting on the ability to scale up analysis to larger systems. Ordinary Petri nets are still an object of intense research aimed at putting Petri nets theory on firm mathematical ground. However, practically speaking, such standard nets are not up to the task of modeling complex systems. For this reason, higher level nets (coloured nets) and stochastic nets have been introduced to extend the modeling power of Petri nets.

3 CRITERIA AND RESULTS

In this section, we summarise a wide spectrum of existing formal methods from the point of view of software re-engineering. Generally speaking, some of them already have a rather good advantage in certain aspects, such as ITL for real-time systems, and TPCCS & TPCTL for systems with reliability and probabilities. However, all of them have certain flaws or weakness in some aspects as described in section 2.

We list our findings through the review in forms of tables according to the following criteria:

- Temporal Model – Temporal model is the model of time used by the formal methods. A sparse model has discrete instances of time and there is a minimum granularity. A dense model is not discrete, between any two instances in time there is an infinite number of other instances.
- Automated Tools – This criterion refers to whether the formal method has relevant automated tools to support its development, such as checking syntax, verifying semantics and auto-execution.
- Reliability – This criterion refers to the reliability of the formalism.
- Proof System – This refers to whether there is any proof system and what the type of the system is (when there is one).
- Industrial Strength – This criterion refers to the potential of the formal method for large-scale/industrial applications.
- Methods of Verification – This criterion refers to the existing methods of verification of the formal method. Normally, there are two types of the methods of verification: model checking and theorem proving.
- Concurrency – This criterion refers to the explicit representation and reasoning of *concurrency.*
- Communication – This criterion refers to the explicit representation and reasoning of *communication.*
- Reverse Engineering – This criterion refers to whether the formal method has been applied in any reverse engineering domain.

Criteria	Z	VDM	B
Temporal Model	none	none	none
Automated Tools	a few	none	good
Reliability	good	good	good
Proof System	semi-axiomatic	semi-axiomatic	axiomatic
Industrial Strength	great	some	great
Methods of Veri.	model-checking	model-checking	both
Concurrency	none	none	none
Communication	none	none	none
Reverse Eng.	yes	no	no

Table 1: Model/State-Based Formalisms

Criteria	HOL	WP-Calc.	TL	ML
Temporal Model	none	none	dense/sparse	none
Automated Tools	some	some	some or few	few
Reliability	good	good	good	good
Proof System	axiomatic	axiomatic	axiomatic	axiomatic
Industrial Strength	some	some	great	great
Methods of Veri.	theorem proving	theorem proving	both	both
Concurrency	none	none	norm exist	none
Communication	none	none	norm exist	none
Reverse Eng.	yes	yes	no	no

Table 2-1: Logic-Based Formalisms

Criteria	ITL	DC	TAM	RTTL	RTL
Temporal Model	sparse	dense	sparse	sparse	sparse
Automated Tools	few	none	none	few	none
Reliability	good	good	good	good	good
Proof System	axiomatic	axiomatic	axiomatic	axiomatic	axiomatic
Industrial Strength	great	some	great	some	some
Methods of Veri.	theorem prov.	theorem prov.	theorem prov.	theorem prov.	theorem prov.
Concurrency	par. comp.	none	exist	interleaved	interleaved
Communication	sync./async.	none	exist	sync.	none
Reverse Eng.	no	no	no	no	no

Table 2-2: Logic-Based Formalisms

Criteria	OBJ	Larch
Temporal Model	none	none
Automated Tools	few	some
Reliability	good	good
Proof System	axiomatic	axiomatic
Industrial Strength	some	great
Methods of Veri.	theorem prov.	theorem prov.
Concurrency	interleaved	interleaved
Communication	sync.	sync
Reverse Eng.	no	no

Table 3: Algebraic Formalisms

Criteria	CSP	CCS	ACP	LOTOS	TCSP
Temporal Model	none	none	none	none	dense
Automated Tools	none	none	good	some	none
Reliability	good	good	good	good	good
Proof System	axiomatic	bisimulation	bisimulation	bisimulation	axiomatic
Industrial Strength	some	some	some	great	some
Methods of Veri.	both	both	both	model-checking	both
Concurrency	interleaved	interleaved	interleaved	interleaved	both
Communication	sync/async.	sync.	sync.	sync.	sync.
Reverse Eng.	no	no	no	no	no

Table 4: Process Algebra Formalisms

Criteria	Petri Nets	Timed Petri Nets	Statecharts
Temporal Model	none	dense/sparse	sparse
Automated Tools	some	none	none
Reliability	good	good	good
Proof System	reachability	reachability	axiomatic
Industrial Strength	some	some	some
Methods of Veri.	model-checking	model-checking	model-checking
Concurrency	interleaved	interleaved	exist
Communication	sync.	sync.	sync.
Reverse Eng.	yes	no	no

Table 5: Graphic-Based Formalisms

The above five categories are corresponding to subsections of Section 2. We believe we should use the sixth category in order to better summarise those so-called "combined" approaches.

Criteria	TPCCS + TPCTL	Petri Nets + Predicate
Temporal Model	sparse	sparse/dense
Automated Tools	none	none
Reliability	good	good
Proof System	axiomatic	reach. ps axiom.
Industrial Strength	some	unknown
Methods of Veri.	theorem proving	model-checking
Concurrency	interleaved	interleaved
Communication	sync.	sync.
Reverse Eng.	no	no

Table 6: Combined Formalisms

Through reading these tables, we can draw the following conclusions of the current situation of formal methods for re-engineering:

- Some formalisms are rather good in certain aspects of software development while others are good in other aspects. For example, ITL has a strong ability for representing and reasoning of most features of real-time systems. TPCCS & TPCTL is good at dealing with systems with reliability and probability features. Z is capable for large scale industrial applications. B has a comprehensive automated toolkit. DC has advantages for its ability of dealing with dense temporal models. Various process algebras are excellent for their abilities of representing and reasoning of concurrency and communication. Finally, the most important features of net-based formalisms are their graphical representations: concise, easy to understand, and very clear.

- Only a very few formalisms have been applied as the theoretical foundation of reverse engineering;

- Although some formalisms are suitable for certain stages of reverse engineering, there is not any formalism covering all reverse engineering stages. For example, Hoare Logic can cope with the low level abstraction of program source code, but not high level abstraction. This also happens to the formalisms such as Wp-Calculus and predicate logic. Therefore, a new wide spectrum formalism is needed for the re-engineering process, e.g. integrating Wp-Calculus and ITL could be a solution.

It is not hard to see that most existing formal methods were not designed for reverse engineering as well as re-engineering. This urges that research into suitable formal methods for re-engineering should be established.

4 DISCUSSION AND ANALYSIS

This review is conducted in the view of developing a practical approach for the re-engineering of existing system including real-time critical application. Re-engineering generally consists of three stages, i.e., restructuring, reverse engineering and forward engineering. Because most existing formal approaches were developed for forward engineering, whether a formal approach has been used for reverse engineering is specially used as a criterion.

Through the review, we found that using formal methods in reverse engineering existing systems (real-time systems, in particular) is still a research area that has not been addressed properly, because (1) there are formal methods for reverse engineering and (2) even if a formal method can cope with reverse engineering well, it is still a problem whether this formal method can be integrated with an existing matured forward engineering formal method.

Graphical notations are also popular notations for reverse engineering (understanding) existing systems. Petri Net is a useful for building a graphical model for re-engineering.

Another factor that should be taken into consideration when re-engineering computing systems is recent rapid development of object-oriented technology. We believe that an approach that integrates formal methods, graphic techniques and object-oriented techniques can contribute to solve the problem of re-engineering:

- existing software can be easily understood and reverse engineered through using a graphic tool;

- object-oriented techniques, which have been recognised as the best way currently available for structuring software systems, can help maintenance in grouping together data and operations performed on them, thereby encapsulating the whole system behind a clean interface, and organising the resulting entities in a hierarchy based on specialisation in functionalities;

- formal methods can provide a solid theoretical foundation for integrating graphic and object-oriented techniques in building a practical software maintenance and re-engineering tool.

Therefor our goal is to devise a uniform coherent semantic theory that enables a comprehensive formal understanding of re-engineering model when applied to the analysis and the development of complex computing systems in real applications. This should allow diverse kinds of formalisms to be developed and integrated.

The unified theory provides a formal basis within which an object-oriented formal notation will be developed that unifies existing widely-used formalisms. In addition, sound transformational calculi together with verification and validation techniques will be developed. Our approach to this is to build a wide-spectrum language in which concrete and abstract (e.g. specification statement) graphical notations could be easily intermixed. The developed calculi will then allow us to transform from one form of specification/program (represented graphically) to another.

The novel aspects of this proposal are in the incorporation of the outcome of extensive research in a number of key areas of software maintenance into a formally unified semantic model.

We intend to use a wide-spectrum language approach to the proposed formal re-engineering of existing computing systems, particularly real-time systems. Our extensive experience with the design and use of the Wide Spectrum Language (WSL) [4,16,63] and TAM [10,59,64] have illustrated the practical use of such an approach. In our next research stage, we therefore aim to:

1. develop a single "wide-spectrum" language in which both abstract specifications written in our extended logic and executable code may be described in graphical notations,
2. define a refinement relation on specifications and programs described in the language, and
3. develop a family of sound refinement calculi to serve for both forward and backward refinement.

References

[1] Abrial, J. R., Schuman, S. A. and Meyer, B., *Specification Language Z*, Massachusetts Computer Associates Inc., Boston, 1979.

[2] Alur, R. and Dill, D. L., "Automata for Modeling Real-Time Systems", In M.S. Paterson editor, ICALP 90: Automata, Languages and Programming, Lecture Notes in Computer Science, 1990.

[3] Baeten, J. C. M. and Bergstra, J. A., "Real Time Process Algebra", *Formal Aspects of Computing*, Vol. 3, pp. 142–188 (Feb 1991).

[4] Bennett, K. H., Bull, T. and Yang, H., "A Transformation System for Maintenance — Turning Theory into Practice", IEEE Conference on Software Maintenance-1992, Orlando, Florida, November, 1992.

[5] Benveniste, A. and Harter, P. K., "Proving Real-Time Properties of Programs with Temporal Logics", Proceedings of ACM SIGOPS 8th annual ACM symposium on Operating systems Principles, December 1981.

[6] Bergstra, J. A. and Klop, J. W., "Process Algebra for Synchronous Communication", *Information and Control*, Vol. 60, pp. 109–137 (Jan/Feb/Mar 1984).

[7] Berthomieu, B. and Diaz, M., "Modeling and Verification of Time Dependent Systems Using Timed Petri Nets", *IEEE Transactions on Software Engineering*, Vol. 17, pp. 259–273 (March 1991).

[8] Billington, J., Wheeler, G. R. and Wilbur-Ham, M. C., "PROTEAN: a High-level Petri Net Tool for the Specification and Verification of Communication Protocol", *IEEE Transactions on Software Engineering*, Vol. 14, pp. 301–316 (March 1988).

[9] Bjϕrner, D. and Jones, C. B., *Formal Specification and Software Development*, Prentice-Hall International, 1982.

[10] Cau, A. and Zedan, H., "Refining Interval Temporal Logic specifications", Draft Paper, November 1996.

[11] Cau, A., Zedan, H., Coleman, N. and Moszkowski, B., "Using ITL and Tempura for Large Scale Specification and Simulation", Proceedings of 4th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE, Braga, Portugal, 1996.

[12] Chaochen, Z., Hoare, C. A. R. and Ravn, A. P., "A Calculua of Durations", *Information Processing Letters*, Vol. 40, pp. 269–276 (05 1991).

[13] Chaochen, Z., Ravn, A. P. and Hansen, M. R., "An Extended Duration Calculus for Hybris Systems", Hybrid Systems, R.L. Grossman, A. Nerode, A. P. Ravn. H. Rischel(Eds.), 1993.

[14] Cheng, B. H. C. and Gannod, G. C., "Abstraction of Formal Specifications from Program Code", Proceedings for the 3rd International Conference on Tools for Artificial Intelligence, 1991.

[15] Chllas, B. F., *Modal Logic: An Introduction*, Cambridge University Press, 1980.

[16] Chu, W. C. and Yang, H., "A Formal Method for Software Maintenance", IEEE International Conference on Software Maintenance (ICSM'96), Monterey, CA , November 1996.

[17] Clarke, E., Emerson, E. A. and Sistla, A. P., "Automatic Verification of Finite Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", Proceedings of the 10th ACM Symposium on Principles of Programming Languages, 1983.

[18] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976, ISBN 0-13-215871-X.

[19] Dijkstra, E. W. and Scholten, C. S., *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

[20] Eijk, P. H. J. van, Vissers, C. A. and Diaz, M., *The Formal Description Technique LOTOS*, Elsevier Science Publishers, 1989.

[21] Etessami, F. S. and Hura, G. S., "Rule-based Design Methodology for Solving Control Problems", *IEEE Transactions on Software Engineering*, Vol. 17, pp. 274–282 (March 1991).

[22] Fencott, C., *Formal Methods for Concurrency*, International Thomson Publishing Company, ISBN 1-85032-173-6, 1996.

[23] Gabrielian, A. and Franklin, M. K., "State-based Specification of Complex Real-time Systems", Proceedings of the 9th Real-Time Systems Symposium, December 1988.

[24] Gannod, C. and Cheng, B. H. C., "Strongest Postcondition Semantics as a Basis for Reverse Engineering", Proc. of IEEE Working Conference on Reverse Engineering, , Toronto, Ontario, July, 1995.

[25] Goguen, J. and Tardo, J., "An Introduction to OBJ: A Language for Writing and Testing Software Specifications,", In Marvin Zelkowitz editor, Specification of Reliable Software, 1979, Reprinted by Addison Wesley in 1985, 'Specification Techniques', p391-420.

[26] Guttag, J. and Horning, J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[27] Hansson, H. A., "Time and Probability in Formal Design of Distributed Systems", *Real-Time Safety Critical Systems Series*, Vol. 2 (1994).

[28] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, 1985.

[29] Hoare, C. A. R., "Communicating Sequential Processes", *Communication of ACM*, Vol. 21, pp. 666–677 (08 1978).

[30] Hooman, J., "Specification and Compositional Verification of Real-Time Systems", PhD Thesis, Eindhoven, the Netherlands, 1991.

[31] Hooman, J., Ramesh, S. and Roever, W. P. de, “A Compositional Semantics for Statecharts”, Technical Report, The Netherland, 1989.

[32] ISO, “Information Systems Processing—Open Systems Interconnection—LOTOS”, Technical Report, 1987.

[33] Jahanian, F. and Mok, A., “Safety Analysis of Timing Properties in Real-Time Systems”, *IEEE Transactions on Software Engineering*, Vol. 12 (09 1986).

[34] Jones, C. B., *Software Development: A Rigorous Approach*, Prentice-Hall International, 1980.

[35] Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall International, Inc., London, 1986.

[36] Lano, K. C. and Haughton, H. P., “Formal Development in B”, *Information and Software Technology*, Vol. 37, pp. 303–316 (June 1995).

[37] Lano, K., *The B Language and Method: A Guide to Practical Formal Development*, Springer-Verlag, ISBN 3-540-76033-4, 1996.

[38] Leveson, N. G. and stolzy, J. L., “Safety Analysis Using Petri Nets”, *IEEE Transactions on Software Engineering*, Vol. 13, pp. 386–397 (March 1987).

[39] Logrippo, L., Melanchuk, T. and Wors, R. J. D., “The Algebraic Specification Language LOTOS: an Industrial Experience”, *ACM SIGSOFT Software Engineering Notes*, Vol. 15, pp. 59–66 (04 1990).

[40] Mahoney, B. P. and Hayes, I. J., “A Case Study in Timed Refinement: A Mine Pump”, *IEEE Transactions on Software Engineering*, Vol. 18, pp. 817–825 (09 1992).

[41] Manna, Z. and Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, ISBN 0-387-97664-7, , 1996.

[42] Mehmet, A. and Wanli, M., “An Overview of Temporal Logic Programming”, *First International Conference, ICTL'94, Lecture Notes in AI*, Vol. 827, pp. 445–481 (1994).

[43] Merlin, P. M. and Segall, A., “Recoverability of Communication Protocols–Implications of a Theoretical Study”, IEEE Transactions on Communications, September 1976.

[44] Milner, R., “Some Directions in Concurrency Theory(panel statement)”, Proceedings of the international Conference on the fifth Generation Computer Systems, 1988.

[45] Milner, R., *Communication and Concurrency*, Prentice Hall, Hertfordshire, 1989.

[46] Moszkowski, B., *Executing Temporal Logic Programs*, Cambridge University Press, Cambridge UK, 1986.

[47] Moszkowski, B., *A Temporal Logic for Multilevel Reasoning about Hardware*, IEEE Computer Society, February 1985.

[48] Narayana, K. T. and Aaby, A. A., “Specification of Real-Time Systems in Real-Time Temporal Interval Logic”, Proceedings of Real-Time Systems Symposium, December 1988.

[49] Ostroff, J. S., “Temporal Logic for Real-Time Systems”, Advanced Software Development Series, England, 1989.

[50] Ostroff, J. S., “Deciding Properties of Timed Transition Models”, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, pp. 170–183 (April 1990).

[51] Ostroff, J. S. and Wonham, W. M., “A Temporal Logic Approach to Real-Time Control”, Proceedings of the 24th IEEE Conference on Decision and Control, Florida, December 1985.

[52] Peterson, J. L., *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[53] Razouk, R. R. and Phelps, C. V., “Performance Analysis of Timed Petri Nets”, Proceedings of 4th International Workshop on Protocol Verification and Testing, June 1984.

[54] Reed, G. M. and Roscoe, A. W., “Timed CSP: Theory and Practice”, REX Workshop—Real-Time : Theory and Practice, 1992.

[55] Reisig, W., *Petri Nets: an Introduction*, Springer-Verlag, Berlin, 1985.

[56] Rescher, N. and Urquhart, A., “Temporal Logic”, Library of Exact Philosophy, 1971.

[57] Scholefield, D., “A Refinement Calculus for Real-Time Systems”, PhD thesis, 1992.

[58] Scholefield, D. and Zedan, H., “TAM: A Formal Framework for the Development of Distributed Real-Time Systems”, Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Nijmegen, Netherland, January 1992.

[59] Scholefield, D., Zedan, H. and He, J., “A Specification-oriented Semantics for the Refinement of Real-Time Systems”, *Theoretical Computer Science*, Vol. 130 (August 1994).

[60] Spivey, J. M., *Understanding Z*, Cambridge University Press, 1988.

[61] Turner, K. J., “DILL-Digital Logic in LOTOS”, Formal Description Techniques, FORTE VII, North Holland, October 1993.

[62] Wordsworth, J., *Software Engineering with B*, Addison Wesley Longman, ISBN 0-201-40356-0., 1996.

[63] Yang, H. and Bennett, K. H., “Aquairing Entity-Relationship Attribute Diagrams from Code and Data through Program Transformation”, IEEE International Conference on Software Maintenance (ICSM ’95), Nice, France, October, 1995.

[64] Zedan, H. and Heping, H., “An Executable Specification Language for Fast Prototyping Parallel Responsive Systems”, *Computer Language*, Vol. 22, pp. 1–13 (01 1996).