Formal Specification: a Roadmap

Axel van Lamsweerde

Département d'Ingénierie Informatique
Université catholique de Louvain
B-1348 Louvain-la-Neuve (Belgium)
avl@info.ucl.ac.be

ABSTRACT

Formal specifications have been a focus of software engineering research for many years and have been applied in a wide variety of settings. Their industrial use is still limited but has been steadily growing. After recalling the essence, role, usage, and pitfalls of formal specification, the paper reviews the main specification paradigms to date and discuss their evaluation criteria. It then provides a brief assessment of the current strengths and weaknesses of today's formal specification technology. This provides a basis for formulating a number of requirements for formal specification to become a core software engineering activity in the future.

1. INTRODUCTION

Formal specifications have been considered since the good old days of Computing Science. In the late nineteen forties, Turing observed that reasoning about sequential programs was made simpler by annotating them with properties about program states at specific points [Ran73]. In the late sixties, Floyd, Hoare and Naur proposed axiomatic techniques for proving the consistency between sequential programs and such properties, called specifications [Flo67, Hoa69, Nau69]. Dijkstra showed how a formal calculus over such specifications could be used constructively to derive non-deterministic programs that meet them [Dij75]. Specific techniques were also proposed to formally express intended properties for special kinds of programs, notably, data-structured programs [Par72, Lis75] and concurrent programs [Pnu77]. This was the starting point for a whole new area of research aimed at specification-in-the-large [Par77, SRS79, Abr80, Hen80]. The interest in formal specifications and their multiple uses in software engineering has been growing continually since that point [Win90, Cra93, Hin95, Cla96, Win99, SCP2K].

What are formal specifications?

Formal specifications may refer to fairly different things in the software lifecycle; the wording is thus heavily overloaded. An additional source of confusion stems from the fact that a single word is used for a product and the corresponding process.

Generally speaking, a *formal specification* is the expression, in some formal language and at some level of abstraction, of a collection of properties some system should satisfy.

This purposely general definition covers different notions dependent on what the word "system" really covers, what kind of properties are of interest, what level of abstraction is considered, and what kind of formal language is used.

Complex software applications are built using a series of development steps: (a) high-level goals are identified and refined until a set of requirements on the software and assumptions on the environment can be made precise to satisfy such goals; (b) a software architecture, made of interconnected software components, is designed to satisfy such requirements; and (c) the various components are implemented and integrated so as to satisfy the architectural descriptions. All along this development/satisfaction chain, knowledge about the application domain is often used to guide the elaboration and to support the validation with respect to upstream prescriptions.

The "system" being specified may be a descriptive model of the domain of interest; a prescriptive model of the software and its environment; a prescriptive model of the software alone; a model for the user interface; the software architecture; a model of some process to be followed; and so on. The "properties" under consideration may refer to high-level goals; functional requirements; non-functional requirements about timing, performance, accuracy, security, etc.; environmental assumptions; services provided by architectural components; protocols of interaction among such components; and so on.

Beyond such different realizations of the general concept of specification, there is a common idea of specifications pertaining to the *problem domain* (as opposed to the solution domain). To make sure some solution solves a problem correctly, one must first state that problem correctly. This dichotomy is however simplistic; a solution to a problem may in general be given as a set of subproblems to be specified and solved in turn [Swa82]. A specification must thus in general satisfy some higher-level specification and be satisfied by some lower-level specifications.

"Formal" is often confused with "precise" (the former entails the latter but the reverse is of course not true). A specification is *formal* if it is expressed in a language made of three components: rules for determining the grammatical well-formedness of sentences (the syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring useful information from the specification (the proof theory).

The latter component provides the basis for automated analysis of the specification.

The collection of properties being specified is often fairly large; the language should thus allow the specification to be organized into *units* linked through *structuring relationships* - such as specialization, aggregation, instantiation, enrichment, use, etc. Each unit in general has a declaration part, where variables of interest are declared, and an assertion part, where the intended properties on the declared variables are formalized. Formal specification techniques essentially differ from semi-formal ones (such as dataflow diagrams, entity-relationship diagrams or state transition diagrams) in that the latter do not formalize the assertion part.

What are good specifications?

Writing a "correct" specification is very difficult - probably as difficult as writing a correct program. A specification must be *adequate*, that is, it must adequately state the problem at hand. It must be *internally consistent*, that is, it must have a meaningful semantic interpretation that makes true all specified properties taken together. It must be *unambiguous*, that is, it may not have multiple interpretations of interest making it true. It must be *complete* with respect to higher-level ones, that is, the collection of properties specified must be sufficient to establish the latter [Yue87]. It must be *satisfied* by lower-level ones. It should be *minimal*, that is, it should not state properties that are irrelevant to the problem or that are only relevant to a solution for that problem [Mey85].

Why specify formally?

Problem specifications are essential for designing, validating, documenting, communicating, reengineering, and reusing solutions. Formality helps in obtaining higher-quality specifications within such processes; it also provides the basis for their automated support.

The act of formalization in itself has been widely experienced to raise many questions and detect serious problems in original informal formulations. Besides, the semantics of the formalism being used provides precise rules of interpretation that allow many of the problems with natural language to be overcome. A language with rich structuring facilities may also produce better structured specifications.

As the major payoff, formal specifications may be manipulated by automated tools for a wide variety of purposes:

- to derive premises or logical consequences of the specification, for user confirmation, through deductive theorem proving techniques [Owr95, Man96];
- to confirm that an operational specification satisfies more abstract specifications, or to generate behavioral counterexamples if not, through algorithmic model checking techniques [Que82, Cla86, Hol91, Hol97, McM93, Atl93, Man96, Hei98a, Cla99];
- to generate counterexamples to claims about a declarative specification [Jac96];
- to generate concrete scenarios illustrating desired or undesired features about the specification [Fic92, Hal95, Hal98] or, conversely, to infer the specification inductively from such scenarios [Lam98c];
- to produce animations of the specification in order to check its adequacy [Hek88, Har90, Dub93, Doug94, Heit96, Tho99];
- to check specific forms of specification consistency/completeness efficiently [Heim96, Heit96];
- to generate high-level exceptions and conflict preconditions that may make the specification unsatisfiable [Lam98b, Lam2K];
- to generate higher-level specifications such as invariants or conditions for liveness [Lam79, Ben96, Par98, Jef98];
- to drive refinements of the specification and generate proof obligations [Car90, Abr96, Dar96];
- to generate test cases and oracles from the specification [Ber91, Ric92, Roo94, Wey94, Man95];
- to support formal reuse of components through specification matching [Kat87, Reu91, Mas97, Zar97].

Formal specifications can also be generated from program code as a basis for reverse engineering and software evolution [Gan96, Ern99].

Specify... for whom?

One of the problems with formal specifications is that they may concern different classes of consumers having fairly different background, abstractions and languages - clients, domain experts, users, architects, programmers, and tools. For example, the specification of a goal or requirement should be checked by clients for adequacy; a domain description should be produced or checked by domain experts; an architectural component specification should be seen in a detailed form by programmers assigned to that component and in a more abstract form by programmers assigned to other components using that component; a tool should see a specification in some efficiently processable form; and so on. One way to handle such clashes is to support multilingual specifications, at the price of raising consistency problems (see below).

It is now well-accepted that a programming language should be a language for the programmer, not for the machine. This principle is still not widely accepted for specification languages; many of them still seem to be designed for programmers or for tools rather than for specifiers.

Specify... when?

As seen before, there are multiple stages in the software lifecycle at which formal specifications may enter the picture, e.g., when modeling the domain; when elaborating the goals, requirements on the software, and assumptions about the environment; when designing a functional model for the software; when designing the software architecture; or when modifying or reengineering the software.

The main focus to date has been on formal specifications written during the *design* of a preliminary functional model for the software [Win90]. We will therefore focus the discussion of past achievements on this kind of specification mainly. We will also take the viewpoint of *specification building* since formal reasoning is covered in another chapter

of this volume.

2. FORMALIZATION: SCOPE AND PITFALLS

Although close to commonsense, there are a few important principles and facts that are often overlooked by champions of formalization.

- Specifications are never formal in the first place. To state properties precisely and formally, one must first figure out what these properties are. The latter must necessarily be formulated in a language all parties can speak and understand, that is, natural language.
- Formal specifications are meaningless without a precise, informal definition of how to interpret them in the domain considered. A formalization involves terms and predicates which may have many different meanings. The specification thus makes sense only if the meaning of each term/predicate is stated precisely, by mapping function/predicate names to functions/relations on domain objects. This mapping must be precise but necessarily informal (to avoid infinite regression). This fairly obvious principle is often neglected [Zav97].
- Formal specification is not a mere translation process from informal to formal. The specification of a large, complex system requires relevant objects and phenomena to be identified, interrelated, and characterized through properties of interest. Model construction and property description are thus tightly coupled components of any specification-in-the-large process.
- Formal specifications are hard to develop and assess.This stems from the diversity and subtlety of errors that can be made (see Section 1) and from the multiplicity of modeling choices that can be made. As a consequence, formal specifications are rarely correct in the first place. It has been frequently noted, however, that even wrong specifications may help finding out problems in original formulations.
- The rationale for specific modeling choices in a specification is important for explanation and evolution [Sou93]. Unfortunately, such rationale is rarely documented.
- The by-products of a formal specification process are often more important than the formal specification itself; they include a better informal specification, obtained by feedback from formal expression, structuring and analysis; and lower-level products that are more likely to satisfy them thanks to such formalization/analysis.
- To be useful, a formal system must have a limited domain of applicability. Specific types of systems require specific types of techniques for natural expression and efficient analysis. For example, the formal specification of a compiler must include a definition of the input grammar. A BNF-style specification would be most appropriate for this domain but clearly inappropriate for the domain of process-control systems. There is thus no point in looking for a universal specification technique.

3. SPECIFICATION PARADIGMS

Formal specification techniques differ mainly by the particular specification paradigm they rely on. In the sequel, we avoid the usual, somewhat confusing model-based vs. property-based dichotomy; the reason is that for large systems any property-based specification involves system modeling and any model-based specification involves system properties.

History-based specification

The principle here is to specify a system by characterizing its maximal set of admissible histories (or "behaviors") over time. The properties of interest are specified by temporal logic assertions about system objects; such assertions involve operators referring to past, current and future states. The assertions are interpreted over time structures. Time can be linear [Pnu77] or branching [Eme86]. Time structures can be discrete [Man92, Lamp94], dense [Gre86], or continuous [Han91]. The properties may refer to time points [Man92, Lam94], time intervals [Mos97], or both [Gre86, Jah86, All89, Ghe91]. Most often it is necessary to specify properties over time bounds; real-time temporal logics are therefore necessary [Koy92, Dub91, Mor92, Dar93, Mos97].

State-based specification

Instead of characterizing the admissible system histories, one may characterize the admissible system states at some arbitrary snapshot. The properties of interest are specified by (a) invariants constraining the system objects at any snapshot, and (b) pre- and post-assertions constraining the application of system operations at any snapshot. A pre-assertion captures a weakest necessary condition on input states for the operation to be applied; a post-assertion captures a strongest effect condition on output states if the operation is applied. The latter may be explicit or implicit dependent on whether or not the assertion contains equations defining the output constructively.

Languages such as Z [Abr80, Spi92, Pot96], VDM [Jon90] or B [Abr96] relyon this paradigm. Object-oriented variants have been proposed as well [Lan95].

Transition-based specification

Instead of characterizing admissible system histories or system states, one may characterize the required transitions from state to state. The properties of interest are specified by a set of transition functions in the state machine transition; the transition function for a system object gives, for each input state and triggering event, the corresponding output state. The occurrence of a triggering event is a sufficient condition for the corresponding transition to take place (unlike a precondition, it captures an obligation); necessary preconditions may also be specified to guard the transition.

Languages such as Statecharts [Har87], PROMELA [Hol91], STeP-SPL [Man92], RSML [Lev94] or SCR [Par95, Heit96] rely on this paradigm.

Functional specification

The principle here is to specify a system as a structured collection of mathematical functions. Two approaches may be distinguished.

Algebraic specification. The functions are grouped by object types that appear in their domain or codomain, thereby defin-

ing algebraic structures (or abstract data types). The properties of interest are then specified as conditional equations that capture the effect of composing functions (typically, compositions with type generators).

Languages such as OBJ [Fut85], ASL [Ast86], PLUSS [Gau92] or LARCH [Gut93] rely on this paradigm.

Higher-Order Functions. The functions are grouped into logical theories. Such theories contain type definitions (possibly by means of logical predicates), variable declarations, and axioms defining the various functions in the theory. Functions may have other functions as arguments which significantly increases the power of the language. Languages such as HOL [Gor93] or PVS [Cro95, Owr95] rely on this paradigm.

Operational specification

At the extreme opposite, a system may be characterized as a structured collection of processes that can be executed by some more or less abstract machine. Early languages such as Paisley [Zav82], GIST [Bal82], Petri nets or process algebras [Hoa85, Mil89] rely on this paradigm.

4. HOW GOOD IS MY FAVORED TECHNIQUE?

Specification techniques may be evaluated and compared against a number of criteria. Unsurprisingly, some of these criteria are interdependent and even conflicting; the choice of a reasonable compromise thus depends on the specifier's priorities for the task and system at hand.

Expressive power and level of coding required. As noted before, each paradigm above has some built-in semantic bias in order to be useful. State-based and functional specifications focus on sequential behaviors while providing rich structures for defining complex objects. They are thus better targeted at transactional systems. Conversely, history-based, transition-based specifications and operational specifications focus on concurrent behaviors while providing only fairly simple structures for defining the objects being manipulated. They are thus better targeted at reactive systems. There are, of course, hybrid approaches that attempt to recover from this, e.g., [Fau92, Geo95].

Beyond such semantic bias, the formal language should allow the properties of interest to be expressed without too much hard coding. Specification is about defining problems, not about programming solutions. Ideally, there should be a simple, straightforward mapping between the natural language formulation of a property and its formal counterpart.

This is, unfortunately, rarely the case. Unlike natural language, formal languages impose limitations. For example, a first-order language makes it impossible to refer to operations as predicate arguments so that coding tricks are required to overcome the problem - such as the introduction of auxiliary events that encode the application of operations. Most languages are weak at supporting temporal referencing; explicit or implicit time references occur frequently in natural formulations. For example, the built-in inability of state-based specifications to refer to the past makes it necessary to introduce auxiliary variables for encoding whether such or such event of interest has occurred, with corresponding update operations to be specified at each state modification (as in imperative programming). History-based specifications are the main exception to this problem. However they may also be problematic for specifying relative orderings of events; e.g., [Dwy99] gives an example of a relatively simple ordering property that requires six levels of operator nesting in linear temporal logic! Algebraic specifications are among those which require the most coding expertise; experience reveals that many novice specifiers incorrectly write fairly simple operations such as deleting an element from a set, because of the distance between their intuition of what this operation is about and the required delete/add commutativity axioms.

Due to language expressiveness problems, specification coding may require a lot of expertise; in the end it makes it questionable whether or not the specification correctly captures the target properties of interest.

Constructibility, manageability and evolvability. The specification technique should provide facilities for building complex specifications in a piecewise, incremental way. Local changes in problem features should be reflected by local changes in the specification. These requirements depend on (a) language mechanisms for specification structuring and compositional reasoning, and (b) the availability of a method for incremental construction, analysis and modification.

Many languages support basic structuring mechanisms for modularizing specifications - such as encapsulation, genericity, inheritance, inclusion, enrichment, etc. State-based and functional languages are probably the richest in that respect.

Some languages also support refinement relationships as a basis for incremental specification development and analysis, e.g., data reification [Jon90, Abr96], component composition/decomposition through logical connectors [Spi92, Aba95], state composition/decomposition [Har87, Lev94], or goal abstraction/refinement [Dar96].

Usability. It should be possible for reasonably well-trained people to *write* high-quality specifications. This soft, higher-level criterion of course depends on all previous ones plus a few more. The language should have a simple theoretical basis. This probably explains the popularity of languages built on simple, well-understood mathematical notions such as sets, relations and functions [Abr80, Spi92, Abr96, Owr95]. The language should also exempt users from intricacies such as, e.g., the need in state-based specifications to specify that "nothing else changes" through additional frame axioms [Bor95].

Communicability. Conversely, the technique should be accessible for reasonably well-trained people to *read* high-quality specifications and check them. This criterion depends on the previous ones (notably, the closeness between the specification and its corresponding natural language formulation), and on the external format the specification may take. It explains the popularity of techniques that support tabular formats [Hen80, Lev94, Par95, Cro95, Heit96] and diagrammatic notations [Har87, Lev94].

Powerful and efficient analysis. The effectiveness of a formal specification technique depends on the degree of satis-

faction of the various objectives mentioned in Section 1. In particular, there is no much sense writing formal specifications without being rewarded by feedback from automated tools. The latter should ideally support a wide range of analysis in the space of possibilities listed in Section 1. With a few notable exceptions (e.g., [Hei98b]) this has mostly been wishful thinking so far. Favoring one kind of analysis or another usually dictates the choice of one specification technique or another.

The more efficient the analysis is, the more coding effort is usually required on the specifier's side. This is the case for specification animation based on executing operational specifications or on term rewriting of algebraic specifications. Model checkers illustrate this as well; the unconvinced reader may look at what their input code for a complex application may look like.

On another hand, the more powerful the analysis is, the more expert intervention is usually required. Proof assistants are a good illustration of this unsurprising fact [Cro95].

It should become clear from our brief review of evaluation criteria that any multicriteria analysis will inevitably result in favoring a multiparadigm framework in which complementary formalisms, methods and tools are integrated in a coherent way so as to combine the best of each paradigm for specific domains, tasks, and concerns. Very preliminary attempts have started in this direction [Nis89, Dar93, Nus93, Zav93, Zav96].

5. TODAY'S GOOD NEWS

The number of success stories in using formal specifications for real systems is steadily growing from year to year. They range from to the reengineering of existing systems (e.g., [Hen80, Crai93]) to the development of new systems (e.g., [Hal96, Beh99]). In the latter case, there was some reported evidence that the development, while resulting in products of much higher quality, did not incur higher costs but rather the contrary. Although many of the stories are in the domain of transportation systems, there are other domains such as information systems, telecommunication systems, power plant control, protocols and security. Good accounts can be found in [Cra93, Hin95, Cla96, SCP2K].

A recent, fairly impressive example is worth pointing out [Beh99]. The Paris metro system has recently opened a new line (line 14, Tolbiac-Madeleine). The traffic on this line is entirely controlled by software. Driverless trains and conventional trains are both supported. The safety-critical components of the software (located on board, along the track, and on ground) were formally developed by Matra Transport using the B abstract machine method [Abr96]. The development includes abstract models of those components, refinements to concrete models, and automated translation to ADA code. According to [Beh99], there are about 100,000 lines of B specification, covering the abstract and the concrete model, and 87,000 lines of ADA code. The refinement was entirely validated by formal proofs. The B tool automatically proved 28,000 lemmas and 65% of the rules added to discharge proofs. Many errors were found thereby, and fixed in the concurrent development. In addition, a conventional testing process was deployed and not a single error was found.

The success of this formal development might be explained by the unusual combination of success factors. The B specification language has a simple mathematical basis that allows engineers to use it after a reasonably short period of training; the specification technique is multi-level and makes it possible to smoothly move from an abstract model up to code in a provably correct way; methodological support was provided in the form of guidelines and heuristics to guide the development and validation processes; a development/validation process model was first designed explicitly and integrated in the company's process model to accommodate conventional practices such as testing (the lack of such integration has been recognized to be a serious obstacle to the adoption of formal methods [Cra95]); last but not least, the process was supported by powerful tools.

The maturity of specification tool technology is also steadily growing from year to year. Tools become more effective in analyzing formal specifications and deriving useful information; their performance on large specifications keeps increasing; they become more usable. Specification animators and model checkers are particularly successful in those respects. Moreover there is a promising tendency towards integrating multiple tools so as to offer a wide spectrum of analysis at various costs - from fully automatic, dedicated checks to interactive assistance in difficult proofs. The SCR toolset is a good illustration of this recent trend [Hei98b].

6. TODAY'S BAD NEWS

In spite of such good news, today's formal specification techniques suffer a number of weaknesses. Some of these explain why in their present form they are inadequate for the upstream critical phase of requirements specification and analysis.

- ***Limited scope***. The vast majority of techniques are limited to the specification of functional properties, that is, properties about what the target system is expected to do. Non-functional properties are in general left outside any kind of formal treatment. The main exception are techniques allowing timing properties to be formalized and reasoned about.
- ***Poor separation of concerns***. Most techniques provide no support for making a clear separation between (a) intended properties of the system considered, (b) assumptions about the environment of this system, and (c) properties of the application domain. One cannot therefore make the essential distinction between descriptive and prescriptive properties (called "indicative" and "optative" in [Zav97]); they are all mixed together in the specification.
- ***Low-level ontologies.*** The concepts in terms of which problems have to be structured and formalized are programming concepts - most often, data and operations. It is time to raise the level of abstraction and conceptual richness found in informal requirements documents - such as, e.g., goals and their refinements, agents and their responsibilities, alternatives, and so forth [Fea87, Fic92, Dar93, Myl98, Myl99].

- ***Isolation***. With a few exceptions mentioned before, formal specification techniques are isolated from other software products and processes both vertically and horizontally. *Vertical isolation:* specification techniques generally pay no attention to what upstream products in the software lifecycle the formal specification is coming from (viz. goals, requirements, assumptions) nor what downstream products the formal specification is leading to (viz. architectural components). *Horizontal isolation:* the techniques generally do not pay attention to what companion products the formal specification should be linked to (e.g., the corresponding informal specification, a documentation of choices, validation data, project management information, etc.).

- ***Poor guidance***. The main emphasis in the formal specification literature has been on suitable sets of notations and on *a posteriori* analysis of specifications written using such notations. Constructive methods for building correct specifications for complex systems in a safe, systematic, incremental way are by and large non-existent. Instead of inventing more and more languages, one should put more effort in devising and validating methods for elaboration and modification of good specifications (in the sense recalled in Section 1).

- ***Cost.*** Many formal specification techniques require high expertise in formal systems in general (and mathematical logic in particular), in analysis techniques, and in the white-box use of tools. Due to the scarcity of such expertise their use in industrial projects is nowadays still highly limited in spite of the promised benefits.

- ***Poor tool feedback***. Many analysis tools are effective at pointing out problems, but in general they do a poor job of (a) suggesting causes at the root of such problems, and (b) proposing recovery actions.

7. BACK TO THE FUTURE

The discussion above provides the material for paving the road ahead. Tomorrow's technology should meet the following requirements and challenges for formal specification to become an essential vehicle for the engineering or reengineering of higher-quality software.

- ***Constructiveness***. The almost exclusive focus on a posteriori analysis of possibly poor specifications should in part be shifted towards a more constructive approach in which specifications are built incrementally from higher-level ones in a way that guarantees high quality by construction. One could then really speak of a method, typically made of a collection of model building strategies, style selection rules, specification derivation rules, guidelines, and heuristics; some might be domain-independent, some others might be domain-specific. Such a method should provide active guidance in the specifier's decision making process. It might be supported by automated specification assistants that would provide advice at decision points and record the process followed, for documentation and possible replay in case of later evolution.

- ***Support for comparative analysis***. Experience in teaching formal specification reveals that different specifiers with the same background may end up with fairly different specifications for the same initial problem formulation. The same is true for programs, but in the latter case there is at least an ultimate moment of truth - the program is running satisfactorily or not. Beyond the specification qualities recalled in Section 1, we need precise criteria and measures for assessing specifications and comparing their relative merits.

- ***Integration***. Tomorrow's technology should care for the vertical and horizontal integration of formal specifications within the software lifecycle - from high-level goals to functional design to architectural components; and from informal formulation to formal specification to related products.

- ***Higher level of abstraction.*** Specification techniques should move from functional design to requirements engineering where the impact of errors is even more crucial. We therefore need languages, methods and tools that support richer, problem-oriented ontologies upstream to the program-oriented ones currently supported. Preliminary attempts in this direction include [Myl92, Dar96] for goal-oriented refinement, [Myl92, Lam98b] for goal-level conflict analysis, and [Lam2K] for goal-level exception handling.

- ***Richer structuring mechanisms.*** Most constructs available so far for modularizing large specifications have been lifted from programming counterparts. Problem-oriented constructs should be available as well such as, e.g., stakeholder viewpoints [Nus93] or problem views [Jac95].

- ***Extended scope***. Specification techniques need to be extended in order to cope with the various categories of non-functional properties that are elicited during requirements engineering and play a prominent role during architectural design, e.g., properties about performance, integrity, confidentiality, accuracy of information, availability, fault-tolerance, operational costs, maintainability, and so forth. The qualitative reasoning techniques in [Myl92] are a first step in this direction. Specific categories might require specific language features and analysis techniques.

- ***Separation of concerns***. As discussed before, formal specification languages should enforce a strict separation between descriptive and prescriptive properties, to be exploited by analysis tools accordingly.

- ***Lightweight techniques.*** The use of formal specifications should not require deep expertise in formal systems. The mathematical intricacies should be hidden; analysis tools should be usable like compilers. The work on pattern-based specification in [Dwy99] is a very promising step in this direction. Patterns may also be used to reuse proofs and generate specifications [Dar96, Lam2K].

- ***Multiparadigm specification***. Complex systems have multiple facets. Since no single paradigm will ever serve all purposes due to semantic biases, frameworks are needed in

which multiple paradigms can be combined in a semantically meaningful way so that the best features of each paradigm can be exploited. The various facets then need to be linked through consistency rules [Nus93]. Multiparadigm frameworks should be able to integrate various formal languages, semi-formal ones, and natural language, together with corresponding analysis techniques and tools. Preliminary linguistic attempts in this direction combine semantic nets, history-based specification, and state-based specification [Dar93]; or state-based specification and transition-based specification [Zav96]. While multilingual integration is fairly easy to achieve among semi-formal languages it raises difficult semantic issues for formal languages.

- ***Multibutton analysis***. A multiparadigm framework should support different levels of optional analysis - from cheap, surface-level analysis (such as traceability analysis, static semantics checks and qualitative reasoning) to more expensive, deep-level analysis (such as algorithmic verification, deductive reasoning, or inductive reasoning from examples). The more heavyweight buttons would be pushed only when needed and where needed. A multibutton environment would also allow end-users to use the typical facilities provided by standard CASE tools in a first stage, and then gradually enter into the more complex world of formal methods as they get more confidence.
- ***Multiformat specification***. To enhance the communicability of the same specification fragment among different types of producers/consumers, the fragment should be maintained under multiple concrete syntaxes - e.g., tabular, diagrammatic, and textual.
- ***Reasoning in spite of errors***. Many specification techniques require that the specification be complete in some sense before the analysis can start. It should be made possible to start analysis much earlier, on specification drafts [Gau92], and incrementally. This would ensure early payback and incremental gain for incremental effort - an important objective already noted in [Cla96]. On another hand, deductive techniques also assume that the specification is consistent for useful information to be derivable. Especially in the context of requirements engineering, where useful information can be inferred from conflicting viewpoints, formal systems and reasoning techniques are needed for deriving such information in spite of temporary inconsistencies [Hun98].
- ***Constructive feedback from tools.*** Instead of just pointing out problems, future tools should assist in resolving them.
- ***Support for evolution***. In general, requirements keep evolving while some core architecture is expected to remain stable. A more constructive approach should also help managing the evolution of formal specifications under such constraints.
- ***Support for reuse***. Problems in the domain considered are more likely to be similar than solutions. Specification reuse should therefore be even more promising than code reuse. Surprisingly enough, techniques for retrieving, adapting, and consolidating reusable specifications have received relatively little attention so far (see, e.g., [Zar97] for some recent work in this direction). A constructive approach to formal specification should also favor the reuse of specifications that proved to be good and effective for similar systems.
- ***Measurability of progress***. To be more convincing, the benefits of using formal specifications in software engineering should be measurable thanks to metrics similar to those used for measuring increase in software productivity.

8. CONCLUSION

Software is increasingly invading many aspects of our life. We increasingly need high-quality software. Formal specifications offer a wide spectrum of possible paths towards that goal. Therefore they are receiving increasing attention in the academia and the industry. Still, there is a long way to go before formal specifications can be used by the average software engineer to provide reasonably fast and visible reward. Among the many challenges raised, we believe that the critical success factors will be the provision of constructive assistance in specification development, analysis, and evolution; the vertical and horizontal integration of formal specifications within the software lifecycle; higher-level abstractions for requirements specification and analysis; the availability of formal techniques for non-functional aspects; and lightweight interfaces for multiparadigm specification and analysis.

Acknowledgment.

Many thanks to Michel Sintzoff for fruitful input and discussions on some issues raised in this paper.

REFERENCES

[Aba95] M. Abadi and L. Lamport, “Conjoining Specifications”, *ACM Transactions on Programming Languages and Systems* Vol. 17 No. 3, May 1995, 507-535.

[Abr80] J.R. Abrial, “The Specification Language Z: Syntax and Semantics”. Programming Research Group, Oxford Univ., 1980.

[Abr96] J.R. Abrial, *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[All89] J.F. Allen and P.J. Hayes, “Moments and Points in an Interval-Based Temporal Logic”, *Computational Intelligence*, Vol. 5, 1989, 225-238.

[Ast86] Astesiano, E., Wirsing, M., “An introduction to ASL”, *Proc. IFIP WG2.1 Conf. on Program Specifications and Transformations*, North-Holland, 1986.

[Atl93] J.M. Atlee, State-Based Model Checking of Event-Driven System Requirements, IEEE Transactions on Software Engineering Vol. 19 No. 1, January 1993, 24-40.

[Bal82] R.M. Balzer, N.M. Goldman, and D.S. Wile, “Operational Specification as the Basis for Rapid Prototyping”, *ACM SIGSOFT Softw. Eng. Notes* Vol. 7 No. 5, Dec. 1982, 3-16.

[Beh99] P. Behm, P. Benoit, A. Faivre and J.M. Meynadier, “Météor: A Successful Application of B in a Large Project”, *Proc. FM-99 - World Conference on Formal Methods in the Development of Computing Systems*, LNCS 1708, Springer-Verlag, 1999, 369-387.

[Ben96] S. Bensalem, Y. Lakhnech and H. Saïdi, “Powerful Techniques for the Automatic Generation of Invariants”, Proc. CAV’96 - 8th Intl Conference on Computer-Aided Verification, LNCS 1102, Springer-Verlag, 1996, 323-335.

[Ber91] G. Bernot, M.C. Gaudel, ad B. Marre, “Software Testing Based on Formal Specifications: A Theory and a Tool”, *Software Engineering Journal*, 1991.

[Bor95] A. Borgida, J. Mylopoulos and R. Reiter, "On the Frame Problem in Procedure Specifications", *IEEE Transactions on Software Engineering*, Vol. 21 No. 10, October 1995, 785-798.

[Car90] C. Morgan, *Programming from Specifications*. Prentice Hall, 1990.

[Cla86] E.M.Clarke and E.A. Emerson, “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”, *ACM Trans. Program. Lang. Systems* Vol. 8 No. 2, 1986, 244-263.

[Cla96] E.M. Clarke, J.M. Wing et al, “Formal Methods: State of the Art and Future Directions”, *ACM Computing Surveys Vol. 28 No. 4*, December 1996, 626-643.

[Cla99] E.M. Clarke, O. Grumberg, and D.A. Peled, *Model Checking*. MIT Press, 1999.

[Cra93] D.Craigen, S. Gerhart and T. Ralston, An International Survey of Industrial Applications of Formal Methods. US Dept. Commerce, NIST, Computer Systems Lab., NISTGCR 93/626, March 1993.

[Cra95] D. Craigen, S. Gerhart and T. Ralston, “Formal Methods Technology Transfer: Impediments and Innovation”, in *Applications of Formal Methods*, M.G. Hinchey and J.P. Bowen (eds.), Prentice Hall, 1995, 399-419.

[Cro95] J. Crow, S. Owre, J. Rushby, N. Shankar, M. Srivas, “A Tutorial Introduction to PVS”. *Proc. WIFT’95 - Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, April 1995. http://www.csl.sri.com/sri-csl-fm.html.

[Dar93] A. Dardenne, A. van Lamsweerde and S. Fickas, “Goal-Directed Requirements Acquisition”, *Science of Computer Programming*, Vol. 20, 1993, 3-50.

[Dar96] R. Darimont and A. van Lamsweerde, “Formal Refinement Patterns for Goal-Driven Requirements Elaboration”, *Proc. FSE’4 - Fourth ACM SIGSOFT Symp. on the Foundations of Software Engineering*, San Francisco, October 1996, 179-190.

[Dij75] E.W. Dijkstra, “Guarded commands, nondeterminacy and the formal derivation of programs”, *Comm. ACM* Vol. 18, August 1975, 453-457.

[Doug94] J. Douglas and R.A. Kemmerer, “Aslantest: A Symbolic Execution Tool for Testing ASLAN Formal Specifications”, *Proc. ISTSTA '94 - Intl. Symp. on Software Testing and Analysis*, ACM Softw. Eng. Notes, 1994, 15-27.

[Dub91] Dubois, E., Hagelstein, J., Rifaut, A., “A Formal Language for the Requirements Engineering of Computer Systems”, in *Introducing a Logic Based Approach to Artificial Intelligence*, A. Thayse (Ed.), Vol. 3, Wiley, 1991, 357-433.

[Dub93] E. Dubois, Ph. Du Bois and M. Petit, "Object-Oriented Requirements Analysis: An Agent Perspective", Proc. ECOOP’93 - 7th European Conf. on Object-Oriented Programming, Springer-Verlag LNCS 707, 1993, 458-481.

[Dwy99] M.B. Dwyer, G.S. Avrunin and J.C. Corbett, “Patterns in Property Specifications for Finite-State Verification”, *Proc. ICSE-99: 21th Intrnational Conference on Software Enginering*, Los Angeles, 411-420.

[Eme86] E.A. Emerson and J.Y. Halpern, ““Sometime” and “not Never” Revisited: on Branching versus Linear Time Temporal Logic”, *Journal of the ACM* Vol. 33 No. 1, 1986, 151-178.

[Ern99] M.D. Ernst, J. Cockrell, W.G. Griswold, and D. Notkin, “Dynamically Discovering Likely Program Invariants to Support Program Evolution”, *Proc. ICSE-99: 21th Intrnational Conference on Software Enginering*, Los Angeles, 213-224.

[Fau92] S. Faulk, J. Brackett, P. Ward and J. Kirby, “The CORE Method for Real-Time Requirements”, *IEEE Software*, September 1992, 22-33.

[Fea87] M. Feather, “Language Support for the Specification and Development of Composite Systems”, *ACM Trans. on Programming Languages and Systems* 9(2), Apr. 87, 198-234.

[Fea98] M. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, “Reconciling System Requirements and Runtime Behaviour”, *Proc. IWSSD’98 - 9th International Workshop on Software Specification and Design*, Isobe, IEEE CS Press, April 1998.

[Fic92] S. Fickas and R. Helm, “Knowledge Representation and Reasoning in the Design of Composite Systems", *IEEE Trans. on Software Engineering*, June 1992, 470-482.

[Flo67] R. Floyd, “Assigning Meanings to Programs”, In. Mathematical Aspects of Computer Science, *Proc. Symp. Appl. Maths., Vol. 19, American Mathematical Society*, 1967, 19-32.

[Fut85] K. Futatsugi, J. Goguen, J.-P. Jounnaud, and J. Mesguer, “Principles of OBJ”, *Proc. POPL’85 - ACM Symposium on Principles of Programming Languages*, 1985, 52-66.

[Gan96] G.C. Gannod and B.H. Cheng, “Strongest Postcondition Semantics as the Formal Basis for Reverse Engineering”, *Journal of Automated Software Engineering* Vol. 3, June 1996, 139-164.

[Gau92] M.-C. Gaudel, “Structuring and Modularizing Algebraic Specifications: the PLUSS specification language, evolutions and perspectives”, *Proc. STAS’92*, LNCS 557, 1992, 3-18.

[Ghe91] C. Ghezzi and R.A. Kemmerer, “ASTRAL: An Assertion Language for Specifying Real-Time Systems”, *Proc. ESEC’91 - 3rd European Softwre Engineering Conference*, LNCS 550, Springer-Verlag, 1991.

[Geo95] C. George, A.E. Haxthausen, S. Hughes, R. Milne S. Prehn and J.S. Pedersen, *The RAISE Development Method*. Prentice Hall, 1995.

[Gor93] M. Gordon and T.F. Melham, *Introduction to HOL*. Cambridge University Press, 1993.

[Gre86] S.J. Greenspan, A. Borgida and J. Mylopoulos, “A Requirements Modeling Language and its Logic”, *Information Systems* Vol. 11 No. 1, 1986, 9-23.

[Gri81] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[Gut93] J.V. Guttag and J.J. Horning, *LARCH: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Hal95] R.J. Hall, “Systematic Incremental Validation of Reactive Systems via Sound Scenario Generalization”, *Automated Software Engineering*, Vol. 2, 1995, 131-166.

[Hal96] A.Hall, “Using Formal Methods to Develop an ATC Information System”, *IEEE Softwar*e Vol. 12 No. 6, March 1996, 66-76.

[Hal98] R.J. Hall, “Explanation-Based Scenario Generation for Reactive System Models”, *ASE’98*, Hawaii, Oct. 1998.

[Han91] K.M. Hansen, A.P. Ravn and H. Rischel, “Specifying and Verifying Requirements of Real-Time Systems”, *Proc. ACM SIGSOFT’91 Conference on Software for Critical Systems*, New Orleans, December 1991.

[Har87] D. Harel, “Statecharts: A Visual Formalism for Complex Systems”, *Science of Computer Programming* Vol. 8, 1987, 231-274.

[Har90] D.Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems”, *IEEE Transactions on Software Engineering*, Vol. 16 No. 4, April 1990, 403-414.

[Heim96] M.P. Heimdahl and N.G. Leveson, "Completeness and Consistency in Hierarchical State-Based Requirements", *IEEE Transactions on Software Engineering* Vol. 22 No. 6, June 1996, 363-377.

[Heit96] C. Heitmeyer, R. Jeffords and B. Labaw, "Automated Consistency Checking of Requirements Specificatons", *ACM Transactions on Software Engineering and Methodology* Vol. 5 No. 3, July 1996, 231-261.

[Hei98a] C. Heitmeyer, J. Kirkby, B. Labaw, M. Archer and R. Bharadwaj, “Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specifications”, *IEEE Transactions on Software Engineering* Vol. 24 No. 11, November 1998, 927-948.

[Hei98b] C. Heitmeyer, J. Kirkby, B. Labaw, and R. Bharadwaj, “SCR*: A Toolset for specifying and Analyzing Software Requirements”, *Proc. CAV’98 - 10th Annual Conference on Comuter-Aided Verification*, Vancouver, 1998, 526-531.

[Hek88] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods, and VDM*. Addison-Wesley, 1988.

[Hen80] K.L. Heninger, “Specifying Software Requirements for Complex Systems: New Techniques and their Application”, *IEEE Transactions on Software Engineering* Vol. 6 No. 1, January 1980, 2-13.

[Hin95] M.G. Hinchey and J.P. Bowen (eds.), *Applications of Formal Methods.* Prentice Hall, 1995

[Hoa69] C.A.R. Hoare, “An Axiomatic Basis for Computer Programming”, *Comm. ACM* Vol. 12 No. 12 No. 10, Oct. 1969, 576-583.

[Hoa85] C.A.R.Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hol91] G.Holzman, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[Hol97] G. Holzman, "The Model Checker SPIN", *IEEE Trans. on Software Engineering* Vol. 23 No. 5, May 1997, 279-295.

[Hun98] A. Hunter and B. Nuseibeh, “Managing Inconsistent Specifications: Reasoning, Analysis and Action”, *ACM Transactions on Software Engineering and Methodology*, Vol. 7 No. 4. October 1998, 335-367.

[Jac93] M. Jackson and P. Zave, “Domain Descriptions”, *Proc. RE’93 - 1st Intl. IEEE Symp. on Requirements Engineering*, Jan. 1993, 56-64.

[Jac95] D. Jackson, “Structuring Z Specifications with Views”, *ACM Transactions on Software Engineering and Methodology* Vol. 4 No. 4, October 1995, 365-389.

[Jac96] D. Jackson and C.A. Damon, Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector, *IEEE Transactions on Software Engineering* Vol. 22 No. 7, July 1996, 484-495.

[Jah86] F. Jahanian and A.K. Mok, “Safety Analysis of Timing Properties in Real-Time Systems”, *IEEE Transactions on Software Engineering*, Vol. 12, September 1986, 890-904.

[Jef98] R. Jeffords and C. Heitmeyer, “Automatic Generation of State Invariants from Requirements Specifications”, *Proc. FSE-6: 6th ACM SIGSOFT Intl Symposium on the Foundations of Software Engineering*, Lake Buena Vista, 1998, 56-69.

[Jon90] Jones, C.B., *Systematic Software using VDM*, 2nd ed., Prentice Hall, 1990.

[Jon93] A.J. Jones and M. Sergot, “On the Characterization of Law and Computer Systems: the Normative System Perspective”, *in* J.Ch. Meyer and R.J. Wieringa (Eds.), *Deontic Logic in Computer Science - Normative System Specification*, Wiley, 1993.

[Kat87] S. Katz, C.A. Richter, K.S. The, “PARIS: A System for Reusing Partially Interpreted Schemas”, *Proc. ICSE-87: 9th Intrnational Conference on Software Enginering*, Monterey, CA, March 1987, 377-385.

[Koy92] R. Koymans, *Specifying message passing and time-critical systems with temporal logic*, LNCS 651, Springer-Verlag, 1992.

[Lam79] A. van Lamsweerde and M. Sintzoff, “Formal Derivation of Strongly Correct Concurrent Programs”, Acta Informatica Vol. 12, 1979, 1-31.

[Lam98a] A. van Lamsweerde and E. Letier, “Integrating Obstacles in Goal-Driven Requirements Engineering”, *Proc. ICSE-98: 20th Intrnational Conference on Software Enginering*, Kyoto, April 1998.

[Lam98b] A. van Lamsweerde, R. Darimont and E. Letier, "Managing Conflicts in Goal-Driven Requirements Engineering", *IEEE Trans. on Sofware. Engineering*, Special Issue on Inconsistency Management in Software Development, November 1998.

[Lam98c] A. van Lamsweerde and L. Willemet, "Inferring Declarative Requirements Specifications from Operational Scenarios", *IEEE Trans. on Sofware. Engineering*, Special Issue on Scenario Management, December 1998, 1089-1114.

[Lam2K] A. van Lamsweerde and E. Letier, “Handling Obstacles in Goal-Oriented Requirements Engineering”, *IEEE Transactions on Software Engineering*, Special Issue on Exception Handling, 2000.

[Lamp94] L. Lamport, “The Temporal Logic of Actions”, *ACM Transactions on Programming Languages and Systems* Vol. 16 No. 3, May 1994, 872-923.

[Lan95] Lano, K., *Formal Object-Oriented Development*, Springer-Verlag, 1995.

[Lev94] N.G. Leveson, M.P. Heimdahl and H. Hildtreth, “Requirements Specification for Process-Control Systems”, *IEEE Transactions on Software Engineering* Vol. 20 No. 9, September 1994, 684-706.

[Lis75] B.H. Liskov and S.N. Zilles, “Specification Techniques for Data Abstractions”, *IEEE Transactions on Software Engineering* Vol. 1. No. 1, March 1975, 7-18.

[MaM95] D. Mandrioli, S. Morasca, A. Morzenti, "Generating test cases for real-time systems from logic specifications", *ACM*

Transactions on Computer Systems, Vol.13 No.4, Nov. 1995, pp.365-398.

[Man92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1992.

[Man96] Z. Manna and the STep Group, "STeP: Deductive-Algorithmic Verification of Reactive and Real-Time Systems", *Proc. CAV'96 - 8th Intl. Conf. on Computer-Aided Verification*, LNCS 1102, Springer-Verlag, July 1996, 415-418.

[Mas97] P. Massonet and A. van Lamsweerde, "Analogical Reuse of Requirements Frameworks", Proc. RE-97 - *3rd Int. Symp. on Requirements Engineering*, Annapolis, 1997, 26-37.

[McM93] K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer, 1993.

[Mey85] B. Meyer, "On Formalism in Specifications", *IEEE Software*, Vol. 2 No. 1, January 1985, 6-26.

[Mil89] R.Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[Mor92] A. Morzenti, D. Mandrioli, and C. Ghezzi, "A Model Parametric Real-Time Logic", *ACM Transactions on Programming Languages and Systems*, Vol. 14 No. 4, October 1992, 521-573.

[Mos97] L. Moser, Y. Ramakrishna, G. Kutty, P.M. Melliar-Smith and L. Dillon, "A Graphical Environment for the Design of Concurrent Real-Time Systems", *ACM Transactions on Software Engineering and Methodology*, Vol. 6 No. 1, January 1997, 31-79.

[Myl92] Mylopoulos, J., Chung, L., Nixon, B., "Representing and Using Nonfunctional Requirements: A Process-Oriented Approach", *IEEE Trans. on Sofware. Engineering*, Vol. 18 No. 6, June 1992, pp. 483-497.

[Myl98] J. Mylopoulos, "Information Modeling in the Time of the Revolution", Invited Review, Information Systems Vol. 23 No. 3/4, 1998, 127-155.

[Myl99] J. Mylopoulos, L. Chung and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis", Communications of the ACM, Vol. 42 No. 1, January 1999, 31-37.

[Nau69] P. Naur, "Proofs of algorithms by General Snapshots", *BIT* Vol. 6, 1969, 310-316.

[Nis89] C. Niskier, T. Maibaum and D. Schwabe, "A Pluralistic Knowledge-Based Approach to Software Specification", *Proc. ESEC-89 - 2nd European Software Engineering Conference*, LNCS 387, September 1989, 411-423.

[Nus93] B. Nuseibeh, J. Kramer and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specifications", *IEEE Transactions on Software Engineering*, Vol. 20 No. 10, October 1994, 760-773.

[Owr95] S. Owre, J. Rushby, and N. Shankar, "Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS", *IEEE Transactions on Software Engineering* Vol. 21 No. 2, Feb. 95, 107-125.

[Par72] D.L.Parnas, "A Technique for Software Module Specification With Examples", *Comm. ACM* Vol. 15, May 1972.

[Par77] D.L. Parnas, "The Use of Precise Specifications in the Development of Software", *Proc. IFIP'77 - Information Processing 77*, North Holland, 1977, 849-867.

[Par95] D.L. Parnas and J. Madey, "Functional Documents for Computer Systems", *Science of Computer Programming*, Vol. 25, 1995, 41-61.

[Par98] D.Y. Park, J. Skakkebaek, and D.L. Dill, "Static Analysis to Identify Invariants in RSML Specifications", *Proc. FTRTFT'98 - Formal Techniques for Real Time or Fault Tolerance*, 1998.

[Pnu77] A. Pnueli, "The Temporal Logics of Programs", *Proc. 18th IEEE Symp. on Foundations of Computer Science*, 1977, 46-57.

[Pot96] B. Potter, J. Sinclair and D. Till, *An Introduction to Formal Specification and Z*. Second edition, Prentice Hall, 1996.

[Que82] J. Queille and J. Sifakis, "Specification and Verification of Concurrent Systems in CAESAR", *Proc. 5th International Symposium on Programming*, LNCS 137, 1982.

[Ran73] B. Randell, *The Origin of Digital Computers*. Springer-Verlag, 1973.

[Reu91] H.B. Reubenstein and R.C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Transactions on Software Engineering*, Vol. 17 No. 3, March 1991, 226-240.

[Ric92] D.J. Richardson, S. Leif Aha, T.O. O'Malley, "Specification-based test oracles for reactive systems", *International Conference on Software Engineering*, Melbourne, Australia, 11-15 May 1992. ACM, 1992, pp.105-118.

[Roo94] D. Roong-Ko, P.G. Frankl, "The ASTOOT approach to testing object-oriented programs", *ACM Transactions on Software Engineering and Methodology*, Vol.3, No.2, April 1994, pp.101-130.

[SCP2K] Science of Computer Programming, Special Issue on *Formal Methods in Industry*, Vol. 36 No. 1, January 2000.

[Sou93] J. Souquières and N. Levy, "Description of Specification Developments", *Proc. RE'93 - First IEEE Symposium on Requirements Engineering*, San Diego, 1993, 216-223.

[Spi92] J.M. Spivey *The Z Notation - A Reference Manual*. Second Edition, Prentice Hall, 1992.

[SRS79]*Proceedings SRS - Specification of Reliable Software*. IEEE Catalog No. 79 CH1401-9C, 1979.

[Swa82] W. Swartout and R. Balzer, "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, Vol. 25 No. 7, July 1982, 438-440.

[Tho99] J.M. Thompson, M.E. Heimdahl, and S.P. Miller, "Specification-Based Prototyping for Embedded Systems", *Proc. ESEC/FSE'99*, Toulouse, ACM SIGSOFT, LNCS 1687, Springer-Verlag, 1999, 163-179.

[Wey94] E. Weyuker, T. Goradia, A. Singh, "Automatically generating test data from a Boolean specification", *IEEE Transactions on Software Engineering*, Vol.20, No.5, May 1994, pp.353-363.

[Win90] J.M. Wing, "A Specifier's Introduction to Formal Methods", *IEEE Computer* Vol. 23 No. 9, September 1990.

[Win99] J.M. Wing, J. Woodcock and J. Davies (eds.), *FM-99 - World Conference on Formal Methods in the Development of Computing Systems*, LNCS 1708 and 1709, Springer-Verlag, 1999.

[Yue87] K. Yue, "What Does It Mean to Say that a Specification is Complete?", *Proc. IWSSD-4, Fourth International Workshop on Software Specification and Design*, Monterey, 1987.

[Zar97] A.M. Zaremski and J. Wing "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, Vol. 6 No. 4, October 1997, 333-

369.

[Zav82] P. Zave, “An Operational Approach to Requirements Specification for Embedded Systems”, *IEEE Transactions on Software Engineering*, Vol. 8 No. 3, May 1982, 250-269.

[Zav93] P. Zave and M. Jackson, “Conjunction as Composition”, *ACM Transactions on Software Engineering and Methodology*, Vol. 2 No. 4, October 1993, 379-411.

[Zav96] P. Zave and M. Jackson, “Where Do Operations Come From? A Multiparadigm Specification Technique”, *IEEE Transactions on Software Engineering*, Vol. 22 No. 7, July 1996, 508-528.

[Zav97] P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering", *ACM Transactions on Software Engineering and Methodology*, 1997, 1-30.