Martin D. Fraser, Kuldeep Kumar, Vijay K. Vaishnavi

Strategies for Incorporating Formal Specifications in Software Development

A **formal software specification is a specification expressed in a language** whose vocabulary, syntax, and semantics are formally defined, and which has a mathematical, usually formal logic, basis. (See the box "A Sketch of Formal Specification" for a short introduction to formal specification methods. For a more detailed introduction to formal methods see [24].)

The growing importance of formal specification methods is reflected in recently published special issues of journals dealing with these methods (e.g., *IEEE Software* and *IEEE Transactions on Software Engineering*.) Furthermore, a number of authors have advocated the use of formal specification methods in the software development process. Included among the justifications suggested by these authors are that the use of formal specification methods in software development enhances the insight into and understanding of software requirements, helps clarify the customer's requirements by revealing or avoiding contradictions and ambiguities in the specifications, enables rigorous verification of specifications and their software implementations, and facilitates the transition from specification and design to implementation. Thus the use of formal methods is expected to lead to increased software quality and reliability. Moreover, early verification of specifications would increase specification quality thereby reducing life cycle costs. Hall [12] suggests that benefits of using formal specifications are obtainable without an increase in, and possibly at lower, development costs.

However, Sommerville [23] has indicated that formal specification methods have not been widely accepted in industrial software development. One of the main reasons for the lack of use of formal specifications in industrial projects is that reports about formal specifications have dealt mostly with the languages, not with the elicitation process [6, 9, 18]. Recently, however, a number of strategies have been proposed for incorporating formal specifications methods into the software development process. With the proliferation of these strategies we have reached a stage in the development of formal specification methods where there is a need for organizing and classifying the proposed strategies. Such an organization is a necessary step for understanding the potential usefulness of these strategies, identifying commonalities and differences among them, and assessing their applicability to different contexts.

A definition and classification scheme for strategies that incorporate formal specification methods into the software development process is useful from both a practitioner as well as a researcher perspective. From a practitioner perspective, the framework will help practicing software engineers make sense of competing proposals for using formal methods

in the software development process, and help them identify the strengths and weaknesses of each proposal. Furthermore, a deeper understanding of these strategies will help the practitioners adopt and adapt the proposed strategies to their own software development environments.

From a researcher perspective, an understanding and assessment of the generic strategies suggested by the framework would help identify gaps in the currently proposed strategies, thus suggesting directions for future development. Additionally, a comprehensive classification scheme would provide the basis for further empirical investigations of the effectiveness (i.e., quality and efficiency consequences) of each strategy.

Benefits and Problems with Formal Specifications

A variety of advantages have been attributed to the use of formal software specifications. These advantages include enhanced insight into and understanding of specifications [23, 24], help in verification of the specifications and their programming implementations [12, 14, 15, 24], and possible assistance in moving from requirements specification to their programming implementation [9].

The enhanced insight and understanding into specifications is achieved in a number of ways. Wing suggests that formal specifications help crystallize the customer's vague ideas, and reveal or avoid contradictions, ambiguities, and incompleteness in the specifications thereby helping clarify the customer's requirements [24]. Sommerville suggests that formal specifications, by providing a unified and concise view of the syntactic and semantic aspects of the specifications, provide insights into and understanding of the software requirements, insights which are not normally possible from informal specifications [23]. Finally, depending on the formal specification language used, it may be possible to animate a formal system specification to provide a prototype system [23]. The prototype can be used by both requirements engineers and end users to gain further insights into the behavior of the specified system.

Second, as formal specifications can be analyzed using mathematical operators, mathematical proof procedures can be used to test (and prove) internal consistency and syntactic correctness of specifications [9, 14, 15, 24]. Furthermore, the completeness of the specifications can be checked in the sense that all enumerated options and elements have been specified.[1] The formal proof procedures can also be used to verify if a design or its implementation satisfies its antecedent specifications [1, 24].

Third, from an implementation point of view, as the final problem solution—the implementation—will be in a formal language (i.e., programming language), it is easier to avoid misconceptions and ambiguities in crossing the divide from formal specifications to formal implementations [1]. Given a formal system specification and a complete formal programming language definition, it becomes possible to prove that the programming implementation conforms to its specification [23]. Furthermore, compilers for rapid prototyping and for transforming specifications into code have been investigated for some traditional formal specification languages such as VDM/Meta-IV. This raises the possibility of automatic code generation from formal specifications. Finally, formal specifications can be used as a guide to the testers of software components in identifying and generating appropriate test cases. Thus, the use of formal methods can lead to higher-quality specifications and implementations.

Sommerville suggests that formal specification techniques have not been widely used in industrial software development environments [23]. (For an account of projects where formal methods have been used successfully in industrial environments, see [12].) A number of reasons by various authors have been suggested for this lack of use:

1. In the past, most effort in formal specification research has been concerned with the development of formal notation and inference rules. Relatively little effort has been devoted to the development of methodological and tool support [18]. This lack of methodology and tools has two adverse consequences. First, though the current state of formal methods provide elaborate notational and conceptual structures to express the formal specifications, only minimal guidelines are provided for eliciting and structuring the requirements before they are expressed in the formal notation [14]. As a result, the software engineer is left to his or her own devices to discover the software requirements and structure them into a requirements architecture. Lack of guidance for this crucial front-end activity makes it difficult to use formal methods on their own. Second, the lack of tool support makes it difficult to develop, analyze, and process large-scale specifications using formal specification languages.

2. The notation and the conceptual grammar of formal specification languages require familiarity with discrete mathematics and symbolic logic which most practicing software engineers, designers, and implementors do not currently have. Most software engineers have not been trained in techniques required to develop formal software specifications, and their inexperience in these techniques makes formal specification development appear difficult [23]. Additionally, as most of the programmers who would be implementing the formal specification are themselves not familiar with these techniques, implementation-related benefits of formal specifications are not currently attainable.

3. The very formality which makes formal specifications desirable during the later phases of requirements specification makes them an inappropriate tool for communicating with the end user during the earlier requirements elicitation and confirmation stages. More so than the software engineers, most end users who pro-

[1] **Note:** *Completeness* is used here in the sense that the specification includes all enumerated system components, elements, and options. However, due to the human-intention natures of functional requirements, no specification language, including a formal language, can ensure completeness in the sense that all of the user's functional requirements are included in the specification document.

vide the requirements and approve the requirement specifications are neither familiar nor comfortable with the formal specification languages [6]. This makes it difficult for such end users who are not mathematically trained to understand and approve specifications expressed in a formal language.

4. Preliminary empirical evidence from cognitive science [10] suggests that in the early stages of problem solving, when the problem area is relatively ill structured, the use of formal representations inhibits the exploration of alternatives and is detrimental to the quality of the outcome. Thus, due to the requirement of a formally defined vocabulary, syntax, and semantics, formal specification languages may not be an ideal tool for exploring and discovering the problem structure during the problem refinement process.

5. Sommerville suggests that management is generally conservative and unwilling to use new techniques whose benefits are not yet established [23]. The payoff of the upfront investment in developing formal software is not immediate and difficult to quantify. However, Sommerville goes on subsequently to indicate that "with formal specification, specification and implementation costs are comparable [to when a conventional process is used] and system validation costs are significantly reduced." Hall discusses evidence that is beginning to accumulate on the costs of projects using formal specification methods. Hall reports that "none of this evidence supports the idea that development costs are higher if you use formal specification; if anything, it suggests they are *lower*" [12, p. 17].

Given these difficulties in using formal methods, challenges remain in integrating formal methods with the system development effort and in scaling up formal method techniques to large-scale real-world development projects [18, 23, 24]. Unless viable strategies for incorporating formal methods in the software development process are developed, it may be difficult to attain the promise of formal methods in real-world software development projects.

A Framework for Classifying Strategies

We now develop a framework for classifying and contrasting strategies for using formal specifications in software development. This framework is developed using an organizing methodology called morphological analysis. Morphological analysis is a technique for building structures (morphologies) of existing information in a subject area. With this taxonomy, formal specification methods can be classified, differences and similarities among methods recognized, and future research topics identified.

As the first step in conducting a morphological analysis is to identify currently known dimensions and parameters in a subject area, our analysis begins with a review of relevant research describing the strategies for developing formal specifications. (The Appendix gives a brief overview of the research which provided the basis for the morphological analysis. Table 1 summarizes this research.) This research represents a cross-section of currently available strategies. Each article in Table 1 proposes strategies for using formal specifications in the software development process. A set of dimensions for classifying strategies are abstracted from an analysis of these strategies.

The Formalization Process Dimension

First, the strategies can be seen to differ according to the process by which formalization of specifications is achieved. Some strategies propose moving directly from high-level, informal (i.e., natural language) specifications to a fully developed set of formal specifications. Thus the representation and specification activity is in the formal domain. For example, the "Integrated" approach of Kemmerer, Kemmerer *(c)* in Table 1, proposes that the high-level formal specification of the system be derived directly from a precise English statement of critical requirements [15]. Jones uses a similar process when he suggests that the proof obligations of VDM decomposition rules can stimulate design steps [14]. Miriyala and Harandi propose a strategy to derive formal specifications from limited natural language specifications directly [21]. Wing also suggests an interaction process between the customer and the specifier to produce formal specifications directly [24].

In contrast to this "direct formalization process," other strategies use intermediate representations of software specifications to help move from initial natural language to formal specifications. In such a strategy, one or more semiformal specifications provide mediating increments of formality between the informal natural language specification and the formal specifications. Andrews and Gibbins use an abbreviated bottom-level Structured Analysis structure chart to guide the operational decomposition of a high-level VDM specification [1]. The "After-the-Fact" and "Parallel" approaches of Kemmerer, Kemmerer *(a)* and *(b)*, respectively, in Table 1, involve the use of standard development methods as intermediate steps in deriving formal specifications [15]. Babin, Lustman, and Shoval use an extended Structured Analysis approach to transform control flows into finite-state machine representations by applying a set of rules [2]. Conger *et al.* produce formal specifications by using Structured Analysis representations and tracking the hierarchical partitioning of high-level data-flow transforms with VDM specifications [4]. Fraser, Kumar, and Vaishnavi also generate formal specifications from Structured Analysis specifications [9]. Kung combines Structured Analysis and Entity Relationship models in sequence with a number of formal modeling methods [16].

Thus, a strategy for using formal methods can be located in a *formalization process* dimension according to whether or not the strategy transitions through intermediate models of requirements as intermediate steps in producing formal specifications and designs. A strategy which begins with an informal model and transitions through intermediate, more formal models to arrive at a formal specification uses a *transitional formalization process*, while one which moves from an informal representation to a formal specification without going through transitional steps uses a *direct*

TABLE 1. A cross-section of published strategies for using formal methods

A Sketch of Formal Specification

Formal Methods

"Formal methods are mathematically based techniques for describing system properties" [24]. A formal method has "an underlying theoretical model against which a description can be verified" [5].

Formal Specification Languages

"To achieve precision, a specification must be written in a language which has a formal basis" [14]. "A *formal specification language* provides a formal method's mathematical basis. . . . A *formal specification language* provides a notation (its syntactic domain), a universe of objects (its semantic domain), and a precise rule defining which objects satisfy each specification" [24].

Some Classifications of Formal Specification Methods

Classification	Discussion (*references*)
Model Oriented vs. Algebraic	Modeling the system using mathematical entities (e.g., sets) *vs.* specifying an object class in terms of relationships between the operations defined on that class [23].
Sequential vs. Concurrent	Methods used to specify sequential systems (e.g., *VDM, Z, Larch,* and *FSMs*) *vs.* methods used primarily to specify concurrent and distributed systems (e.g., *Petri Nets, Temporal Logic, Transition Axioms, CSP, CCS,* and *PAISLey*) [22, 24].

Some Examples of Formal Specification Methods

Language	Description (*references*)
Meta-IV (VDM meta-language)	*VDM* supports a *model-oriented, sequential* specification and includes built-in data types (e.g., sets, lists, and mappings) which can be used to define other data types. *Meta-IV* is used to specify, independent of implementation, the software architecture [14].
Z [Zed]	*Z* is a *model oriented, sequential* language based on typed set theory because sets are mathematical entities whose semantics are formally defined. *Z* allows specifications to be integrated (reused) and has graphical highlighting (human interface presentation).
Larch	*Larch* is an *algebraic, sequential* language in which there are no built-in types; a mnemonic notation rather than specialized symbols is used. The user need not learn special vocabulary or mathematical notation and introduces (typically many) relevant symbols with defining equations. Specifications can be expressed with a standard keyboard [11].
Finite-State Machines (FSM)	*FSM* is a *model-oriented, sequential* method for functional specification consisting of a set of states, a set of inputs, a transition function which specifies the next state given the current state and input, an output function which specifies actions associated with transitions, the initial state, and the final state.
Petri Nets	*Petri Nets* provide a *model-oriented, concurrent* method for specifying timing and parallelism. A *Petri net* is typically represented as a bipartite digraph whose nodes represent places (marked by tokens) and transitions (which are enabled by a firing rule to change markings).
Temporal Logic (TL)	*TL* is a *model-oriented, concurrent* modal logic language that supports statements about properties of states without requiring arguments which specify the time at which a statement is true. Program behavior is specified by interpreting formulas involving temporal operators (e.g., "next") over structures of states, the computations executed by a program, which gives a programming language temporal semantics [19].
Transition Axioms (TA)	This *model-oriented, concurrent* method combines an axiomatic method to describe the behavior of operations with temporal logic assertions which specify safety and liveness [17].
Communicating Sequential Processes (CSP)	*CSP* is a *model-oriented, concurrent* language in which information is transmitted from one process to another through a communication line (channel). Two processes rendezvous if one is prepared to send information over the channel and simultaneously the other is prepared to receive from the channel. Rendezvous achieves communication (e.g., of a value of a variable) and provides synchronization during message transmission [13].
Calculus of Communicating Systems (CCS)	*CCS* is an *algebraic, concurrent* alternative to *TL, TA,* and *CSP.* The notation describes concurrent functions and arguments competing and choosing interactions. Denotational semantics are provided by a communication tree of evaluation paths of an expression [20].
PAISLey	A *model-oriented, concurrent* executable specification language which provides for internal consistency checking via a simulation of the software system run by an interpreter [25].

TABLE 1. A cross-section of published strategies for using formal methods

A Sketch of Formal Specification	
Authors	**Strategy for Using Formal Methods**
D. Andrews and P. Gibbins [1]	Develop formal specifications using an informal Structured Analysis structure chart to guide operation decomposition.
G. Babin, F. Lustman, and P. Shoval [2]	Beginning with an extension of Structured Analysis, Finite-State Machine (FSM) representations are used to formally describe control flows; transformation of control flows into FSM representations is achieved using a set of rules; a decomposition algorithm is used to refine the FSM specification into a hierarchical set of FSMs.
S. A. Conger et al. [4]	Formal specifications are developed by tracking, at each level, the hierarchical partitioning of high-level Structured Analysis data-flow transforms with VDM specifications; data flows are represented in an abstract syntax, and a VDM specification is given for each transform in the DFD set; VDM specifications are combined according to the architecture provided by the leveled DFD set.
M. D. Fraser, K. Kumar, and V. K. Vaishnavi [9]	Control processes in the Structured Analysis specifications are identified as while-structures through analyst interaction; bottom-level processes are described with decision tables which are mapped into VDM specifications using a conversion rule; the resulting VDM specifications are composed bottom-up using sequence and while-structure composition rules based on the precedence analysis of the DFD.
C. B. Jones [14]	Abstract models of data types are chosen using natural language problem representations; proof obligations of VDM decomposition rules can be used to stimulate and guide intuition about program design (i.e., operation decomposition) steps; decomposition proof obligations provide a framework into which design commitments can be recorded.
R. A. Kemmerer [15]	(*a*) "After-the-Fact": after the system is built with a standard method, a formal specification is derived, and formal verification of the implementation may be done. (*b*) "Parallel": formal specification and verification are carried out in parallel with a standard development method. (*c*) "Integrated": the formal method is completely integrated into the development cycle; critical requirements, usually an English statement of what is desired, are stated in precise mathematical terms; a high-level formal specification is then produced which gives a precise mathematical description of the system's behavior without many implementation details; less abstract specifications implement the next higher-level specifications, with increasing detail, until the system can be coded in a high-level language and the implementation is shown to satisfy the original critical requirements.
C. H. Kung [16]	A number of modeling techniques (DFDs, Petri nets, Entity-Relationship models, and relational calculus) are combined in sequence to develop a conceptual model of the system.
K. Miriyala and M.T. Harandi [21]	A user interacts with a knowledge-based assistant which derives formal specifications from informal descriptions expressed in a restricted subset of natural language; the assistant makes all decisions, prompts the user for information needed, and operates by schemas, analogy, and difference-based reasoning.
J. M. Wing [24]	A mapping between informal and formal requirements specifications can be achieved iteratively through interaction between the customer and the specifier.

formalization process.

Locating a strategy in the formalization process dimension can be made more precise by defining degrees of formalism. This categorization is defined in the upper panel of Table 2. The definitions are adapted from two sources. Dart *et al.* propose definitions of *informal, semiformal,* and *formal* specification and design methods [5]. Similarly, in discussing the suitability of techniques for metamodeling, Brinkkemper defines *informal, structured,* and *formal* techniques [3]. Both Dart *et al.* and Brinkkemper use their definitions to classify and organize discussions of a variety of specification and design methods along the formalization process dimension. Schach cites the work of Dart *et al.* and uses their definitions to categorize and differentiate specification methods [22]. The lower panel of Table 2 gives examples of methods categorized by their degree of formalism.

Table 3 gives the definition of the *formalization process* dimension. *Direct* process strategies are characterized by the absence of any intermediate use of semiformal representations. In the case of *transitional* process strategies, the analyst produces and uses semiformal representations as intermediate steps which aid in producing a formal specification and design. The degree of completeness and preciseness of the semiformal model varies and may focus on key aspects (constructs and rules) of the semiformal representation that aid in the derivation of formal specifications and designs; Andrews and Gibbins' use of an informal structure chart is an example [1].

An inspection of the strategies pro-

posed by Kemmerer (Table 1) suggests a further refinement of this definition. The transitional process strategy can be partitioned into *sequential* and *parallel successive-refinement* approaches. In a *sequential* approach, the complete semiformal models are produced first, from which the formal specifications follow. (See [1, 2, 9], approach (a) of [15, 16].) Other approaches suggest that semiformal and formal representations may be produced in parallel through successive refinements ([4] and approach (b) of [15]).

Refining the transitional strategy into the sequential and parallel successive refinement approaches completes the definition of the formalization process (Table 3).

The Formalization Support Dimension

The strategies in Table 1 can be seen also to differ according to the computer support that a strategy uses for producing formal specifications. On one hand the process of producing formal specifications may be unassisted, relying only on the innate problem-solving capabilities of the requirements engineer. On the other hand the strategy can use computer-based assistance.

For example, the strategies of [1, 14], all three approaches of [15, 16, 24] rely only on human problem-solving capability to produce the formal specifications. In contrast, the strategy proposed by Miriyala and Harandi uses an automated knowledge-based assistant to derive formal models [21]. Such a strategy uses heuristics and human knowledge, usually domain specific, for guiding the formalization process. Finally, [2] and [9] propose strategies which provide an algorithmic approach to deriving formal specifications. These strategies use transformational, computer-executable procedures for formalizing a specification. In either case, given that we are starting from informal or semiformal specifications which may be ambiguous or incomplete, the computer support needs to be interactive.

Thus, a strategy for using formal methods can be located in a second dimension, the *formalization support* dimension, according to the computer-based assistance that the strategy uses in producing formal specifications (Table 4).

The cross-product of the two dimensions, formalization process and formalization support, thus provides our framework for strategies for incorporating formal specifications in software development (Table 5). This framework identifies four generic strategies: direct unassisted, direct computer-assisted, transitional unassisted, and transitional computer-assisted. The two transitional generic strategies are further subdivided into transitional-sequential and transitional-parallel unassisted and computer-assisted strategies, respectively. The framework is populated by mapping the 11 strategies used in the morphological analysis (Table 1) into the framework.

An Exercise in Validating the Framework

Table 5 shows 13 strategies cross-classified into the framework. Eleven of these are the strategies used originally in the morphological analysis (Table 1) which lead to the definition of the framework, and two are strategies not discussed before. The two

TABLE 2. Definitions of specifications and design methods with examples

Categories of Formalism[1]		
Informal	**Semiformal**	**Formal**
These techniques do not have complete sets of rules to constrain the models that can be created. Natural language (written text) and unstructured pictures are typical instances.	*These techniques have defined syntax. Typical instances are diagrammatic techniques with precise rules that specify conditions under which constructs are allowed and textual and graphical descriptions with limited checking facilities.*	*These techniques have rigorously defined syntax and semantics. There is an underlying theoretical model against which a description expressed in a mathematical notation can be verified. Specification languages based on predicate logic are typical instances.*
Examples of Categorizing Methods		
Natural Language Specifications *(B, D, S)*[2] *(Although often used, published studies are less numerous; e.g., Schach [22], Sommerville [23])*	Variations on Data/Control Flow Diagrams *(B, D)* Entity-Relationship Diagrams *(D)* DeMarco *(S)* Gane and Sarson *(S)* PSL/PSA *(D, S)* SADT *(D, S)* SERM *(D, S)* IORL *(D)* CORE *(D)* SDL *(D)* JSD[3]	Petri Nets *(B, D, S)* State Machines *(D, S)* Executable Specifications *(S)* GIST *(D)* Refine *(D)* VDM *(B, D, S)* Anna *(D)* Z [Zed] *(B, D)* CSP *(S)* GIST *(D)* Predicate-Transition-Nets *(B)*

Notes: [1]adapted from Dart *et al.* [5] and Brinkkemper [3]
[2]Brinkkemper *(B)*, Dart *et al. (D)*, Schach *(S)*
[3]JSD is not classified by *B, D,* or *S.* Wing [24] includes JSD among semiformal and informal methods. We would further classify JSD as semiformal.

recent strategies reported in Fields and Elvang-Gøransson [7] and France [8] were appraised using the framework. These strategies were classifiable on each of the two dimensions of the framework. This small exercise in validating the framework is a step toward demonstrating the robustness of our framework.

Fields and Elvang-Gøransson propose that "the development of a small safety critical system, like most computer systems begins . . . as an informal description written in a natural language. Using this as a guide, a formal specification is written . . . , which, it is intended, captures the essence of the informal description" [7]. Although they propose an ingenious, useful tool, *mural*, for working with VDM, the computer support proposed takes over some of the labor-intensive aspects of entering a formal specification into a usable theorem prover. That is, the high-level formal specification is generated first, using the informal description as a guide, and then refined with the help of the theorem prover. Thus, the strategy proposed is to generate directly the first, high-level formal specification from an informal, natural language specification without automated tool assistance, placing the strategy in the direct unassisted cell of the proposed framework.

On the other hand, France develops an extended Data Flow Diagram (DFD) formal specification which generates a formal specification with a "technique for associating semantics with control-extended DFD's. The specification characterizing the semantics of a C-DFD can be viewed as formal design specification of the application modeled by the C-DFD . . . We are currently working on developing a specification-development tool based on DFD's, consisting of an informal 'front-end' supported by a formal 'back-end.' The front-end supports the creative development of DFD specifications . . . , while the back-end supports the generation of formal specifications from DFD's and the rigorous investigation of semantic properties" [8]. The characterization of the technique as one which associates (and hence as one which is algorithmic) semantics with control-extended DFD's to produce a formal specification places this strategy in the transitional-sequential computer-assisted cell of the framework.

Discussion and Assessment of Strategies

This framework can be useful for describing, understanding, and comparing strategies for incorporating formal specifications in software development. As an illustration of this usefulness, we use the framework to assess the generic strategies generated by combining the process and support dimensions of the framework. Such assessments can, in turn, be a guide to evaluating the applicability of the strategies proposed in the literature to specific development contexts.

Note, however, that ultimately the assessment of these generic strategies should be based on large amounts of data collected from a number of projects that are representative of application domains of interest. Judging from the limited number of published case studies, it is apparently still too early to develop a significant base of empirical data about strategies that have been used a substantial number of times with projects representative of the general categories of software development.

Worse, most published accounts of the use of formal methods do not describe explicitly the strategies followed to produce formal specifications. For example, [12] describes a variety of projects that use formal specification methods, but does not include an explicit description of the strategies used to produce these specifications. Such omissions cannot be taken to mean that no strategies were followed. Rather, the issues of the formalization strategy used and its resource requirements might have been simply considered as details in comparison with larger issues such as describing the specification language and providing fragments of formal specifications to illustrate the formal method in action. In any case, such omissions make it all the more difficult to assess the reported experiences with formal methods with regards to the strategies used to produce the formal specifications.

The following discussion relies mainly on broad deductive reasoning to provide at least preliminary evaluations of the generic strategies suggested by the framework.

Direct Unassisted Strategy. Refer to the upper left corner of the two-dimensional framework (Table 5). The generic strategy defined by this cell—the **direct unassisted** strategy—is characterized by a direct formalization process that relies entirely on the problem elicitation, structuring, and requirements specification skills of the requirements engineer unassisted by any computer-based support. For the strategy to be successful, it requires first that the requirements engineer has a thorough knowledge of the application domain, and second, that she or he can grasp and formalize the whole of the problem in its entirety. Furthermore, as the requirements elicitation is primarily in an informal (natural) language, whereas the specification is produced in a formal language, the requirements specification and validation process requires close collaboration between the user and the requirements engineer.

Taken together, these considerations point to small, well-structured problems with which the analyst is completely at ease, and to users who are either mathematically sophisticated enough to understand and validate the formally stated specifications or are willing to rely upon informal restatements of the formal specifications for validation. This general inference is not contradicted by the case studies in [7] and [15]. (Of the four strategies classified in this cell, only these two strategies were published with accompanying case studies.) Fields and Elvang-Gøransson illustrate the use of their strategy by specifying a "small and easy to understand" safety-critical reactor watchdog system [7]. Similarly, Kemmerer illustrates the use of his "integrated" (direct) strategy by specifying a small, well-defined secure release terminal [15].

Another possible use of this strategy would be in prototypical situations where the focus is on the "proof-of-concept" (as in the early

days of aviation when "a good landing was one you walked away from"), rather than on the usual project accountability considerations of industrial-strength projects (e.g., traceability, project cost, manpower availability, and resource efficiency). Such situations would be characterized by the use of highly trained and motivated research or leading-edge professional personnel (who may be in short supply) and relatively less restrictive resource constraints. Again, the projects described in Fields and Elvang-Gøransson and Kemmerer are examples of such prototypical situations.

However, the transition from small, well-structured, or prototypical projects endemic to this cell, to industrial-commercial projects introduces additional considerations. First, the large size and complexity of typical industrial projects may make it difficult for the requirements engineer to elicit and structure the informal (and in some cases as yet to be discovered and therefore unexpressed) requirements into formal specifications directly [1, 4, 14]. Second, in the semi- to ill-structured situations encountered in real-world applications, the problem structure may not be evident, and the analyst may need intermediate representations to help her or him discover the underlying problem structure [10]. Third, as typical users (especially in other than engineering and scientific domains) are likely to be unfamiliar with the mathematical notation of formal specification languages, requirements elicitation and verification of such large and complex applications using formal specification models may not be practical. Finally, the small likelihood of finding requirements engineers who are both comfortable with the application domain and are trained in formal methods, coupled with the large personnel requirements in typical industrial projects, makes it impractical to use formal methods in these projects in a direct unassisted manner. Taken together, the above considerations constitute a *scalability problem.*

Direct Computer-Assisted Strategy. A direct computer-assisted strategy re-

TABLE 3. Definition of formalization process

Formalization Process

= ***Direct*** moving from informal to formal specifications without the use of (semiformal) intermediate specifications.

= ***Transitional*** moving from informal to formal specifications through the use of (semiformal) intermediate representations

Transitional Approaches

= ***Sequential*** formal specifications are derived from a final set of semiformal models.

= ***Parallel Successive Refinement*** the semiformal and formal specifications are produced through successive refinements of each simultaneously.

TABLE 4. Definition of the support dimension

Formalization Support

= ***Unassisted*** human problem solving.

= ***Computer Assisted*** human problem solver is supported by computer-based methods which use heuristics and knowledge (knowledge-based) or transformational computer-executable procedures (algorithmic).

TABLE 5. Taxonomy for strategies for producing formal specifications

		Formalization Support	
		Unassisted	**Computer Assisted**
Formalization Process	**Direct**	*Jones [14]; Kemmerer (c) [15]; Wing [24]; Fields and Elvang-Gøransson [7]**	*Miriyala and Harandi [21]*
	Transitional — **Sequential**	*Andrews and Gibbins [1]; Kemmerer (a) [15]; Kung [16]*	*Babin, Lustman, and Shoval [2]; Fraser, Kumar, and Vaishnavi [9]; France [8]**
	Transitional — **Parallel Successive Refinement**	*Kemmerer (b) [15]; Conger et al. [4]*	

*These two strategies were not used in the morphological development of the framework, but were added as a "validation exercise" after the framework had been defined.

lies on computer-based support to develop formal specifications directly from informal natural language specifications. Given the somewhat preliminary, vague, and abstract nature of informal specifications, such computer assistance is usually in the form of knowledge-based support for eliciting, discovering, and creating the formal specifications. The domain-dependent nature of knowledge bases, however, could limit the applicability of such assistance. For example, SPECIFIER, a natural-to-formal language "specification-derivation system" developed by Mirayala and Harandi, uses concepts extracted from informal specifications as a guide to derive specifications in a formal specification language similar to the *Larch Shared Language*. It does this by recognizing and instantiating *schemas*, by applying *analogy mapping*, and by performing *difference-based reasoning*. However, given the domain-dependent nature of analogies, and the primitive (syntactic) level of commonly occurring operations for which the schemas are currently available, the scaling up of this strategy for real-world applications would require a large amount of work in amassing a variety of analogies and acquiring and encoding myriad schemas [21, p. 1141]. Given further progress in such support, in the future it may be possible to employ direct-assisted strategies in eliciting and discovering requirements in specific application domains. In the meantime, the use of direct-assisted strategies would be limited to a restricted set of small syntactic-level problems.

Transitional Unassisted Strategies. In a transitional unassisted strategy one or more semiformal specifications provide mediating increments of formality between the informal natural language specifications and the formal specifications. The transitional unassisted strategy, however, relies entirely on the formal language skills of the requirements engineer to translate between semiformal and formal requirement specifications.

The transition through semiformal specifications has a number of advantages over a direct strategy. First, recent research from cognitive science [10] suggests that semiformal representations which mediate in the transition process may be better suited than formal representations to exploring and discovering the problem structure in ill-structured problems. Second, Denning states that "the language used to describe businesses and organizational processes is different from the language used for formal specification" [6]. Denning further suggests that end users and requirement engineers would be in a better position to know if they are in agreement if the language of the specifications is closer to the semiformal language of business. Semiformal specifications, thus, provide a useful bridge between users and requirements engineers. Third, as shown in [1] and [4], semiformal specifications can guide the stepwise refinement of formal specifications. Thus, transitional strategies can provide elicitation, structuring, and validation advantages over direct strategies in the case of large or semistructured applications in problem domains with relatively mathematically unsophisticated users.

Transitional unassisted strategies may be further subdivided into **transitional-sequential** and **transitional-parallel (successive refinement)** unassisted strategies. In the case of **transitional-sequential** unassisted strategies, complete semiformal specifications are produced first, from which formal specifications follow. This approach is feasible in those situations where detailed and complete requirements are either previously known or are easily discovered. Semiformal specifications then become the means for structuring and validating the complete requirements prior to their manual (unassisted) translation to formal specifications. An example of this strategy would be the reengineering (i.e., respecifying) of specifications from existing software (e.g., [12]).

Alternatively, in cases where the problem domain is either ill defined or ill structured, and therefore, the requirements are yet to be discovered, a **transitional-parallel** unassisted strategy would be appropriate. In this strategy synchronized semiformal and formal system representations are produced through parallel concurrent refinements of the specifications. During the refinement process, the partitioning and decomposition heuristics of the semiformal methods guide the stepwise refinement of the semiformal specifications. At each new level, the newly refined semiformal specifications are manually translated into their corresponding formal specifications. The refined formal specifications, in turn, can be formally verified against the higher-level formal specifications, thus ensuring that the refined specifications actually represent the higher-level requirements. Thus, the transitional-parallel strategy has the potential of letting semiformal and formal specifications aid each other in a synergistic fashion during the requirements discovery and refinement process.

In either case, the translation from semiformal to formal specifications is performed by the requirements engineer without computer assistance. The manual (human) translation process would require additional time and resources over and above those required for producing the semiformal specifications. Thus, Kemmerer argues that due to the duplication of effort in producing both semiformal and formal specifications, a direct (integrated) strategy has time and cost advantages over sequential ("after-the-fact") and parallel strategies [15, p. 37]. He observes further that in the case of a parallel strategy, if the semiformal and formal specifications are developed by two parallel requirements specification teams, problems of communication between the two teams and synchronization between the semiformal and formal specifications can arise [15, p. 38].

Transitional Computer-Assisted Strategies. These strategies are the last set of generic strategies suggested by the framework. Like the transitional unassisted strategies, these strategies too are characterized by the use of semiformal specifications to mediate between informal natural language specifications and formal language specifications. Therefore, they have the same advantages over the direct strategies as discussed in the case of transitional unassisted strategies.

However, unlike the transitional unassisted strategies, in the case of transitional computer-assisted strategies, computer assistance would be available to move back and forth between semiformal and formal specifications. Leveson [18] and Sommerville [23] suggest that for formal methods to become practical in real-world projects, computer-assisted tools are needed. The use of computer assistance would provide a number of advantages. First, computer assistance, by supporting the requirements engineer in requirements elicitation and problem-structuring tasks and by replacing human labor for routine translation tasks, would ameliorate the time and cost disadvantages of transitional strategies suggested by Kemmerer. Second, Mirayala and Harandi suggest that "writing formal specifications is a knowledge-intensive and error-prone activity" [21, p. 1126]. The use of computer assistance in the derivation of formal specifications would tend to reduce these errors. Third, computer support in translating between semiformal and formal specifications would eliminate the synchronization problems between parallel semiformal and formal specifications [15] which can arise in a transitional-parallel strategy. Finally, as the majority of elicitation and problem-structuring tasks can now be performed in the semiformal domain, this would reduce the need for expensive highly trained personnel in the formal specifications domain. Thus, it would be possible to staff large industrial-strength formal specification projects adequately.

Finally, like the transitional unassisted strategies, transitional computer-assisted strategies are also further subdivided into **transitional-sequential** and **transitional-parallel (successive refinement)** computer-assisted strategies. In the case of **transitional-sequential** computer-assisted strategies, first a complete set of semiformal specifications are produced. Next, a computer-based tool is used to translate the set of semiformal specifications into their corresponding formal specifications, usually with some interactive help from the analyst. The computer assistance could be in the form of a computer algorithm or a knowledge-based system [2, 9].

Alternatively, in the case of a **transitional-parallel** computer-assisted strategy, synchronized semiformal and formal system representations are produced through parallel concurrent refinements of the specifications. During the refinement process, the partitioning and decomposition heuristics of the semiformal methods guide the stepwise refinement of the semiformal specifications. At each new level, computer-based support can be used to translate the newly refined semiformal specifications into the corresponding formal specifications. The refined formal specifications, in turn, can be formally verified against the higher-level formal specifications thus ensuring that the refined specifications truly represent the higher-level specifications. Any problems discovered in this verification process can be relayed back to the semiformal domain where they can be used to correct the semiformal specifications. Thus, the transitional-parallel strategy has the potential of letting semiformal and formal specifications aid each other in a synergistic fashion during the requirements discovery and refinement process. At this time no examples of computer-assisted transitional-parallel formalization strategies have been reported.

Summary and Conclusions

In this article, we have developed a two-dimensional framework for describing and assessing strategies for incorporating formal specifications in software development. The framework identifies four generic strategies: direct unassisted, direct computer assisted, transitional unassisted, and transitional computer assisted. Each of the two generic transitional strategies are further subdivided into transitional-sequential and transitional-parallel (successive refinement) unassisted and computer-assisted strategies respectively.

An examination of these strategies suggests the direct unassisted strategy is most appropriate for small, well-structured, or prototypical problems with which the analyst is completely at ease and in projects where close collaboration between users and analysts exists. However, due to the scalability problems endemic to the direct unassisted strategy its use in industrial-strength projects is impractical. The direct computer-assisted strategies currently deal with syntactic-level domain-specific problems only. Much work needs to be done before they could become useful in nontrivial industrial-commercial applications.

The two generic transitional strategies, unassisted and computer-assisted, provide elicitation, problem structuring, and validation advantages over direct strategies for large or semistructured applications or applications in problem domains with relatively mathematically unsophisticated users. The unassisted transitional strategies, however, are labor intensive and can be subject to human error in translating between semiformal and formal specifications. On the other hand, in the case of transitional computer-assisted strategies, the use of computer-based assistance maintains the advantages of transitional strategies while ameliorating the manpower requirements and human error disadvantages of the transitional unassisted strategies. Thus, transitional computer-assisted strategies provide most promise in addressing the scalability problem.

Within the transitional computer-assisted strategies, transitional-sequential strategies are useful in situations where detailed and complete requirements are either known up front or are easily discovered (e.g. in software reengineering). Alternatively, in cases where the problem domain is either ill defined or ill structured, and therefore, the requirements are yet to be discovered, a transitional-parallel stepwise refinement strategy would be appropriate.

An examination of strategies proposed in literature (Table 5) shows most of the recent work in developing and using strategies for formal specifications has been primarily in the direct unassisted and transitional-sequential unassisted and computer-assisted strategies. At this time no examples of transitional-parallel computer-assisted formalization strategies have been reported. However,

given the promise of such strategies, work in developing them is needed. The authors of this article are currently working on developing such strategies.

References

1. Andrews, D. and Gibbins, P. *An Introduction To Formal Methods Of Software Development.* The Open University Press, Milton Keynes, UK, 1988, Units 1–4.
2. Babin, G., Lustman, F., and Shoval, P. Specification and design of transactions in information systems: A formal approach. *IEEE Trans. Softw. Eng. 17,* 8 (Aug. 1991), 814–829.
3. Brinkkemper, J.N. *Formalization of Information Systems Modelling.* Katholieke Universiteit te Nijmegen, The Netherlands, doctoral dissertation published by Thesis Publishers, June, 1990.
4. Conger, S.A., Fraser, M.D., Gagliano, R.A., Kumar, K., McLean, E.R., Owen, G.S., and Vaishnavi, V.K. A structured stepwise refinement method for VDM. In *Proceedings of the 8th Annual Conference on Ada Technology.* ANCOST, Inc., Culver City, Calif., 1990, 311–320.
5. Dart, S.A., Ellison, R.J., Feiler, P.H., and Habermann, A.N. Software development environments. *IEEE Comput.* (Nov. 1987), 18–28.
6. Denning, P.J. What is software quality? *Commun. ACM. 35,* 1 (Jan. 1992), 13–15.
7. Fields, B., and Elvang-Gøransson, M. A VDM case study in *mural. IEEE Trans. Softw. Eng. 18,* 4 (Apr. 1992), 279–295.
8. France, R.B. Semantically extended data flow diagrams: A formal specification tool. *IEEE Trans. Softw. Eng. 18,* 4 (Apr. 1992), 329–346.
9. Fraser, M.D., Kumar, K., and Vaishnavi, V.K. Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. Softw. Eng. 17,* 5 (May 1991), 454–466.
10. Goel, V. Ill-structured representations for ill-structured problems. Rep. No. DPS-4, Univ. of Calif., Berkeley, Calif., May, 1991.
11. Guttag, J.V., and Horning, J.J. An introduction to the Larch shared language. In *Proceedings of the 9th IFIP World Computer Congress* (Paris). North-Holland, Amsterdam, 1983, 809–814.
12. Hall, A. Seven myths of formal methods. *IEEE Softw. 7,* 5 (Sept. 1990), 11–19.
13. Hoare, C.A.R. *Communicating Sequential Processes.* Prentice-Hall International, Hemel Hampstead, 1985.
14. Jones, C.B. *Systematic Software Development Using VDM.* 2d ed. Prentice-Hall, Englewood Cliffs, NJ, 1990.
15. Kemmerer, R.A. Integrating formal methods into the development process. *IEEE Softw.* (Sept. 1990), 37–50.
16. Kung, C.H. Conceptual modeling in the context of software development. *IEEE Trans. Softw. Eng. 15* (Oct. 1989), 1176–1187.
17. Lamport, L. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst. 5,* 2 (Apr. 1983), 190–222.
18. Leveson, N.G. Formal methods in software engineering. *IEEE Trans. Softw. Eng. 16,* 9 (Sept. 1990), 929–931.
19. Manna, Z., and Pnueli, A. Verification of concurrent programs, part 1: The temporal framework. Tech. Rep. STAN-CS-81-836, Dept. of Computer Science, Stanford Univ., Stanford, Calif., 1981.
20. Milner, R. *A Calculus of Communicating Systems. Lecture Notes in Computer Science,* vol. 92. Springer-Verlag, New York, 1980.
21. Miriyala, K., and Harandi, M.T. Automatic derivation of formal software specifications from informal descriptions. *IEEE Trans. Softw. Eng. 17,* 10 (Oct. 1991), 1126–1142.
22. Schach, S.R. *Software Engineering.* Richard D. Irwin, Inc., and Aksen Associates, Inc., Homewood, Ill., 1990.
23. Sommerville, I. *Software Engineering.*

APPENDIX: Overviews of the Strategies Used in Morphological Analysis

Andrews and Gibbons [1].

Andrews and Gibbins use Structured Analysis to obtain the "overall architecture" of the system. The strategy is first to obtain a complete architectural representation of the specification as a hierarchical structure chart. The structure chart is then "translated" into VDM specifications by the requirements engineer without any computer assistance. The structure charts are used to identify states of the system that the VDM specifications need to account for and to guide the stepwise refinement ("operation decomposition" and data reification) of VDM specifications.

Babin, Lustman, and Shoval [2].

Babin, Lustman, and Shoval propose a strategy in which requirements are expressed in the ADISSA notation using the ADISSA method, a transaction-oriented refinement and extension of Structured Systems Analysis. The internal architecture of the system is a set of transactions which are activated by events and user requests. The strategy is to develop a complete ADISSA representation, and then use a finite-state machine to represent the flow of control of a transaction. The representation is achieved by applying a set of rules to map the control portion of a transaction to a finite-state machine. Thus, the strategy produces formal specifications by a rule-based transformation of semiformal specifications.

Conger et al. [4].

Conger *et al.* provide a cognitive (computer-unassisted) transformation of Structured Analysis Data Flow Diagrams (DFDs) to VDM specifications. The strategy is first to obtain a top-down hierarchically partitioned data-flow diagram using Structured Analysis heuristics. The DFDs provide a guide for the partitioning and stepwise refinement of VDM specifications. Formal specifications are developed by tracking the hierarchical partitioning of data-flow diagrams with VDM specifications. Data-flows and data stores are represented in the abstract syntax, and a VDM specification is produced for each data transformation process in the DFD set. VDM specification are constructed according to the overall architecture provided by the DFD set.

Fraser, Kumar, and Vaishnavi [9].

The strategy proposed by Fraser, Kumar, and Vaishnavi uses data-flow diagrams and decision tables first to develop a leveled and complete set of Structured Analysis specifications. These Structured Analysis specifications are then translated into VDM specifications using an interactive rule-based algorithmic method. Bottom-

4th ed. Addison-Wesley, Reading, Mass., 1992.
24. Wing, J.M. A specifier's introduction to formal methods. *IEEE Comput. 23*, 9 (Sept. 1990), 8–24.
25. Zave, P. An operational approach to requirements specification for embedded systems. *IEEE Trans. Softw. Eng. 18*, 3, (May 1982), 250–269.

About the Authors:

MARTIN D. FRASER is a professor in the Department of Mathematics and Computer Science, Georgia State University. His current research interests focus on software engineering, simulation, telecommunication networks, neural networks, and distributed resource allocation. **Author's Present Address:** Dept. of Mathematics and Computer Science, Georgia State University, Atlanta, GA 30303-3083, matmdf@lochness.cs.gsu.edu.

KULDEEP KUMAR is currently a visiting professor/scientist at the department of Decision and Information Sciences at Erasmus University in the Netherlands. He is also an associate professor in the College of Business Administration at Georgia State University. His curent research interests include information systems development, automated support for systems development, and the management of information systems. **Author's Present Address:** Dept. of Decision and Information Sciences, Rotterdam School of Management, Erasmus University, The Netherlands, kkumar@fac.fbk.eur.nl.

VIJAY K. VAISHNAVI is a professor in the Department of Computer Information Systems, Georgia State University. His current research interests include software engineering, object-oriented knowledge modeling, management of emerging software technologies, and efficient data and file structures. **Author's Present Address:** Dept. of Computer Information Systems, Georgia State University, P.O. Box 4015, Atlanta, GA 30302-4015, cisvkv@gsusgi2.gsu.edu.

level processes are described with decision tables which are mapped into VDM specifications using a decision table conversion rule. The resulting VDM specifications are composed bottom-up using sequence and while-structure composition rules based on the precedence analysis of the DFD. Iterative control processes (while-do and repeat-until) in the DFD specifications are identified as while-structures by the analyst in examining the DFDs. Thus, the strategy is to produce formal specifications algorithmically from Structured Analysis specifications.

Jones [14].

Jones proposes a "style of formal specification [which] uses (abstract) models of data types and implicit specification by pre- and post-conditions. High-level design decisions normally involve choosing the representation of data . . . Operation decomposition . . . is the process of choosing, and justifying, a sequence of transformations which can be [ultimately] expressed in the implementation language" (p. 180). In several specification examples in Chapters 4, 6, and 7 (e.g., see pages 103, 104, and 148) Jones directly chooses initial models (i.e., the models of data types and invariants) to underlie specifications using natural language expressions of problems. Operation decomposition is guided by "proof obligations to stimulate design steps [decomposition]." Jones cautions, however, not "to expect too much from this idea. Design [decomposition] requires intuition and cannot, in general, be automated. What is offered is a framework into which the designer's commitments can be placed."

Kemmerer [15].

Three strategies are discussed for using formal specification methods. *(a)* The after-the-fact, or sequential, strategy consists of building the system "using a standard approach, and, after it is completed, a formal specification for the system is written." Kemmerer notes that this strategy is costly and generally has been used to "increase assurance of a critical system's reliability." *(b)* The verification-in-parallel strategy calls for "performing the formal specification and verification effort in parallel with the [standard] development." Kemmerer envisions two teams. The development team uses a standard method, and, at the same time, the formal-verification team writes and verifies formal specifications. The two-team approach is both costly and highly dependent on good interteam communication for success. *(c)* The integrated verification strategy is the focus of Kemmerer's article.

Formal methods are completely integrated into the development process. The two teams should be one, and "there should not be two separate processes, but rather a single integrated process where the developers use formal specifications as their design notation." This strategy incurs less time penalty than the after-the-fact approach and a lower cost than either of the two previous approaches. Kemmerer outlines the strategy: "First, you state the critical requirements, which are usually an English statement of what is desired, in precise mathematical terms. . . . Next, you provide a high-level formal specification of the system. This specification gives a precise mathematical description of the system's behavior . . . You may follow this by less-abstract specifications that implement the next higher level specification . . ." That is, the formal specifications are developed directly from an informal description of the system by the requirements engineer.

Kung [16].

Kung proposes a strategy for using formal specification methods based on conceptual modeling which "emphasizes active participation of users in requirements specification." Static aspects of the application domain are modeled in an ER-like language, and dynamic aspects are modeled by the traditional DFD technique. Process interface models "describe the communication or synchronization among the processes [in the dynamic model] and can be regarded as modeling of the external logic of the processes." The strategy is to first obtain static and dynamic models, and then to model the process interfaces to provide a basis for formal checking of such qualitative aspects as consistency.

Miriyala and Harandi [21].

Miriyala and Harandi present an automated tool for deriving formal specifications. The strategy is to provide computer assistance in producing formal specifications in the form of an interactive system that provides intelligent assistance to the requirements analyst. The tool guides the analyst through the derivation of formal specifications from an informal requirements document ex-

pressed in a restricted subset of natural language. The strategy here is to produce formal specifications directly from the informal requirements statement by augmenting the capabilities of the analyst with a computerized tool that gives schema-based (domain-independent knowledge of commonly occurring operations) and analogy-based (past analogous specifications are used in the derivations of new specifications) assistance. In either case, a "structure tree" of the informally stated problem is first developed. The structure tree is a hierarchical organization of information present in the informal specifications and is used by both schema-based and analogy-based approaches to derive the formal specification.

Wing [24].

In addition to providing noteworthy introductory treatments of a number of important formal specification methods, Wing presents a strategy for using formal methods that recognizes that "formal methods are based on mathematics but are not entirely mathematical. Formal methods users must acknowledge two important boundaries between the mathematical world and the real world." These boundaries span the mappings from the informal requirements to a formal specification and from the real world to an abstract model. Wing recognizes that the informal-to-formal mapping is fundamental to the task of producing formal specifications. The user's requirements are mapped from an informal expression into a formal one through an iterative process between specifier and user that is not subject to proof. Wing explains this process as one in which "a specifier might write an initial specification, discuss its implications with the customer, and revise it as a result of the customer's feedback." On the other hand, Wing points out that formal specification languages encode abstractions that must be reified in a computer representation. ◻