Formal Methods and Social Context in Software Development

Goguen, Joseph A.

Formal Methods and Social Context in Software Development*

Joseph A. Goguen
Programming Research Group, Oxford University Computing Lab
Luqi
Naval Postgraduate School, Monterey California

Abstract: Formal methods have not been accepted to the extent for which many computing scientists hoped. This paper explores some reasons for that fact, and proposes some ways to make progress. One major problem has been that formal methods have not taken sufficient account of the social context of computer systems. For example, social context causes a continuous evolution of requirements for large complex systems. This implies that designs, specifications and code must also evolve with requirements, and that traceability is important. We discuss a traceability technique called hyper-requirements. To better understand social context, we discuss ethnomethodology, a branch of sociology, and situated abstract data types, which help bridge the gap between the technical and the social. These attempt to provide a scientific basis for requirements capture. Some case studies are briefly described. We distinguish between small, large and huge grain formal methods, arguing that small grain methods do not scale up. This motivates our discussions of software composition and a new paradigm of "Domain Specific Formal Methods."

1 Introduction

Failures of large software development projects are common today, due to the ever increasing size, complexity and cost of software systems. Although billions are spent each year on software in the US alone, many software systems do not actually satisfy users' needs. Moreover, many systems that are built are never used, and even more are abandoned before completion. Many systems once thought adequate no longer are. To remedy this situation, we recommend a two-fold approach: take better account of the social context of computing; and use formal models as a basis for computer support of software evolution.

Experience shows that many failures of large software projects arise from social, political or cultural factors. Hence it is crucial to take account of the

*The research reported in this paper has been supported in part (for J. Goguen) by British Telecommunications plc, the CEC under ESPRIT-2 BRA Working Group 6071, IS-CORE (Information Systems COrrectness and REusability), and Fujitsu Laboratories Limited, and (for Luqi) by the National Science Foundation and the Army Research Office under grant numbers CCR-9058453 and ARO-145-91.

social context of computer-based systems, in addition to the usual technical factors. Social context appears in requirements, where the properties a system must have in order to succeed are determined [11]. The requirements phase of a large system development project is the most error-prone, and these errors are the most expensive to correct [3, 5], so improvements here will have the greatest economic leverage. Unfortunately, requirements are one of the least developed areas of software engineering. Sections 2.2 and 3.1 discuss ethnomethodology, a promising branch of sociology, and situated abstract data types, a new concept that helps bridge the gap between computer technology and its social context. These attempt to provide a scientific basis for requirements capture. Some case studies are given in Section 3.2.

Taking better account of the social context of computing can also lead to faster and more effective system development. For example, requirements for large complex systems are usually wrong initially, and they evolve continually. This has important implications for methodology: first, any methods used to implement requirements should be flexible, so that they can accommodate the ongoing flood of requirements changes, and second it should be easy to trace design changes back to the requirements that triggered them. Section 5.1 describes a technique for traceability called *hyper-requirements.*

We distinguish between formal methods and formal models, in that *formal methods* try to handle some large class of systems (such as information systems), or even all possible systems, whereas *formal models* guide the construction and use of a single system, or a narrow class. We suggest using mechanically processable formal models for building and integrating tools to produce software faster, cheaper, and more reliably, by increasing automation and decreasing inconsistency. This contrasts with formal methods that call for mathematical rigor throughout the development process, usually through a formal notation with a precise mathematical semantics. Section 4.3 distinguishes between small, large and huge grain formal methods, and explains why small grain methods, which are the most common, fail to scale up. The Domain Specific Formal Methods in Section 4.4 illustrate the use of formal models.

2 Social Context and Requirements

Requirements are properties that a system should have in order to succeed in the environment where it will be used [11]. This refers to the system's context of use, and thus to the social as well as the technical. Much of the information that requirements engineers need is embedded in the social worlds of users and managers, and is extracted through interaction with them. This information is informal and dependent on its social context for interpretation. Moreover, much information needed for requirements is *tacit*, i.e., cannot be verbalized by the members who have that information. On the other hand, the representations that appear in constructing computer-based systems are defined by formal rules. Both the formal, context insensitive, and the informal, socially situated aspects of information are crucial to the success of requirements engineering; these two aspects are called "the dry" and "the wet" in [11], which says that the essence of requirements engineering is to reconcile them.

2.1 Video-Based Requirements Elicitation

The Video-Based Requirements Elicitation project the Centre for Requirements and Foundations at Oxford University is exploring techniques from sociology to reveal tacit, interactional work practices that are invisible to standard requirements methods. The following are some goals of this project:

1. To develop an effective new requirements method that can be used by ordinary computer scientists in actual projects.
2. To reduce the risk of delivering inappropriate systems by discovering what work practices must actually be supported.
3. To ease the introduction of new systems by understanding where disruptions might and might not be tolerable.
4. To help manage user expectations by determining where users might want a new system to give a better service than the old one, through analysis of current work practices.

In this project, audio-visual recordings of actual work are analyzed using principles from ethnomethodology, to better understand social and interactional practices in the workplace.

2.2 Ethnomethodology

Traditional sociology is much influenced by what it considers to be orthodox science, where the scientist first formulates a theory, on the basis of which predictions are made, and then tested empirically. The aim is to achieve *objectivity*, in the sense that the desires and biases of the scientist cannot affect the conclusions. Hence, there is a rigid separation between subject and object, between observer and observed. Since modern physics has already moved far from this kind of objectivity, it should not be surprising if sociology, and the social aspects of computing, had to go even further. In particular, if objective information is replaced by situated information, then orthodox techniques for formulating and testing hypotheses, e.g., statistical sampling, are not valid, because the events observed can no longer be assumed statistically independent. However, statistical methods are the foundation for much sociology, e.g., the design and evaluation of questionnaires. This is not to say that statistics and questionnaires are never useful, but that they are *not always valid*, and in particular, that they should not be used where context plays a significant role.

Ethnomethodology can be seen as a reaction against the "scientific" approach of traditional sociology. Ethnomethodology reconciles a radical empiricism with the situatedness of social data, by looking closely at how competent members of a group actually organize their behavior. A basic principle underlying ethnomethodology is that members are held *accountable* for certain actions by their social groups; moreover, exactly those actions are considered socially significant by those groups. A member performing such an action can always to be asked for an account, that is, a justification[1]. Let us call this the *principle of accountability*. From this follows the *principle of orderliness*, that social interaction is *orderly*, in the sense that it can be understood. This follows from the fact that the participants themselves understand it, because of accountability; therefore analysts should also be able to understand it, if they can discover the methods and categories that members themselves use to make sense of their interactions. This implies it is important to use "naturally occurring" data, collected in a situation where members are engaged in activities that they regularly and ordinarily do; otherwise, the basic principle of accountability will not apply, and we cannot be sure that events in the data have any natural social significance. For example, data collected from interviews cannot be used.

Ethnomethodology tries to determine the *categories* and *methods* that members use to render their actions intelligible to one another; this contrasts with presupposing that the categories and methods of the analyst are necessarily superior to those of members. The methods and categories of members are identifiable through the ways that members are held socially accountable by other members of their group. Through immersion in data from some particular social group (such as stock brokers), particular competencies are gradually acquired that enable an analyst to be a sensitive, effective "measuring instrument" in that domain. In this way, subjectivity is harnessed rather than rejected.

Unfortunately, ethnomethodology can be hard to understand; relatively comprehensible expositions of some important points are in [16], [24], and [11], which we have followed here. Conversation analysis studies details of timing, overlap, response, interruption, repair, etc. in ordinary conversation [21], while interaction analysis uses video data.

We can now be more precise about what it means to say that social interaction is *situated*: it means that the events in some interaction can only be fully understood in relation to the concrete situation in which they actually occur. The following *qualities of situatedness* (from [11], inspired in part by Suchman [24]) may help to further clarify this point:

1. *Emergent:* Social events cannot be understood at the level of the individual, that is, in terms of individual (cognitive) psychology, because they are jointly constructed as social events by the members of some group through their on-going interactions.

2. *Local:* Actions and their interpretations are constructed in some particular context, including a particular time and place.

3. *Contingent:* The construction and interpretation of events depends upon the current situation (potentially including the current interpretation of

[1]This does not mean that such accounts are always, or even usually, requested by members of the group, or that they are necessarily given when requested.

prior events). In particular, interpretations are subject to *negotiation*, and relevant rules are interpreted locally, and can even be modified locally.

4. *Embodied:* Actions are linked to bodies that have particular physical contexts, and to the particular way that bodies are embedded in a context may be essential to the social interpretation of some events.
5. *Open:* Theories of social events cannot in general be given a final and complete form, but must remain open to revision in the light of further analyses and further events.
6. *Vague:* Practical information is only elaborated to the degree that it is useful to do so; the rest is left grounded in tacit knowledge.

We will see that these qualities give rise to basic limitations of formalization.

2.3 An Hypothesis and Some Consequences

The *retrospective hypothesis* [11] says that it only becomes clear what the requirements really are when the system is successfully operating in its social and organizational context. This explains why it can be so difficult to manage the requirements of a large system. The retrospective hypothesis also explains why it can be so difficult to enforce rigid process models on actual software projects: it is difficult even to know what phase a given action fits into until some coherence has emerged retrospectively. Note that it takes work by members to achieve a retrospective reconstruction, and that this work is often not done in real projects because of the effort required.

We can now understand why it is impossible to completely formalize requirements: it is because they cannot be fully separated from their social context. More specifically, the qualities of situatedness explain why the lifecycle phases cannot be fully formalized or separated. Indeed, the activities that are necessary for a successful system development project cannot be expected always to fit in a natural way into any system of pre-given categories, and practising software engineers often report that they have to spend much of their time circumventing narrowly prescriptive plans and rules [4]. In general, abstract representations have only a practical utility, and must be interpreted concretely in order for that utility to be made manifest [24]; this includes software production plans and process models.

These considerations have consequences for software engineering. Perhaps the most important is that tools must provide very strong support for retrospective revision; in particular, they must be very flexible, to accommodate the frequent changes in requirements and their links with other objects. Another consequence is that degrees of formalization are needed, ranging from raw data to mathematical formulae. Moreover, information that is heavily situated should come with pointers into its context (e.g., background ethnographic information, audio and video clips of work and interviews, questionnaires and their analyses, sample documents from the work environment, etc.), in order to make it understandable by those who have not had direct contact with the client group.

3 Requirements Elicitation

This section discusses a new approach (from [11]) for transferring information from requirements analysts to system developers, using concepts from both sociology and computing science. Two case studies are briefly described.

3.1 Situated Abstract Data Types

In ordinary social interaction, including cooperative work, there are many structures that participants use and represent in a variety of ways, e.g., with verbal descriptions, drawings, tables, graphs, etc. For example, consider sporting events. Figure 1 shows a table from a newspaper representing the order and participants in a boat race, the Henley Regatta, while Figure 2 shows the same information in the form of a tree. This structure could also be conveyed by a table on a scoreboard, or a sequence of phrases in spoken English. Thus there is a precise structure that is independent of how it happens to be represented; i.e., we have an ***abstract data type***, abbreviated *ADT*. As in [11], we use order sorted initial algebra semantics (see [12]) to formalize this structure. A complete formal specification of the Henley Regatta ADT is given in the appendix, using the specification language OBJ3 [14]. This specification has been executed. (Experience shows it is necessary to test all but the most trivial specifications in order to eliminate bugs.) The Henley Regatta example was inspired by Toulmin [25], although Toulmin only used concrete representations without realizing they were algebras, that different representations give isomorphic algebras, or that there is no unique best representation.

Visitors' Cup. Heat 1: Jesus, Cambridge *v.* Christ Church; Heat 2: Oriel *v.* New College; ... Heat 8: Lady Margaret *v.* winner of Heat 1; ... Heat 26: Winner of Heat 23 *v.* winner of Heat 24; Final: Winner of Heat 25 *v.* winner of Heat 26.

Figure 1: A Draw for the Henley Regatta

Several different kinds of entity are involved in a regatta. Some of these can be arranged in a hierarchical classification scheme according to the subsort relation. Sorts correspond to an important class of members' categories, although not every members' category is formalized by a sort; for example, heats are not formalized this way. We will say that boats have sort `Boat`, completed regattas have sort `Reg`, possibly not yet specified boats have sort `Boat?` (a supersort of `Boat`), and possibly not yet completed regattas have the supersort `Reg?`; the latter includes all draws. It is convenient to assume that `Boat` is a subsort of `Reg` and that `Boat?` is a subsort of `Reg?`; these assumptions imply that there are trivial regattas consisting of just one boat, which could even be the unknown boat, denoted "?". Some sorts are *built in*, in the sense that they are already

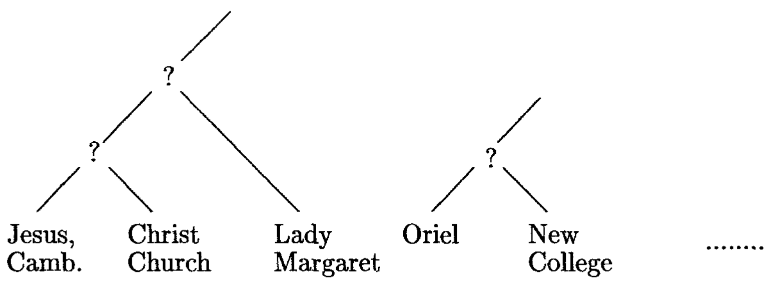

Figure 2: Tree for the Henley Regatta Draw

defined. Two examples are integers and identifiers. The latter have sort `Id` from the built in module `QID` that provides identifiers, here used for naming boats by letting `Id` be a subsort of `Boat`.

Since members use a variety of representations, we should ask how to avoid being tied to any particular representation. The key is to focus on the *methods* that members use to describe (or construct) representations. We distinguish two kinds of method: *constructors*, for building representations from more primitive parts, and *selectors*, for extracting particular information[2]. For regattas, the most important method is a constructor that adds a new heat; it must specify the two contestants, and also provide a slot for the winner. In Figure 2, each non-tip node represents a heat, where the two contestants are the winners of subregattas, or else are given boats; the query mark represents a not yet determined winner. In general, if `R` and `R'` are regattas and `B` is a boat, then `heat(R,R',B)` constructs a new regatta, by adding a heat in which the winners of the subreggatas `R` and `R'` race against each other, with `B` the winner of that heat. Selectors correspond to certain categories used by members. For example, there is a selector that extracts the winner of a given regatta by taking the winner of its final heat.

Methods respect the sorts of representations. For example, the `heat` method takes three inputs, two of which are regattas, and one of which is a boat; it is not meaningful to give a regatta, or an integer, for its third input. In addition, there can be "constants" that do not have any inputs, but do have an output sort; for example, the unknown boat "?" has output sort `Boat?`. We think of these as methods with no input. For example, the method that adds a new heat to a regatta is a function

```
heat : Reg? Reg? Boat? -> Reg?
```

which takes two regatta representations and a boat, and constructs the new regatta where the winners of the subregattas race against each other. Similarly, the selector that gives the winner of a regatta is a function

```
winner : Reg? -> Boat?
```

from regattas to boats.

[2]In object oriented programming, "methods" are operations that can modify; this use of "method" is roughly consistent with that of ethnomethodology. Also, "attributes" are operations that extract information; here these are called "selectors."

Given a particular representation, say by trees, we can collect the possible structures of that representation into an *algebra*, where each sort corresponds to the set of representations of that sort, and the methods correspond to functions that map representations to other representations (or else to built in values, such as numbers). If A is a given algebra, then A_s is its set of representations of sort s. If s' is a subsort of s, then $A_{s'}$ is a subset of A_s (see [12]). Constant methods (such as "?") designate particular representations in an algebra.

It is convenient to overload a method for not yet completed structures with another for completed structures. For example, in addition to `heat` as defined above, we may also have

```
heat : Reg Reg Boat -> Reg
```

for constructing completed regattas.

A set of sorts (with their subsort relations) and a set of methods are together called a *signature* Given a signature, we construct its terms as follows: all constants are terms; and if m is a method with input sorts $s_1, ..., s_n$ and $t_1, ..., t_n$ are terms such that t_i has output sort s_i, then $m(t_1, ..., t_n)$ is a term, with the same output sort as m. For example,

```
heat(heat('JesusCam, 'ChristCh, ?), 'LadyM, ?)
```

is a term, using OBJ3-style identifiers as constants for the names of boats; these begin with a quote and contain no spaces (also we have further abbreviated the boat names).

Given any term t and algebra A over the same signature, that term denotes a unique representation in A. This value or denotation of t is determined by finding the values in A of any constants in t, then applying the functions in A that correspond to the methods in t to those values, then applying further functions to further values, etc., until a final value is obtained. For example, the above term denotes the left subtree in the representation shown in Figure 2. More technically, the denotation of terms is A is given by the *unique homomorphism* from the term algebra to A.

Now we can say that two representation systems are "essentially the same," in the sense that they can represent exactly the same things, if and only if the two algebras are *isomorphic*.

But we are still dealing with representations. How can we obtain structures that are truly independent of how they are represented? The answer has two steps. The first is to describe the *equations* that necessarily hold among the given methods. For example, the fact that the winner of any regatta of the form `heat(R,R',B)` is `B` is expressed by the equation

```
winner(heat(R,R',B)) = B .
```

This is a relationship between the category `winner` and the constructor `heat`.

The second step has to do with limiting the possible models of these equations. So far, we have described an ADT by giving a set of sorts with subsort relations, a set of methods, and a set of equations; let us call these three together a *specification*. Then a *model* of such a specification is an algebra, providing sets for sorts (with subsets for subsorts) and functions for methods, such that all the equations are satisfied. The elements of such a model are a system of representations for the categories and methods in the signature. We now have a way of specifying representations that is truly abstract, in that it says nothing at all about the representations themselves. But unfortunately, there are too many models, and they are not isomorphic to each other. We need one more principle to get what we want; it is called *initiality*, and amounts to the following:

1. *No junk*: every representation in the algebra can be constructed using methods in the given signature.
2. *No confusion*: two terms denote the same representation if and only if they can be proved equal using the given equations.

So now we take as models only those algebras that satisfy not only the equations, but also the two principles above. Such models are called *initial models*, and it can be shown that any two initial algebras are necessarily isomorphic. This gives us the representation independent way of specifying structures that we wanted.

We should distinguish between the object and meta levels of description of this example; a different language is used at each level, for a different purpose. The object level language involves boats, heats, and so on, and its terms construct draws, announce winners, etc. The meta language involves sorts, methods, equations, etc.; at this level we can add new methods, revise equations, etc.

The complexity of the specification in the appendix may seem surprising. But the Henley Regatta really does have boats, heats, regattas, winners, not yet determined boats, etc., and the relationships among them really are rather complex. Also, we know from experience that it can take quite some effort to learn how some unfamiliar sport is structured. It is clear that the methods for constructing and restructuring regattas really are rather complex. Moreover, this kind of complexity is not unique to this example, but is typical of sporting events, games, and many other social phenomena.

Now let us consider what *situatedness* means for this example. First, we must distinguish between the "actual" situated ADT and its formalization in OBJ. The formal code is fixed: it has 3 modules, 32 lines, etc. The structure of an actual Henley Regatta is much more elusive. No doubt there is a rule book; but there are also disgreements, which are negotiated by Stewards and other officials. Any actual Henley Regatta is *emergent* from the myriad interactions among members of a large group, and much of what goes on is *contingent* upon the *local* details of that particular context. It is *open*, in that we cannot hope to formalize everything that could potentially occur, and it is *vague*, because much of what goes on is unarticulated (tacit) and perhaps unarticulatable.

The qualities of situatedness impose limits on the utility of any formal specification. It is doubtful if the Henley Stewards would be interested in the OBJ code, and certainly the sports fans would not be. On the other hand, the code could be useful in designing a computer system to store and display the results of races. The trouble with the formal specification is that it is *too* precise and rigid; it fails to incorporate the richness and flexibility of an actual sporting event. But it does help bridge the huge gap between the rich situatedness of social interaction and the needs of those who develop systems.

Finally, we consider how a situated ADT could be justified by ethnographic data. Since we claim that members recognize that the "same information" is present in different representations, it is natural to find support for the structure in actual instances of such recognitions. Because these would likely be rare in naturally occurring data, requirements engineers could provoke such events by directly posing appropriate questions to members.

3.2 Two Case Studies

This section discusses two case studies done at Oxford using video-based requirements elicitation and situated abstract data types. They are based on live interactions, as opposed to the artificial case study of the Henley Regatta sketched above. The fieldwork was done by Marina Jirotka with help from Jonathan Hindmarsh, and the analysis was done by them with Joseph Goguen, Christian Heath, and Paul Luff. The situated ADT analyses are due to Goguen.

3.2.1 Financial Dealing Rooms

Financial dealers buy and sell financial instruments, such as stocks, bonds and futures. The process of recording a deal is called "deal capture." In the sites studied, this is done by the dealer writing the information on a ticket, often in a very rapid and abbreviated script; this is both time consuming and error prone. Moreover, errors in this task can be extremely costly to correct. A deal is officially a 6-tuple, consisting of the stock (or other financial instrument) name, quantity, date, buyer, seller, and price. Although this ADT is adequate for the legal purpose of registering a deal, it was found to be far from adequate for situations in the dealing room where one deal is part of a complex package of deals negotiated simultaneously, e.g., buying the same stock on one market and selling it on another, with a concomitant foreign exchange transaction.

Over the years, many technical systems have been proposed for making deal capture more accurate and less time consuming. Unfortunately, most systems have failed. These failures can be spectacular, as dealers, who may be angry, tense and very busy, simply throw the new equipment at the wall and then resume business in the old way. This has motivated increasingly radical suggestions for new technology.

One group believed that voice recognition could be used to automate deal capture, and was designing a system based on that technology. They claimed that a very limited number of words needed to be recongized, including numbers and some jargon. However, a detailed video-based analysis of several dealing rooms showed that voice recognition could not support the complex interactions that actually occur in this highly competitive environment [15]. This research potentially saved about a million pounds for prototyping and testing a system that could not succeed. A promising alternative is an active document system, possibly based on a touch sensitive LCD desktop, and virtual reality is promising for the long term.

3.2.2 A Telecommunications Central Operations Unit

A second case study considered a new integrated database for the Fault Restoration Office of a telecommunications Central Operations Unit. This office tries to restore service when lines go down, by finding and connecting alternative lines. Analysis showed that the new database being designed might actually make it harder to get the required information and do the job. An interesting ADT was found to be used by personnel, namely a directed labelled graph showing the faulty route and possible alternatives, labelling edges and nodes with capacities, locations, relevant phone numbers, etc.

It was also found that it would be counter-productive to use the huge video wall in the Central Operations Unit, because this could not support the kind of cooperative work that is actually done. For example, personnel often point at particular items, but pointing at a distant image does not enable co-present personnel to tell which item is indicated. A smaller display located on or above a desk, such that information can be accessed by touching relevant parts of the appropriate graph, should support cooperative work practices in a highly productive way.

4 Formal Methods

After discussing what formalization is, we discuss some limitations of formal methods, building on our previous discussion of the social context of computing. We then discuss the granularity of formal methods, and an emerging paradigm.

4.1 What is Formalization?

According to Webster's Dictionary, "*formal*" means definite, orderly, and methodical; it does not necessarily entail logic or proofs of correctness. Everything that computers do is formal in the sense that syntactic structures are manipulated according to definite rules. Formal methods are syntactic in essence but semantic in purpose.

The prototypical example of a formal notation is first order logic. This notation encodes the semantics of first order model theory with certain formal rules of deduction that are provably sound and complete. Unfortunately, theorem provers for first order logic can be difficult to work with. Formal notations can also capture higher levels of meaning, e.g., they can express certain requirements, but such notations will be much harder to work with, and will have fewer nice properties. By contrast, equational logic is simpler and computationally easier than first order logic, and has many pleasant properties.

The orderliness of social life (due to accountability, as discussed in Section 2.2) and the example in Section 3.1 suggest that social interaction might be formalizable; but there are limits to how successful any such formalization can be. In particular, it will not be easy to formalize domains where there are many *ad hoc* special cases, or where much of the knowledge is tacit. Formalization will be more successful on narrow and orderly domains, such as sporting events, that have long traditions, rule books, referees, regulating bodies, etc. For example, it would be more difficult to formalize a children's game than a boat race, and much more difficult still to formalize human political behavior. There are degrees of formalization, from dry to wet, and it can be important not to formalize beyond the appropriate degree. Cooking recipes are an interesting example, showing how an intermediate degree of formalization is possible and helpful, whereas a very formal treatment would be unhelpful, if it were even possible.

In the driest formalizations, the meta language is also formalized, so that the object level model is a formal theory in the meta language. In less fully formalized models, the meta language may simply be a natural language, or a somewhat stylized dialect. Note that there can be rules at both the object and meta levels. Rules at the object level are part of the model, while rules at the meta level define the language that is used for formalization. Any use of a formalism is situated. Therefore the qualities of situatedness impose basic limitations on any formalization: it will necessarily be emergent, contingent, local, open, and vague. All this is illustrated by the Henley Regatta example.

4.2 Limits and Problems

This section discusses six problems with formal methods:

(1) Formal notation is alien to most programmers, who have little training or skill in higher mathematics. This problem seems to be worse in the U.S. than Europe. For example, set theoretic notation is better accepted in Europe. This may be due to the higher level of mathematics education in Europe.

(2) Another problem is that some advocates of formal methods take a very dogmatic position, that absolutely everything must be proved, to the highest possible degree of mathematical rigor; it must at least be machine checked by a program that will not allow any errors or gaps, and preferably the proof should be produced by a machine. However, mathematicians hardly ever achieve, or even strive for, such rigor; published proofs in mathematics are highly informal, and often have small errors; they never explicitly mention rules of inference from logic (unless they are proving something *about* such rules). In fact, there are various levels of formality, and the most rigorous levels are very expensive; such efforts are only warranted for critical aspects of systems.

(3) A major problem is that formal methods tend to be inflexible; in particular, it is difficult to adapt a formal proof of one statement to prove another, slightly different statement. Since requirements and specifications are constantly changing in the real world, such adaptations are frequently necessary. But classical formal methods have great difficulty in dealing with such changes; we might say that they are a discontinuous function of how the hypotheses to be proved are formulated.

(4) Another problem is that formal methods papers and training often deal only with toy examples, and often these examples have been previously treated

in other formal methods papers. Although it may not be possible to give a detailed treatment of a realistic example in a research paper or a classroom, it is still necessary that such examples exist for a method to have credibility. To be effective, training in formal methods should treat some parts of a realistic (difficult) application.

(5) A technical deficiency of many formal methods is that first order logic is inadequate for loop invariants, as noted long ago by Engeler [6]. However, second order logic is adequate, and has been used by the authors for some years in teaching and research at Oxford [10] and the Naval Postgraduate School [2].

(6) Finally, the fundamental limits imposed by the qualities of situatedness imply that without human intervention, a formalization will often be inadequate for its intended application.

4.3 Small, Large and Huge Grain Methods

It is useful to distinguish among small, large, and huge grain formal methods. This distinction refers to the size of the atomic components that are used, rather than the size of the system itself. The "classic" formal methods fall into the small grain category. These methods have a mathematical basis at the level of individual statements and small programs, but rapidly hit a complexity barrier when programs get large. In particular, pre- and post- conditions, Hoare axioms, weakest preconditions, predicate transformers and transformational programming all have small size atomic units, and fail to scale up because they do not provide structuring or encapsulation. In general, small grain methods have great difficulty handling change, and thus fit poorly into the lifecycle of large complex projects. Transformational programming is less resistant to change than other small grain methods, but has the particular problem that there is no bound to the number of transformations that may be needed; this restricts its use to relatively small and well understood domains (see Section 4.4).

The main techniques of large grain programming involve module composition. We briefly describe an approach based on module expressions, theories, views, and a distinction among sorts for values, classes for objects, and modules for encapsulation. This allows expressing designs and high level system properties in a modular way, and allows the parameterization, composition and reuse of designs, specifications, and code.

The main programming unit is the *module*, which allows multiple classes to be declared together. Module composition features include renaming, sum, parameterization, instantiation, and importation. These constitute *parameterized programming* [8], which can be seen as functional programming with modules as values, theories as types, and module expressions as (functional) programs. *Renaming* allows the sorts, classes, attributes and methods of modules to get new names, while *sum* is a kind of parallel composition of modules that takes account of sharing. The interfaces of parameterized modules are defined by *theories*, which declare both syntactic and semantic properties. *Instantiation* is specified by a *view* from an interface theory to an actual module, describing a binding of parts in the theory to parts in the actual module; *default views* can be used to give "obvious" bindings. A design for a system (or subsystem) is described by a *module expression*, which can be parameterized, and can be evaluated to produce an executable version of the system. *Importation* gives multiple inheritance at the module level. Parameterized programming is implemented in OBJ [14], has a rigorous semantics based on category theory, and has influenced the designs of ML and Ada. Much of the power of parameterized programming comes from treating theories and views as first class citizens. For example, it can provide a higher order capability in a first order setting.

A major advantage of parameterized programming is its support for *design* in the same framework as specification and coding. Designs are expressed as module expressions, and they can be executed symbolically if specifications of a suitable form are available. This gives a convenient form of prototyping. Alternatively, prototypes for the modules involved can be composed to give a prototype for the system, again by evaluating the module expression for the design. An interesting feature of the approach is to distinguish between horizontal and vertical structuring. *Vertical structure* relates to layers of abstraction, where lower layers implement or support higher layers. *Horizontal structure* is concerned with module aggregation, enrichment and specialization. Both kinds of structure can appear in module expressions, and both are evaluated when a module expression is evaluated. We can also support rather efficient prototyping through *built-in* modules, which can be composed just like other modules, and give a way to combine symbolic execution with access to an underlying implementation language.

Parameterized programming is considerably more general than the module systems of languages like Ada, CLU and Modula-3, which provide only limited support for module composition. For example, interfaces in these languages can express at most purely syntactic restrictions on actual arguments, cannot be horizontally structured, and cannot be reused. LILEANNA [26] implements many ideas of parameterized programmming, including horizontal and vertical composition (following LIL [7]) for the Ada language. In [13], some further features are described: dynamic binding with views, abstract classes, private class inheritance, and dynamic integration of components from different libraries.

CAPS [19] is a rapid prototyping system with a data flow like semantics supporting hard real time constraints. It has module composition and powerful facilities to retrieve software components [17] and to support evolution [18].

Developing systems with huge grain components is qualitatively very different from working with small and large grain components. For example, very different ways to handle errors are needed. In systems with huge components, correcting errors in the components is generally impossible; such errors must be accepted and worked around. For example, a network protocol such as TCP/IP may have been obtained from an external vendor, so that the developers of the larger system will not have access to the code. If the version being used has a bug, there is no choice but to find some way to avoid that bug. This is often possible because of the multiplicity of features provided in such components.

4.4 Domain Specific Formal Methods

There is much more to formal methods than suggested by the themes dominant in the past, namely synthesis and correctness proofs for algorithms. Although both remain interesting for theoretical research, their impact on the practice of large scale software development is limited. A number of successful recent tools suggest a new formal methods paradigm having the following attributes:

1. A narrow, well defined, well understood problem domain is addressed; there may already be a successful library for this domain.
2. There is a community of users who understand the domain, have good communication among themselves, and have potential financial resources.
3. The tool has a graphical user interface that is intuitive to the user community, embodying their own language and conventions.
4. The tool takes a large grain approach; rather than synthesizing procedures out of statements, it synthesizes systems out of modules; it may use a library of components and synthesize code for putting them together.
5. Inside the tool is a powerful engine that encapsulates formal methods concepts and/or algorithms; it may be a theorem prover or a code generator; users do not have to know how it works, or even that it is there.

Some systems that fit this description are: CAPS [19]; ControlH and MetaH [27]; AMPHION [23]; Panel [22]; and DSDL [1]. This emerging paradigm might be *Domain Specific Formal Methods*, in recognition of the role played by the user community and their specific domain. This falls into the category of large grain methods, and can potentially be extended to huge grain problems.

4.5 Education

Teaching a formal method while ignoring the social, political and cultural problems that necessarily arise in real projects can have a negative impact. For example, students may be taught programming from formal specifications, but not that specifications come from requirements, and that requirements are always changing. As a result, they are not prepared for the rapid pace of evolution found in real industrial work. A related problem is that many students feel that formal methods turn programming from a creative activity into a boring formal exercise. The failure of teachers to deal with these problems has caused students to leave computing science.

Students need to know how to deal with real programs having thousands or even millions of lines of code. Carefully crafted correctness proofs of simple algorithms give an entirely misleading impression of what real programming is like. Most of the examples in textbooks and the classroom are very small, and most of the techniques are small grain.

Reliable formal method based tools can let students do problems that would be impossible by hand; this should increase their confidence. Teachers could also present methods and tools that work on large grain units, that is, on modules, rather than on small grain units like statements, functions and procedures, because such methods can scale up, whereas the small grain methods can not. It is desirable to develop suites of sample problems that systematically show how and when to apply formal methods, and how to combine them with informal approaches.

5 Software Evolution

A traditional view is that software evolution only occurs after initial development is completed. For example, software evolution has been defined to consist of the activities required to keep a software system operational and responsive after it is accepted and placed into production; this is synonymous with maintenance, but avoids the deadly negative connotation of that word. Evolution has the connotation of life, and if used in the context of an alternative software lifecycle like prototyping, it captures the dynamic aspects of all activities from requirements specification and system construction to updating operational systems [18].

Difficulties associated with evolution are not purely technical; social, political and cultural factors are important, and can dominate cost. Tools based on formal models can help with both technical and management tasks. They can maintain the integrity of a software development project by scheduling tasks, monitoring deadlines, assigning tasks to programmers, keeping on-line documentation, maintaining relations among system components, tracking versions, variations and dependencies of components, and merging changes to programs. These problems are especially important when a large group of programmers work concurrently on a large complex system.

An important practical problem is dealing with so-called "legacy code," i.e., old code that is poorly structured, poorly documented, and often in an obsolete language. For example, many banks depend on huge COBOL programs, but find it extremely difficult to modify these programs when business conditions change.

5.1 Hyper-Requirements

The Centre for Requirements and Foundations at Oxford has a project on improving the *traceability*, *accessibility*, *modularity*, and *reusability* of the numerous objects that arise and are manipulated during software development. An initial study administered a detailed two-stage questionnaire to requirements engineers at a large firm. Analysis of the results showed that there are many different traceability problems. Major distinctions are between pre-RS (Requirements Specification) traceability and post-RS traceability, and between forward and backward traceability. Analysis also showed that "access to users" was a very common difficulty. Further investigation revealed certain policies and traditions that restrict communication within the firm, so that requirements engineers often could not discover what users really needed. One problem was an "internal market" which restricted communication between "vendors" and "clients" within the firm. Abolition of the internal market for requirements projects and improving the openness of information could potentially save enormous sums for such firms.

A major aspect of the traceability problem is the difficulty of maintaining the huge mass of dependencies among components in a large system development effort. Often the components are not adequately defined, e.g., module boundaries may be incorrectly drawn, or not even explicitly declared; also, module interfaces may be poorly chosen and badly documented. Without formal models of dependencies and tool support for managing them, it is impossible to know what effect a change to a component will have, and in particular, to know what other components may have to be changed to maintain consistency.

The second phase of this project is designing a flexible object oriented database to support links among related objects [20], in order to ground decisions in the prior objects that justify them, and to track module dependencies. These links may be of a variety of different kinds, which are user-definable, and the objects may be in different media. Particular subproblems include formalizing dependencies, developing methods to calculate dependencies, and propagating the implications of a change. We intend to support the *situatedness* of requirements decisions, as well as their *traceability* through an idealized chain of stages. This associates related objects into what are called *module clusters* in hyperprogramming [9]. Techniques of parameterized programming, as described in Section 4.3, should improve reuse, and a generalized notion of *view* should help with organizing links. These techniques should be useful for design, when specifications are produced from requirements, as well for coding and maintenance.

A Formal Specification for the Henley Regatta

This appendix gives a formal specification for the Henley Regatta, using the executable part of the programming and specification language OBJ3, which is described in detail in [14]. First we give a rough overview of the specification below. There are three modules, beginning with the keyword "`obj`" and ending with the keyword "endo." Immediately after "`obj`" comes the name of the module. Sorts, operations (for methods), subsorts, variables, and equations are declared after fairly obvious keywords; also, "`cq`" indicates a conditional equation, while "`pr`" and "`dfn`" indicate module importations, the latter with a renaming of the principle sort (in this case, from `List` to `Index`).

The first module, named "12", merely introduces two constants, "1" and "2", used to indicate the two boats in a heat. Note that this is an abstract data type, in the sense that we could have chosen different representations for the two boats, such as "`A`" and "`B`", or "`1st`" and "2nd"; any such choice will yield an isomorphic (two element) initial algebra. The second module, named "LIST", is a *parameterized module* for forming lists of anything; the list constructor has the syntax "_ _" for placing a new element at the head of a list; `nil` is the empty list. Inside the third module, we form and import the module `LIST[12]`, renaming its principal sort `List` to be `Index`; these lists are used for picking out particular instances of a boat racing in a regatta; a typical term of sort `Index` is "`1 1 2 nil`". This module also introduces the constructor `heat` for regattas, the method `swin` for setting winners of heats, and the selector `winner`.

```
obj 12 is sort 12 .
  ops 1 2 : -> 12 .
endo

obj LIST[X :: TRIV] is sort List .
  op nil : -> List .
  op _ _ : Elt List -> List .
endo

obj HENLEY is
  sorts Boat Reg Boat? Reg? .
  pr QID .
  subsorts Id < Boat < Reg Boat? < Reg? .
  dfn Index is LIST[12].

  op ? : -> Boat? .
  op heat : Reg? Reg? Boat? -> Reg? .
  op heat : Reg Reg Boat -> Reg .

  var B B' : Boat? .
  vars R R' R'' : Reg? .
  var I : Index .

  op winner : Reg? Index -> Boat? .
  eq winner(heat(R,R',B), nil) = B .
  eq winner(heat(R,R',B), 1 I) = winner(R,I) .
  eq winner(heat(R,R',B), 2 I) = winner(R',I) .
  eq winner(B,nil) = B .
  cq winner(B,I) = ? if I =/= nil .

  op swin : Reg? Index -> Reg? .
  eq swin(heat(R,R',B), nil) = heat(R,R',B).
  eq swin(heat(R,R',B), 1 nil) = heat(R,R',winner(R,nil)).
  eq swin(heat(R,R',B), 2 nil) = heat(R,R',winner(R',nil)).
  cq swin(heat(R,R',B), 1 I) = heat(swin(R,I),R',B) if I =/= nil .
  cq swin(heat(R,R',B), 2 I) = heat(R,swin(R',I),B) if I =/= nil .
  eq swin(B,I) = ? .
endo
```

This code, with many test cases, has actually been run in OBJ3.

References

[1] Jeffrey Bell, Richard Kieburtz, *et al.* Software design for reliability and reuse: a proof-of-concept demonstration. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, 1994.

[2] Valdis Berzins and Luqi. *Software Engineering with Abstractions.* Addison-Wesley, 1990.

[3] Barry Boehm. *Software Engineering Economics.* Prentice-Hall, 1981.

[4] Graham Button and Wes Sharrock. Occasioned practises in the work of implementing development methodologies. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 217–240. Academic Press, 1994.

[5] Alan M. Davis. *Software Requirements: Analysis & Specification.* Prentice-Hall, 1990.

[6] Erwin Engeler. Structure and meaning of elementary programs. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, pages 89–101. Springer, 1971. Lecture Notes in Mathematics, Volume 188.

[7] Joseph Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986. Reprinted in *Tutorial: Software Reusability*, Peter Freeman, editor, IEEE Computer Society, 1987, pages 251–263, and in *Domain Analysis and Software Systems Modelling*, Rubén Prieto-Díaz and Guillermo Arango, editors, IEEE Computer Society, 1991, pages 125–137.

[8] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[9] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology.* Joint System Development Corporation, Tokyo, Japan, January 1990.

[10] Joseph Goguen. Proving and rewriting. In Hélène Kirchner and Wolfgang Wechler, editors, *Proceedings, Second International Conference on Algebraic and Logic Programming*, pages 1–24. Springer, 1990. Lecture Notes in Computer Science, Volume 463.

[11] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Marina Jirotka and Joseph Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic Press, 1994.

[12] Joseph Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105(2):217–273, 1992.

[13] Joseph Goguen and Adolfo Socorro. Module composition and system design for the object paradigm. *Journal of Object Oriented Programming*, to appear 1995.

[14] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen, editor, *Algebraic Specification with OBJ: An Introduction with Case Studies.* Cambridge, to appear.

[15] Christian Heath, Marina Jirotka, Paul Luff, and Jon Hindmarsh. Unpacking collaboration: the interactional organisation of trading in a city dealing room. In *European Conference on Computer Supported Cooperative Work '93*. IEEE, 1993.

[16] Steven Levinson. *Pragmatics*. Cambridge University, 1983.

[17] Luqi. Normalized specifications for identifying reusable software. In *Proceedings of the 1987 Fall Joint Computer Conference*, pages 46–49. IEEE, October 1987.

[18] Luqi. A graph model for software evolution. *IEEE Transactions on Software Engineering*, 16(8):917–927, 1990.

[19] Luqi. Real-time constraints in a rapid prototyping language. *Journal of Computer Languages*, 18(2):77–103, 1993.

[20] Francisco Pinheiro. TOOR: An object oriented tool for hypermedia requirements, 1994.

[21] Harvey Sacks, Emanuel Schegloff, and Gail Jefferson. A simplest systematics of the organization of turn-taking in conversation. *Language*, 504:696–735, 1974.

[22] Jacob T. Schwartz and W. Kirk Snyder. Design of languages for multimedia applications development. In *Proceedings of 1994 Monterey Workshop: Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development*, pages 46–55, 1994.

[23] Mark Stickel, Richard Waldinger, Michael Lowry, T. Pressburger, and I. Underwood. Deductive composition of astronomical software from subroutine libraries. In *Conference on Automated Deduction*, volume 12, 1994.

[24] Lucy Suchman. *Plans and Situated Actions: The Problem of Human-machine Communication*. Cambridge University, 1987.

[25] Stephen Toulmin. *The Uses of Argument*. Cambridge University, 1958.

[26] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings, Second International Workshop on Software Reuse*, March 1993. Lucca, Italy.

[27] Steve Vestal. Integrating control and sopftware views in a CACE/CASE toolset. In *Proceedings, Symposium on Computer-Aided Control Systems*, 1994.