CHAPTER 5

Tools and Materials

5.1 Introduction

An interactive system is seen by different people from different points of view. The system user is concerned with external properties, such as those that influence task coverage, flexibility and robustness during system use. The developer is often more concerned with those internal properties which address such things as the costs and reliability of development throughout the entire development life cycle.

The subject of this chapter is the interactions between properties and software techniques that are methodological in nature. Chapter 3 identified interactions with these software techniques as the next most significant after those with software architecture (this formed the subject of Chapter 4).

Effective and efficient use of methodological techniques is unlikely without tool support. For example, quality procedures, which validate that a system meets its requirements, are likely to fail without extensive support from tools and materials. Since these interactions are mediated by the use of various tools and materials, this chapter examines how these influence properties.

The wide range of tools and materials used for examples in this chapter (e.g. the use of automatic code generation techniques and hypertext approaches to requirements structuring) extends the subset of software techniques identified in Chapter 3. However, no attempt is made to cover all possible software techniques (e.g. change control tools and protocols).

5.1.1 Definitions

In the context of this chapter, *materials* are defined broadly as anything that someone in a development role produces for a specific project. In addition to specifications, implemented code, and evaluation reports, materials may include design documents, system administrator as well as well as user documentation, on-line help and tutorials, and various training aids. Even marketing strategies can be regarded as materials given the above definition. However, the definition unfortunately excludes any re-usable code that was not produced for a specific project (e.g. interaction toolkits). The definition could clearly be improved, but in order to keep it simple, the

C. Gram et al. (eds.), *Design Principles for Interactive Software*

comment may be added that anything that could be produced by a development role for a specific project can be alternatively provided by re-using existing materials.

Materials mark the boundaries of software development phases. They are used to pass information between phases (evaluation materials can reflect information back into a phase). In contrast, *tools* embody the activities that carry a project forward. Materials produced with tools in one phase are used either in subsequent phases to generate further materials, or within the same phase to refine other materials within it. For example, design specifications can be transformed by model-based tools into executable code, but they can also be analysed with evaluation tools in order to produce evaluation reports that guide further refinement of the design specification.

5.1.2 Potential Scope of this Chapter

The potential scope of this chapter is very large. For example, five broad categories of material can be identified.

1. **Requirements materials** specify the requirements for an interactive system. Requirements specialists select external and internal properties and allocate weights to them.
2. **Specifications and Design materials** provide a detailed description of the interactive system.
3. **Coded modules** implement the components of an interactive system. Implementers transform design materials into these coded modules, except where existing materials can be re-used.
4. **Working system**, i.e. the coded modules, bound together and executable, resulting in an interactive system performing useful work, and containing state information about a user's current interactive tasks.
5. **Evaluation reports** describe the strengths and weaknesses of a working system, partly expressed in terms of the properties identified in Chapters 2 and 3.

Development roles from Chapter 1 appear above (e.g. Requirements specialists, Implementers) and below. Recall that multiple roles can be filled by the same person, and that a single person may fill multiple roles.

The usual input/output relationship in any one phase is defined by the application of a transformation to the input in order to create the output (e.g. transform a task method into a dialog sequence). However, other forms of input/output relationship may be needed during system development. For example, in the case of mapping from requirements to specifications, a solely transformational approach cannot handle the pervasive nature of

'non-functional' requirements, because there is no simple mapping of requirements onto design features. In this case, a *checking* relationship predominates, designs being checked against requirements. In all cases, tools may be used to move between different categories of material. Five categories of such tools can be identified.

1. **Requirement tools** are used by requirements specialists to formulate requirements.
2. **Specification tools** are used by system designers to produce specification materials that describe intended solutions.
3. **Construction tools** are used by implementers to transform specification materials into coded modules.
4. **Execution tools** are used by system administrators to assemble and bind modules into interactive systems.
5. **Evaluation tools** are used by validators in evaluating interactive systems by exercising and measuring various usability aspects.

The above lists of categories of materials and tools cover a potential range that is so large as to make this chapter's analysis unmanageable. The working group therefore made several pragmatic decisions that restrict the scope of analysis. The scope is also restricted by a number of logical considerations that do exclude many tools and materials from the discussion.

5.1.3 Restricting the scope of this chapter

The main pragmatic restrictions result from an assumption that the most relevant materials for an analysis of interactions with properties are those that describe the final system, evaluate these descriptions, specify properties for the final system, or remain in the final system (i.e. as coded modules or resources that are referenced during execution). It is also appropriate to consider detailed architectures as materials (they are extensions of the architectural models considered in Chapter 4). Given the restrictions on 'relevant' materials, 'relevant' tools generate such materials (this extends transitively to all 'ancestor' tools and materials in a 'generation pipeline').

These pragmatic restrictions exclude tools that support standard methods from Usability Engineering (Nielsen, 1993), as these evaluate the final system rather than a description of it. This is a little late for our purposes. Such tools (e.g. Hix and Hartson, 1994) are placed outside the scope of this chapter, even though user testing is required to establish the satisfaction of all user-dependent properties (e.g. honesty), as well as properties given a low (even no) priority in an initial property profile.

A further pragmatic restriction is that we ignore hardware materials, even though pace tolerance is generally determined by system response time (which in turn is improved by high-performance processors, accelerator boards etc.).

Other restrictions placed on the tools and materials considered below can be supported by argument. Primarily, many tools and materials in routine use do not interact with properties. For example, general purpose text editors can be used to create specifications, but provide no support for internal or external properties. At best, such tools provide some support for development efficiency, but experiences with CASE over the last decade suggest that even this is open to question. Actual usage can often be an act of faith, and some tools used in good faith are actually detrimental to the achievement of high-quality interactive systems.

Further logical restrictions arise from the development life cycle. Tools and materials for the early phases of development can be ignored, since properties are not selected nor are weightings allocated until late in the requirements specification phase. Neither should the phases following the system test be considered, since properties should have been established long before system installation. The scope of this chapter may therefore be logically confined to the phases from the start of system design until the end of system testing (but with usability evaluation tools already excluded from the scope). These remaining phases may be collected into the following three groups.

Specification – the phases of system design, software design and module design.

Construction – the phases of coding, module tests and integration tests.

Evaluation – the system test phase.

The fine-grained phases of Chapter 1 are now, therefore, replaced by coarse-grained groups of phases. These are used to organise the analysis in the same way as functional partitions did in Chapter 4. However, there is one class of tools and one develoment practice that clearly cut across the above coarse groups. These must be considered before examining interaction in detail.

Model-based tools cut across coarse development phases. These address specification, construction and evaluation by automatically generating large parts of the final system. Clearly, however, internal properties may be assisted, as these tools integrate and manage activities that span most of the development life cycle. For example, ADEPT (Johnson *et al.*, 1995) begins with task analysis and user modeling, expresses requirements as a task model and a user model, and then generates intermediate models (design specifications) from which code is generated. Tools such as DON (Kim and Foley, 1993) and TRIDENT (Vanderdonckt and Bodart, 1993) also evaluate models for various qualities. Such tools can be accommodated by treating them as a family of implicit tools. Each implicit tool is used to address a separate coarse development phase.

Prototyping (see Section 1.3.1) cuts across coarse development phases. At

least three uses for a prototype are possible once an iterative development is halted.

1. The prototype and evaluation reports become materials for *requirements specialists*, who transform them into formal requirements that reflect the whole prototyping experience.
2. *System designers* draft formal specifications that capture key features of the prototype at all levels of abstraction – here prototypes are materials that specifications are checked back against.
3. The prototype contributes coded modules for use by an *implementer* of the final system (who will add modules for new functionality and/or user interface capabilities).

The third use arises with evolutionary prototypes, while the first two arise with rapid prototypes (as defined in Section 1.3.1).

Rapid prototypes can be constructed using paper and pencil, or with tools such as HyperCard (Goodman, 1993) and Director (Macromind, 1990). They can be evaluated, for example, to provide some confidence about interaction flexibility or robustness. Tools with explicit high-level configuration languages could even support proof of some properties. Rapid prototypes are thus *reference* materials against which requirements or specifications can be checked. They are strictly part of the early development phases and thus rapid prototyping tools are outside this chapter's scope.

When using rapid prototypes, the speed at which they can be produced limits evaluation to the external properties defined for the system since software quality standards (i.e. internal properties) must necessarily be relaxed in creating them. However, when developing evolutionary prototypes, high software quality standards must be maintained at all times since the prototypes develop into the final system for which high standards are wanted. It follows from this that evolutionary prototypes will usually be developed using commercially available construction tools.

For evolutionary prototypes, the relevant properties, tools and materials are no different to those for final systems that have been developed without prototyping. Such commercially available user interface development products include:

- UIMX from Visual Edge
- InterfaceBuilder from NeXT
- Prototyper from SmethersBarnes (SmethersBarnes, 1990)
- Visual Basic from Microsoft
- PowerBuilder from PowerSoft
- XFaceMaker from Non-Standard Logics.

In summary, this chapter will focus, for a mixture of pragmatic and logical reasons, on materials and tools that are used or produced during

specification and construction. Model-based tools can be analysed first for their specification support, and then for construction. Evolutionary prototypes can be regarded as being no different to any other final system for our analysis. The ten categories of tools and materials that could provide the scope for this chapter have thus been reduced to a manageable four (there are no evaluation tools or materials to consider).

Detailed discussion of relevant interactions begins by considering first those interactions between properties and tools/materials within the specification and construction phases of development (Sections 5.2 and 5.3 respectively). Examples will be drawn from a wide range of existing tools and materials. This analysis of interactions is then extended in two ways: first by examining three well-established tools across a representative range of properties (Section 5.4); then by presenting current practice in using such tools at four representative development sites (Section 5.5).

5.2 Specification Tools and Materials

The time consuming task of specification spans system design, software design and module design. Recall that the properties which must be satisfied during these design phases will have been selected and allocated weighting during earlier development.

5.2.1 Flexibility Properties

Consider first the need for flexible planning of task execution. This involves the properties of reachability, non-preemptiveness and multi-threading.

Reachability can be proved when using some notations, especially ones for dialog abstractions such as transition networks. The RAPID prototyping tool (Wasserman, 1985) configures dialogs using state transition networks, letting reachability be at least assessed at the dialog level of abstraction. Proof is obstructed by RAPID's traversal semantics – there are side-effects on transition conditions, input consumption and time-outs (Cockton, 1985). For tools with cleaner transition conditions and traversal functions, path algebras (Alty, 1984) can be used. These can compute the transitive closure of state transition graphs, and thus be used to assess reachability and representation multiplicity (for input, through multiple dialog structures). Such tools are still only present in research environments and applicable only to moderately sized systems (Alty and Ritchie, 1985).

Transition networks and similar dialog abstractions also support assessment or proof (given a formalization) of *non-preemptiveness*. Unfortunately, these abstractions effectively obstruct the *multi-threading* property. This is because networks can only represent interleaving of processes by having a path for every possible trace through the process complex. Interleaving two network-specified dialogs with m and n states respectively

Table 5.1 *Specification Phase Interactions between Tools/Materials and Interaction Flexibility Properties*

Property	Interaction	Comment
Reachability	Prove	Most straightforward with 'clean' dialog abstractions
Non-preemptiveness	Assess	By inspecting specifications that support proofs of reachability
	Deliver	By using dialog abstractions with process constructs
Multi-threading	Obstruct	By using any sequential dialog abstraction
	Address	By using dialog abstractions with process constructs
Device Multiplicity, I/O Re-use and Human Role Multiplicity	None	Dependent on construction tools/materials
Representation Multiplicity	Assess	Same relationships as observability with constraints, view controllers, model-based tools and cognitive walkthrough
Reconfigurability, Adaptivity and Migratability	None	Dependent on construction tools/materials

requires a combined interleaved network with $m \times n$ states, whereas production systems require only $m + n$ rules to interleave two rule sets of m and n rules (Hill, 1987).

This problem with networks can be overcome quite simply by directly addressing the *multi-threading* property by adding process constructs (England, 1988; Jacob, 1986), but care must be taken with the underlying *schedulers* that distribute events to different dialog networks. Often the schedulers will not preserve the desired properties of good concurrent processes. Multithreading and non-preemptiveness are thus best supported by using integrated specification constructs that address them directly, (e.g. Statecharts – Harel, 1988).

Table 5.1 summarizes interactions between properties and tools or materials (T/M). Overall, tool support for flexibility properties is biased towards properties that can be proved or directly provided in some form. Only reachability can be easily proved (and this will largely be at the dialog level).

Multi-threading and non-preemptiveness can be delivered by appropriate control constructs, which again are best suited to specialization for the dialog level of abstraction.

Support for interaction flexibility during specification is very limited, and largely restricted to flexible planning of task execution. This is because many of the capabilities required for interaction flexibility must be provided by construction and execution tools. Specification tools tend to lag behind construction and execution tools, but there are clear opportunities for better support here.

Classifying Interactions

Several new terms are introduced in the second column of Table 5.1, and these will be explained before returning to the main analysis. Each form of interaction is defined by a set of activities in which developers must engage in order to exploit the interaction (with the exception of *None* and *Obstruct*, see below). To define each form of interaction, we must first identify the defining activities. For the forms of interaction in Table 5.1, these are.

Specialization – developers use a basic knowledge of a property to instantiate a construct (e.g. instantiating an interaction specification for a pull-down menu using an appropriate* construct such as a transition network).

Formalization – developers must use extensive knowledge of a property to express it as a formal predicate.

Proof Discharge – developers must use formalization, specialization (to produce a specification) and extensive skills at following proof procedures to establish that a property holds for the specification.

Inspection – developers must use extensive knowledge of a property to inspect specifications (which ideally should be formed from instantiations of appropriate constructs).

Given these activities, four interactions can be defined as follows.

Delivery – involves none of the four activities, and yet a positive interaction still results (by use of appropriate tool or re-use of materials).

Proof – requires formalization then specialization then proof discharge;

Addressing – requires only specialization.

Assessing – requires inspection after specialization.

* An *appropriate* construct is implicitly defined as one that requires only a basic knowledge of a property to instantiate it. Once properly instantiated, in the sense that the instantiation is well formed, the property is delivered. The developer has to do nothing other than to 'fill in the blanks', although this may involve quite complex expressions.

The table also includes an *obstruction* interaction. This can be the opposite of either assessing or proof, as the degree of obstruction may be assessed by inspection or established by a proof procedure. Here, developers must engage in activities to establish that a property *cannot* be (fully) supported.

When there are no positive or negative interactions between a specific tool or material and a property, then the tool or material is said to be *neutral* with respect to this property. In this case, no combination of development activities using the tool or material could exploit a property or demonstrate that it was obstructed. 'None' in the table indicates that all the tools and materials that were considered were neutral with respect to the property in question.

There are clear advantages in defining interactions in terms of required development activities. Firstly, it reveals some forms of interaction as being specific to a development phase. Thus tools that support *proof* of properties are used during specification. Furthermore, some interactions during early phases create further tasks for later phases. Properties that are proved or addressed during specification must be *preserved* during construction (for addressing, this can be achieved by construction tools that address properties to the same standard as specification tools).

A second advantage of defining interactions in terms of required activities is that it reveals differences in entailed developer effort for each form of interaction. Delivery requires no further developer effort beyond use of appropriate tools or re-use of materials (as, for example, use of a true functional programming language will deliver *referential transparency* for all programs); proof requires extensive developer effort and skill – someone must produce design specifications, someone must formalize the property and someone must do the proof; addressing requires some developer effort – someone must form relevant parts of a specification by instantiating an appropriate construct (e.g. processes are appropriate for forming instances of non-modal dialog boxes); assessing requires developer effort and good human factors skills – someone must produce design specifications and someone must inspect them.

A third advantage of defining interactions in terms of required activities is that it reveals differences in likely attainment of each form of interaction. Consider, for example, the activities involved in proof. Only the specialization activity is straightforward (for developers who can use specification tools). In contrast, formalization of properties can be very frustrating. It was not completed for any informal property in Chapter 2, despite several efforts. Many apparently acceptable predicates turn out to be too weak or too strong, i.e. they admit or exclude design features that do not or do support the property that they should formalize. Proof discharge is also a risky enterprise. The ability to prove a property depends on the notations used. Thus proof procedures for some properties are well understood when established constructs such as transition networks and context-free gram-

mars are used. However, Chapter 4 identified no proof interactions between architectural models and properties, for although *architectural description languages* are being developed (Garlan and Shaw, 1993; Luckham *et al.*, 1995); this work is still at an early stage.

In summary, forms of interaction between tools/materials and properties have a straightforward definition that involve one or more development activities in a particular order. As long as the simple basis of their definition is remembered, they support valuable analyses that identify what developers must do and when they must do it. This is especially valuable when identifying properties that can be ignored from an early stage in development, cannot be attended to during long periods of construction, require little developer effort, require much developer effort, can be reliably exploited or incur risks of failure. Having presented the basis for such judgements, we can resume the main analysis.

5.2.2 Robustness Properties

Flexibility properties are quite general and can thus often be addressed by general formal computing constructs, at least for flexible planning of task execution. In contrast, robustness properties require more constructs specific to interactive systems. As will be seen, general constructs can be used for pace tolerance, but several properties can only be assessed by using walkthrough techniques. Overall, several constructs are often required to address a robustness property comprehensively. This is particularly the case with deviation tolerance. But specific architectural support is needed for observability, and this robustness property is examined first.

Observability is defined as the possible rendering (at the logical level of abstraction) of relevant state (at the functional level of abstraction). Architectural models described in Chapter 4 support separation into different levels of abstraction. So do UIMSs that link functional components to interaction components via a dialog component.

When such architectures are embodied in specification tools, then *link constructs* are required to address observability. A link construct is any specification construct (or software entity) that forms connections between separate architectural components. A range of link constructs is mentioned in one of the example site reports (Section 5.5.2 below).

Dialog functions that use specialized link constructs address observability. This holds because information must be rendered to become observable, and dialog functions are responsible for initiating such renderings (by transferring selected information from the functional core adapter to selected logical interaction functions). Hence link constructs such as SERPENT's View Controllers (Bass *et al.*, 1990) address observability by encapsulating design decisions on when and how to render information.

Developers need not be aware of link constructs in model-based user

interface development tools such as ITS (Wiecha *et al.*, 1990), Humanoid (Szekely *et al.*, 1993), and UIDE (Sukaviraya *et al.*, 1993). They generate appearance and behavior from higher-level models, addressing observability by (semi-)automatically linking functional level models to presentation models.

Lastly, link constructs can also partially deliver *honesty* as long as the system is *pace tolerant* with minimal delays between functional state and corresponding display updates, since a value on the display should always accurately reflect its underlying value. Automatically created links can promote honesty if pace tolerance conditions are met.

In the absence of architectural support, *observability* and related robustness properties (i.e. *insistence*, *honesty*) can be assessed in specifications, primarily by combining design descriptions with walkthrough procedures. For example, *Cognitive Walkthrough* (Wharton *et al.*, 1994) combines task descriptions in any format chosen by developers with a simple set of questions. Four questions are asked at any interaction point:

1. Will the user try to achieve the right effect?
2. Will the user notice that the correct action is available?
3. Will the user associate the correct action with the effect they are trying to achieve?
4. If the correct action is performed, will the user see that progress is being made towards solution of the task?

Two of these questions address two stages of Norman's Seven Stage Model of Human-Computer Interaction. In this model, users cycle through command execution and result evaluation (Norman, 1986). Before entering commands, users must work out what to enter. Norman calls this stage 'action specification', and question 2 addresses it. Question 4 addresses all three of Norman's three result evaluation stages (perception, interpretation and evaluation).

There are clear similarities between some robustness properties and Cognitive Walkthrough questions.

Predictability is covered by question 3;

Honesty is covered by question 4;

Insistence is covered by question 2 (and question 4 is implied by honesty);

Observability: both questions 2 and 4 are implied by insistence.

Cognitive Walkthrough thus supports assessment of four robustness properties. It is, however, a paper-based tool, although specification tools could easily be extended to step designers through each of the four questions at each interaction point.

Other assessment methods focus on a single robustness property. For example, predictions of learnability made by Cognitive Complexity Theory

(CCT) let *predictability* be assessed (predictable systems are more consistent than unpredictable ones and thus require fewer rules than a user model for an inconsistent system), although the effectiveness of CCT is disputed (Knowles, 1988).

CCT focuses on a single robustness property, and it also requires a complete system model for CCT at some level of abstraction (usually the dialog level). In contrast, for Cognitive Walkthrough, task descriptions may only give partial coverage at mixed levels of abstraction. In short, designers can walkthrough what they *imagine* the system to be, rather than what some specification says *it is*. There are trade-offs between developer effort and comprehensiveness here. Only partial assessment can be expected unless task descriptions are formally derived from a complete system specification.

Considering other robustness properties during specification, *pace tolerance* can be assessed, and perhaps even delivered, by general computing constructs. Real time specification languages can be combined with real time scheduling algorithms to establish that a response to an event will occur within a given time (Burns, 1994). For example, rate-monotonic scheduling algorithms (Sha and Sathaye, 1993) can be used to establish pace tolerance, although they currently ignore system overheads. They also largely address hardware issues such as bus protocols. Their applicability to current interactive systems is limited, but progress with such approaches could be relevant for pace tolerance. With control over processing time, predictability in the form of response time stability would also be addressed.

More specific tools can focus on interface details that impact pace tolerance, for example, messages that are displayed and removed under system control. Users must be given time to read these. Algorithms for calculating the necessary duration of a message exist (Bevan, 1983), and thus the times could be specified. A focused tool could address by making such calculations.

The authors are aware of no interactions between specification tools or materials and the robustness property of *access control*.

The remaining robustness property, *deviation tolerance*, must be addressed by a wide range of constructs during specification. For example, input validation constructs, which are increasingly found in user interface builders, only address the error detection aspect of deviation tolerance. The property must be further addressed by constructs for error prevention and error recovery. For example, error prevention constructs can be found in screen layout tools that include constructs for preventing users from performing inappropriate commands (e.g. the presentation of an undesirable command can be visually distinguished at the logical interaction level).

Error prevention can be given more general support by specification languages with command pre-conditions. The earliest UIMS work here was by Mark Green (1985). Pre-conditions over states at the functional level

are used in the UIDE environment by Sukaviraya *et al.* (1993), where they support automatic generation of various user support features such as intelligent help (Sukaviraya *et al.*, 1992). More extensive pre-conditions, which support error detection, error prevention and error recovery, are found in the NUF notation, a specification notation for the functional level (Cockton *et al.*, 1995). This has four types of pre-conditions for abstract commands: availability, prevented failure, automated recovery and mixed-initiative recovery.* Availability pre-conditions prevent users from initiating a command (typically by greying it out and making it unselectable). Prevented failure pre-conditions are the simplest form of error detection, which merely specify an error state from which no further recovery is possible. Automated recovery pre-conditions specify an error state from which automated recovery is possible. Mixed-initiative recovery pre-conditions specify an error state from which recovery is possible, but only with user involvement.

Consider, for example, a document editor which offers a command to the user to save the document as a file to be named by the user as part of the command interaction (sometimes known as a 'Save As' command). When no documents are being edited, this command is made unavailable (availability condition is a non-empty set of open documents). The editor must also prevent the file save failing because an unacceptable file name is given (prevented failure condition is presence of special characters, incorrect `<prefix>.<suffix>` format, or name too long), as well as negotiating mixed-initiative recovery for the case where the named file already exists (that is, the error detection condition) – to prevent unintentional over-writing of contents (error recovery takes the form of a yes/no/cancel question). Also, where a programming environment offers a command to run the program being developed it can automate error recovery where the current version of the source code needs recompilation before execution (the error detection condition is 'source changed since last compilation', and error recovery takes the form of recompilation).

Current model-based tools do not provide such recovery mechanisms. In Humanoid (Szekely *et al.*, 1993), for example, side effects must be used to provide for error recovery. In UIDE (Sukaviraya *et al.*, 1993), a generated dialog model may have to be extended by hand to include recovery. Support for the deviation tolerance property may therefore be diminished between specification and construction.

Overall, there are few sufficiently general constructs that address robustness properties. Developers must thus wait until construction phases where re-usable library materials can provide specialized assistance for specific

* Mixed-initiative recovery involves both the user and the system in deciding whether to quit without saving.

Table 5.2 *Summary of Specification Interactions between Tools/Materials (T/M) and Interaction Robustness Properties*

Property	Interaction	Comment
Observability	Address	Constraints/View Controllers
	Assess	Model-Based User Interface Generators
		Cognitive Walkthrough Questions 2 and 4
Insistence	Assess	Cognitive Walkthrough Questions 2 and 4
Honesty	Assess	Cognitive Walkthrough Question 4
		Temporal aspects assessed in conjunction with both observability and response time conformance
Predictability	Assess	Cognitive Complexity Theory (but effectiveness disputed (Knowles, 1988))
		Response Time Stability assessed along with pace tolerance
		Cognitive Walkthrough Question 3
Access Control	None	Dependent on construction materials
Pace Tolerance	Deliver	Real time scheduling algorithms (potentially)
Deviation Tolerance	Address	Partial support from UI management tools/builders with input validation construct
	Address	Pre-conditions as used in NUF (Cockton *et al.*, 1995) and Model-Based User Interface Generators
	Assess	Cognitive Walkthrough can establish effects of errors

design features. Interactions between tools/materials and robustness properties are summarized in Table 5.2.

Robustness properties are given more extensive, but less effective, support than flexibility properties by tools and materials. Only observability, pace tolerance and restricted forms of honesty and deviation tolerance can be addressed. Otherwise, assessment is the best support available. For example, the combination of cognitive models and walkthroughs allows assessment of insistence, honesty, predictability and deviation tolerance. This

is because these properties are more user-dependent than the other robustness properties. They can be assessed during specification, but cannot be re-tested until evaluation. This poses a problem in that they cannot be attended to during construction. There is thus a gap between their assessment by analysis and their confirmation by (user) testing. This kind of insight is a further benefit of exposing different forms of interaction between properties and tools/materials.

5.2.3 Internal Properties

The main interactions between tools/materials and internal properties during specification are with development efficiency, modifiability and user interface integratability. Other interactions are minor, such as those with evaluability, portability and maintainability, which are due to positive interactions with architectural models (i.e. appropriate architectures will promote these properties, and thus properties achieved for an architectural model must be preserved during architectural design). Minor interactions here between these properties and architectural models must be preserved during architectural refinement in the software design phase. In particular, modifiability must be carefully considered during all refinements. However, all current model-based tools impose architectures on the final systems, and may thus obstruct properties that interact with architectural models.

Support for external properties can obstruct *run time efficiency* unless steps are taken to counteract this. One possible step is to use *virtual separation* (Shevlin and Neelamkavil, 1991). This can be used to reduce the tension between, for example, observability and run time efficiency by not generating separate coded modules for different levels of abstraction in the final system. Separation is thus *virtual*: it exists during specification but is not preserved in the final system (in Figure 1.4, there would be no separate FCX and UIS as the binding services would create PAC-like agents instead).

Virtual separation lets properties be addressed when they are most relevant during specification. Once established, specification constructs that address them (e.g. link constructs for observability) can be compiled away in the interests of run time efficiency. Returning to more major interactions, *development efficiency* is clearly a key property for specification and design tools. Tools that automatically generate the user interface such as TRIDENT (Vanderdonckt and Bodart, 1993), ADEPT (Johnson *et al.*, 1995), UIDE (Sukaviraya *et al.*, 1993), ITS (Wiecha *et al.*, 1990), and Humanoid (Szekely *et al.*, 1993) improve development efficiency by reducing development decisions. For example, UIDE automatically chooses the appropriate interaction object from interaction specifications.

Tools should deliver known degrees of development efficiency. One way to establish such degrees is to use tools for bench-mark (standard) devel-

opment tasks, and then to assess the speed up over untooled development. For limited tools such as user interface builders, the task for creating a 'hello world' pop-up window is often used to compare their efficiency with toolkits and lower-level libraries.

Benchmarks, however, are only a start. The development efficiency of tools should be assessed for real work (to do this, a software team need a good guess at how long tasks took without tool support). It is questionable to use tools without known levels of development efficiency. For such tool usage to be worthwhile, there must be extensive compensating support for other properties.

Development efficiency is very dependent on the appropriateness of constructs supported by a specification tool. For example, several constructs can be used to specify the dialog level of interaction. However, each construct is biased towards a specific set of dialog requirements (e.g. sequence, interleaving, permutation, ease of walkthrough), and thus a tool based on an inappropriate construct for a system's requirements can obstruct development efficiency. As a simple example, consider simulating interleaved processes with state transition networks. This would quickly bring development to a halt, due to the explosion of interaction points when interleaving two or more processes (see page 139).

A degree of development efficiency can also be *delivered* by reducing the number of design decisions that developers must make. User interface standards attempt to do this by taking many design decisions away from developers. GUI guidelines standardize many features, but mostly for 'look and feel' at the logical interaction level. Example guidelines include CUA (Windows and OS/2), Motif style guidelines (OSF, 1990), and guidelines for constructing Macintosh user interfaces (Apple, 1992). Guidelines may come in printed or on-line versions (Sadler, 1993), and there is anecdotal evidence that the latter format is preferred by developers.

Conversely, development efficiency is reduced when written style descriptions are ambiguous or incomplete. To take a detailed example, the Windows 3.1 Application Design Guide (Microsoft, 1993a) did not specify what should happen on pull-down and pop-up menus when the mouse re-enters the menu with the left button depressed. Developers often copied the common option of ignoring this event and forcing the user to re-select a menu (title). However, this obstructed deviation tolerance by penalizing users who slip off the bottom or side of a long menu. This omission was rectified for the Windows 95 style guide (Microsoft, 1995).

Guidelines are be best developed using formal notations, where ambiguities and incompleteness would be easier to detect (Chen, 1993). These could be left as they are for developers who can read them, and re-expressed in natural language for those who cannot. Better still, interaction techniques should be embodied in materials for use at the construction stage (e.g. the

Macintosh tool box, Visual Basic and similar implementations of Microsoft Windows style guides (Microsoft, 1993a, 1995)).

In summary, appropriate specification constructs and thorough unambiguous guidelines result in favorable interactions between *development efficiency* and related tools and materials.

The second significant interaction with an internal property during specification concerns *modifiability*. Absolute unconditional modifiability is only encountered in science fiction. Modifiability only extends to known classes of potential change for which a software architecture is known to be suitable.

Further interactions with modifiability result when system designers and software engineers use the same tools to modify a system as they use to initially design and implement the system. The system designer and software engineer (i) modify the original requirements, (ii) modify the specification to reflect the changes in the requirements, and then (iii) modify the code to reflect the changes in the specification.

This three-step process is facilitated if the system designer and software engineer use tools that automatically transform materials at one level into more detailed materials at the next lower level. If system designers and software engineers avoid manually modifying materials at lower levels without modifying the corresponding material at the next higher level, then no system modifications will be lost when the system designer and software engineer use this three-step process to modify a system. Modifiability can thus be further assisted by the use of model-based generation tools. The use of these tools has largely been confined to research teams, although ITS was used successfully to generate the public information system at the Seville EXPO (Wiecha, personal communication).

ITS demonstrates the potential of generators in user interface development. Modeling facilities in ITS proved to be adequate during the EXPO, despite significant changes to the public information system following observation of its usage. Most modifications added capabilities to the functional core (e.g. broadcast of text and images for update to the electronic news service). Since EXPO, three more applications of similar complexity to the EXPO system have been built and developers other than the EXPO team now use ITS within IBM (Wiecha, personal communication).

If tools that automatically transform materials are not available, then the system designer or software engineer must manually change the requirements, the corresponding specifications and the corresponding code. Given that many non-functional requirements are pervasive constraints on design and implementation decisions, it is hard to generate specifications from a full range of requirements. The only tool support for what seems to be an inherently manual activity comes from design rationale tools and from Hypertext links between requirements and design decisions (Kaindl, 1993). Such tools assist modifiability by keeping developers aware of all

the requirements that relate to a design feature under modification. The efficiency of modifications is improved as a result. The need for maintenance will be further reduced by avoiding modifications that adversely affect related requirements because developers were unaware of their relevance when designing the new modification.

User interface integratability can also be assisted by specialized user interface development tools (e.g. UIMS, UI builders). Consider two different applications with different user interfaces that have not been well engineered (for one, there is little documentation, also no tools were used in their construction). If they must be integrated, it may be necessary to reverse engineer design or requirements specifications from working systems. Specification tools assist with the reverse engineering of designs, but this does not provide enough support. For simple applications, and ones with limited interaction, reverse compilers, cross-compilers, and high-level translators are currently used to support integration. However, we know of no such tools that provide extensive assistance for user interface integration. If they did exist, they could perhaps even *deliver* user interface integratability.

Table 5.3 summarizes the interactions between tools/materials and internal properties during specification phases. Internal properties are more evenly covered than external ones during specification, although much of this depends on preservation of properties supported by architectural models.

5.3 Construction Tools and Materials

The coarse phase of construction spans module coding, module tests and integration tests. Properties that have been delivered or proved during specification must be preserved during these phases. Preserving properties from specification requires considerable developer effort if there are no simple equivalences between key constructs in specification notations and the constructs provided by construction tools.

Properties that have been addressed or assessed during specification can only be systematically preserved by formal program transformation. Alternatively, a constructed system must be shown to conform to a specification after each software design decision.

Properties that could not be addressed during specification can be *assisted* at this stage by the use of appropriate materials. The materials here are always some form of re-usable code modules, which can be services within a target environment or capabilities provided by class or module libraries.

Table 5.3 *Specification Interactions between Tools/Materials and Internal Properties*

Property	Interaction	Comment
Development Efficiency	Deliver	UI Management Tools/Builders with validated efficiency, but such tools are rare Model-based UI generators (mostly research and industrial prototypes) Appropriate specification abstractions, but only dialog level abstractions are well established Detailed unambiguous style guides, but these are rare (toolkit implementations for construction are better)!
System Modifiability	Deliver	Architectural refinement, but only for anticipated potential changes Model-based UI generators (mostly research and industrial prototypes) Hypertext requirements linking tools such as RETH (Kaindl, 1993)
User Interface Integratability	Deliver	Limited support from general (UI) tools
Run Time Efficiency	Deliver	Virtual separation
Portability, Evaluability and Maintainability	None	Preservation of property from architectural model

Assisting: a Further Form of Interaction

In Section 5.2.1 the possible interactions between properties and specification T/M were split into five (or six) classes: inspection, delivery, proof, addressing, assessing (and obstruction). It may now be useful to make a further distinction netween addressing and a weaker interaction *assistance*, where a T/M gives an implementer *some* constructs that may be used to obtain a certain property.

Assistance is specific to construction, where re-use of code (and tools) is worthy of consideration. As with other forms of positive interaction, it is defined solely in terms of the developer activities required to exploit the interaction. Assistance requires two activities: implementation followed by

specialization. Implementation is an activity specific to the construction phase.

Implementation – developers use extensive knowledge of a property to program an appropriate construct by using general system, programming or algebraic constructs (e.g. implementing Yang's (1988) two-stack undoing model using standard imperative data structures).

Some forms of interaction between properties and tools/materials were introduced in Section 5.2, but assistance is a distinct form of interaction. Assistance interactions are not instances of addressing, because the materials involved require more than basic knowledge to specialize them. Assistance interactions are not instances of neutrality because the materials involved do provide some support. Assistance is thus in-between 'what we want and what we have got'. It is better than nothing, mostly because assisting materials or basic tool constructs can be combined, refined and extended to form appropriate constructs that do address a property.

It is important to distinguish between assistance and addressing. The difference is due to the development activities required for each. To address a property, a tool or material must provide a general construct that only needs to be specialized. This can involve setting attributes, filling in slots, specifying logical conditions or naming functions and procedures to be called to perform a specific function. Specializing a construct generally instantiates one part of a design. Where the construct addresses a property, this instantiation will deliver the property for the corresponding design element.

To assist a property, a tool or material need only provide the means for implementing a general construct that can then be further instantiated to deliver a property. The implemented construct will thus address the property, but it must be implemented. This is the key difference, although it is easy to overlook. With assistance interactions, implementation of constructs that address properties is possible and perhaps even relatively straightforward, but there is no guarantee that its existence or potential will be realized and exploited. Furthermore, constructs that address properties must be implemented and specialized before any assessment is possible. Such assessment will often require users to interact with the system – inspection of coded modules will not suffice.

The difference between addressing and assistance is illustrated by the development of structured programming. Specialized constructs in structured programming languages directly address iteration, selection and procedural abstraction. The provision of stack frames and a run time stack directly addresses the needs of recursion. In contrast, assembly languages only assist with these fundamental control constructs and information structures (selection, iteraction, procedures, stack frames for recursion), since jumps, comparisons and stack pointers must be skilfully combined to implement

structured programming constructs. Assembly language programmers still fail to make use of these constructs, with obvious implications for software quality.

Assistance with an external property can clearly vary, but variations will generally be reflected in development efficiency. An overall evaluation of tools and materials can thus distinguish different levels of assistance. With this diffuse form of interaction introduced, interactions between properties and tools/materials during construction can be analysed.

5.3.1 Flexibility Properties

Flexibility properties are rarely proved or delivered before construction. Requirements and specification tools usually only indirectly support these properties by letting developers specify a system's requirements. Construction tools actually implement the materials that establish properties.

Properties that concern the flexible representation of information require extensive run time support. These run time materials must be in place at the outset of construction stages.

Device multiplicity is delivered directly by resource managers, such as that found in graphics libraries and window systems. However, window systems (notably X Window, Scheifler *et al.*, 1992) may restrict devices to a raster display, a keyboard, and a mouse with up to five buttons. Thus while some device multiplicity may be provided, it may not take the form required for a specific system.

When providing device multiplicity, and thus multiple foci of control, various resource managers must cooperate with each other. Support for such cooperation could be handled by a further software component which can be viewed as a resource manager manager. A resource manager manager must handle cross-resource manager transfer and sharing of control, while handling the synchronization of various resource managers. An example application for a resource manager manager is the synchronization of real time video with real time audio, where the cooperation of audio and video resource managers must be controled by a yet higher level manager.

World Wide Web browsers are now providing extensive support for device multiplicity. Some web browsers adapt layout according to the display device in use. These capabilities have been extended to input widgets in tools such as Sun's Hot JAVA (Gosling and McGilton, 1995).

Representation multiplicity is assisted (for output) by mechanisms such as the View Controllers in SERPENT (Bass *et al.*, 1990). Indeed, these link constructs almost address the property, since representation multiplicity is easily supported by having multiple view controllers for a single functional value. Still, there is no construct to encapsulate the representations for a single (group of) value(s), and thus developers must manage the modularization themselves. In contrast, the DIAMANT UIMS (Trefz and Ziegler,

1989) does have a representation manager that directly addresses representation multiplicity by encapsulating the relevant information.

'Separable' user interface tools support separate specification at different levels of abstraction for interactive systems. They can assist with representation multiplicity for input. The dialog notations in such tools (e.g. UIMSs) may be used to configure alternative user inputs, and also support their translation into a common functional level representation. No further effort from the developer is required. The tool itself preserves properties configured during specification, so developers need not attend to them during construction. However, most current construction support for flexible representation of information is in the form of materials rather than tools.

Moderate assistance for representation multiplicity (output) is provided by the standard Smalltalk methods for Model-View communication. These have been generalized in the paradigm of access-oriented programming and the associated use of *Active Values* (Myers, 1988). When state values are 'active', pre-specified actions are triggered when a value is changed. Active values almost directly address representation multiplicity (and also observability) by letting multiple rendering actions be triggered whenever a value is changed. The actions associated with a value change do encapsulate the multiple representations of that value, but they also encapsulate other behaviors associated with value changes.

Access-oriented programming is largely restricted to research systems. Much more restricted support is provided by typical target environments, for example, application events such as those found in version 7 of the Macintosh operating system (Apple, 1993). Here, different programs can send and receive arbitrary events, once they have registered them, and interests in them have been noted. This is a basic capability that requires detailed development effort, and can reduce development efficiency. However, it could provide direct support for access-oriented programming. For example, in an object-oriented programming language, the assignment operator could be overridden for a class of 'active' objects. The overriding assignment operator would make the assignment, but also call all the methods in a dependency list. These methods may be imperative procedures, or simply broadcast events to notify the value change.

This example demonstrates the difference between different extents of assistance. Neither active values, nor View Controllers, nor application events directly address representation multiplicity. However, each requires increasing levels of developer effort and expertise to create constructs that do address representation multiplicity. If developers are unaware of access-oriented approaches, they may program an event to be raised when the functional state is changed, but they will also have to program the dialog to respond to this event by propagating display changes into logical interaction events. The extent of assistance is thus crucial in any evalua-

tion of tools or materials, since this interaction covers support that almost addresses a property to interactions which come close to neutrality.

Representation multiplicity is given broader, albeit conceptual, support in the PAC model. Specialized PAC components such as Multi-View agents (Nigay, 1994) can support representation multiplicity for output. For input, PAC-Amodeus constructs such as melting-pots (Nigay and Coutaz, 1995) assist in the provision of multi-modal representation multiplicity. However, re-usable materials for these constructs are not widely available, so most developers will have to specialize more general constructs in order to implement these PAC concepts.

I/O re-use is assisted by inter-application communication facilities. Re-usable code for clipboards is one specific example. Such code delivers I/O re-use, as do history modules. An alternative to modules for history support comes from multiple inheritance in object-oriented languages such as Eiffel, where all interactive objects can inherit the capabilities of a history class (Meyer, 1988).

Whatever the mechanism for I/O re-use, there must be compatibility between the source and the target of the re-used information, and this usually must reflect different levels of abstraction. For example, plain text is a material at the logical level of interaction, as its pure ASCII format is device-independent. Re-use of such values is easy to provide, whereas other values present more difficulties. For example, re-use of commands at the functional level is not straightforward, nor is re-use of exotic media, such as real time video at the physical level.

Within single systems, compatibility of re-used information can be addressed, although development effort can be high. However, re-use between systems requires standards (e.g. OLE, Williams, 1994) that allow the re-use of information between disparate systems such as splicing of a video image into a spreadsheet. Extensive re-use is hard without standards. Thus the *LiveText* prototypes developed at AT&T (Fraser and Krishnamurthy, 1990) could achieve only a fair level of I/O re-use on the basis of existing Unix text output conventions. More extensive I/O re-use was seen to require new standards, for example the output of records similar to those found in text editor 'piece-tables'. Such standards would have required major departures from text I/O conventions for Unix commands.

To achieve properties for flexible planning of task execution requires skilled use of materials. For example, groupware toolkits (Gibbs, 1989; Knister and Prakash, 1990; Dewan, 1993) deliver *human-role multiplicity*. Similarly, resource management code addresses multi-threading directly, by letting multiple processes share the same physical devices and related resources, but multi-threading constructs need to be used with skill.

Resource managers also assist in the satisfaction of *non-preemptiveness*, since multiple processes make pre-emption easy to avoid. However, this automatic provision may obstruct the satisfaction of pre-emptiveness when

this is required. This is because non-preemptiveness is always delivered along with multi-threading. Developers must thus implement extra constructs to re-introduce pre-emptiveness.

Reachability can be proved during specification and preserved during construction. Alternatively, reachability can be delivered by construction materials such as re-usable history modules (Berlage and Spenke, 1992). Such modules provide generic capabilities for stepping backwards and forwards through the interaction history, as well as skipping unwanted steps.

Properties that relate to flexible representation of information (representation multiplicity, device multiplicity, I/O re-use) and flexible planning of task execution (reachability, non-preemptiveness, multithreading, human role multiplicity) require extensive run time support. So too do properties that address adaptation of dialog forms (reconfigurability, adaptivity, migratability).

Properties that address adaptation of dialog forms are currently almost entirely supported by materials. Only *reconfigurability* is given extensive support (see below). Other properties are less well supported. For example, *adaptivity* can be provided by specialized articicial intelligence (AI) tools such as User Modeling shells (Kobsa, 1990), but the property requires more extensive support than this. The most successful widespread adaptive approach is 'plug and play' as used by hardware manufacturers to ease the installation of various user interface devices such as mice, video interfaces and audio interfaces. These are intelligent devices that have knowledge of what other kinds of devices can be plugged into the parent system. When the devices are plugged into the system they autonomously determine what other devices are present and using this information configure themselves appropriately so as not to interfere with the other devices. This relieves users from having to manually resolve bus conflicts and similar complex problems. As this affects devices, the property of device multiplicity is also supported, and it provides a much needed alternative to the restrictions of current window managers.

No tools that provide specialized support for *migratability* are known. However, when functions or tasks migrate to the system, some system component must take control. Materials in some form are required. Such components are often called *agents*. These may operate at the dialog level (e.g. Microsoft's Wizards), and thus relieve users of planning decisions. Other agents can operate at the functional level. Software materials could provide re-usable agent 'skeletons', but we are unaware of such support being currently available in any form.

Returning to *reconfigurability*, construction tools and materials support this in many ways, but there is no overall coherence at present. In the past, reconfigurability has been assisted by materials that underpinned the 'table-driven software' approach. Here configuration files hold values that set various system options, such as the right margin setting in the case of a

text editor. This approach requires a module to read configuration files at start up, and to then modify the state at any level of abstraction to reflect expressed user preferences. Each user can have their own configuration file, and thus their own view of how the system should perform.

Reconfigurability can be obstructed by virtual toolkits, especially if they take a lowest common denominator approach. In this approach, the available widgets/controls and their look and feel are restricted to a set of widgets that is common to all the styles for the merged platforms. This minimal set may be so restricted that reconfiguration becomes impossible. Problems here are recognized, with key vendors currently moving away from the lowest common denominator approach. The situation is thus improving, and virtual toolkits should in the future obstruct reconfigurabiity less than they originally did.

A typical approach to letting users reconfigure systems is provided by the X Window System Resource Manager. It employs a form of table-driven customization (Scheifler *et al.*, 1992). While X-based applications usually use this at system start up, nothing prevents dynamic use of data from the resource database.

Both end-user and developers' tools allow reconfiguration of many features, e.g. window decorations (e.g. scroll bars, command icons, borders), key bindings, mouse button bindings, default fonts and colors, interpretation of various mouse movements (e.g. focus follows mouse, focus changes on click), maximum time between single clicks for them to represent a double click event, and the contents and representation of window manager commands.

It is an important question whether typical users require such reconfigurability. Furthermore, it is not clear that end-users can use all the tools that developers find straightforward. However, the perspective taken in this chapter is one of possibility, rather than ease of learning. Properties associated with ease of learning were not considered in Chapter 2, and thus the learnability of reconfiguration tools cannot be considered systematically in this chapter.

Usability on the other hand can be considered. Reconfiguration tools often obstruct robustness properties such as predictability, since the terms used to describe attributes of window managers (e.g. scroll bars, borders, key bindings, mouse button bindings, default fonts, focus follows mouse entry, focus changes on clicks) have subtly different meanings in different specific commercial products. The result is that the effect of items on control panels and dialog boxes for window manager reconfiguration may be so hard to predict that users give up trying to get any windowing system working the way they want.

Current window systems thus provide considerable support for reconfigurability, but this support is neither coherent, comprehensive nor compre-

hensible. Features have accumulated in a piecemeal manner, with limited thought for the user's view of reconfiguration.

A tool that supports reconfigurability and need not be part of a window system is a macro recorder. This lets users record sequences of actions (keystrokes, mouse movements, screen touches), name, save and edit sequences, and then play sequences back such that they appear to be coming from the user. Such tools relieve users from having to perform complex, error-prone repetitive tasks. However, reconfiguration here is largely restricted to the dialog level of abstraction.

Macro languages and associated script editors also assist in the provision of reconfigurability, but unlike macro recorders, users must program macros themselves. For example Tcl/Tk (Ousterhout, 1994) is a graphical toolkit (Section 5.4.2 below). With it, users can dynamically change many aspects of widgets during system execution. These changes are programmed using a simple command language.

Compared with tools such as Tcl/Tk, users are better supported by UIMSs with customization features, such as S/X Tools (Küehme and Schneider-Hufschmidt, 1992), which provides widgets with several customization options. In contrast, customization options are rarely found in hand-coded widgets for specific projects.

Construction phase interactions with interaction flexibility properties are summarized in Table 5.4. Most support takes the form of assistance for a few detailed approaches to partial delivery of a property. Most exceptions to this are properties that could be proved during specification, which can thus be delivered during construction. However, one property that could not be addressed during specification can be delivered, but only in a limited form (device multiplicity). Overall, support appears to be patchy, with an unprincipled set of local solutions to the challenge of using design principles to guide software development. However, the table does not include properties that can be preserved from specification through the use of model-based UI tools and UIMS. The use of such tools improves support for interaction flexibility during construction, but only for properties that could be addressed during specification (see Table 5.1).

5.3.2 Robustness Properties

Properties for the robustness principle were largely supported by assessment during specification. Few tools or materials assist during construction. Support here is very specific and rarely provides general support for a property. Still, partial support exists for most properties.

Observability is assisted by all declarative constructs (e.g. view controllers) that assist representation multiplicity, with differing extents of support for each tool or material. Observability can be further assisted by context-sensitive help. Such help can tell users what is currently possible

Table 5.4 *Construction Interactions between Tools/Materials and Interaction Flexibility*

Property	Interaction	Comment
Device Multiplicity	Deliver	Resource Manager, but often restricted to specific drivers in window systems, unless Plug and Play supported
Representation Multiplicity	Assist	By View Controllers (SERPENT), but mostly support from materials (e.g. Model-View Controller (Smalltalk), Multi-View Agents
I/O Re-use	Assist	Inter-Application communication facilities, if compatability problems avoided Object Linking and Embedding
Human-Role Multiplicity	Assist	By groupware toolkits
Multi-threading	Deliver	Resource Manager
Non-preemptiveness	Assist	Resource Manager, but pre-emptiveness can be obstructed
Reachability	Deliver	By re-usable history module (or class)
Reconfigurability	Obstruct	By virtual toolkits, but situation is improving
	Assist	By table-driven software, macro recording, feature modification (e.g. changing menu items) and tools such as Tcl/Tk
Adaptivity and Migratability	Assist	Limited support from materials (e.g. User Modeling Shells, Plug and Play, AgentWare?)

and how to accomplish it, thus making the current state of the user interface observable. When a broad range of context-sensitive help facilities is encapsulated in a re-usable module, then this is a specific form of material that addresses a small part of observability.

Insistence is delivered in one specific form by materials that implement modal dialog boxes in toolkits, and in other materials that implement repeated replay of audio until some user acknowledgement.

Honesty is assisted by all declarative constructs that assist representation multiplicity, and by all materials that improve response time, as users can perceive a system as lying when known 'out-of-date' information stays rendered. It also requires good response times even at the physical level of interaction, where immediate character-by-character feedback during typing is preferred to delayed output. Many aspects of honesty however are given no support, for example the suppression or revision of warnings and error messages that no longer hold (because the monitored condition has changed).

Access control is delivered in a broad form by access control lists that hold information on access to data and commands by user roles. There are materials that provide some basic assistance with access controlability. For example, (the code for) a file system manages read, write, and/or execute permissions. More extensive support can be envisaged, and is provided in part by Suite (Dewan and Shen, 1992). Instead of providing access control in the back-end or persistent store of an application, Suite implements access control in the front-end or user-interface of the application. As a result, it is able to provide earlier feedback to access violations and protect fine-grained operations (such as move cursor) on logical user-interface objects (such as paragraphs) instead of coarse-grained operations on physical objects (such as files). Such support is important in collaborative environments.

An alternative (or complementary) approach is to have appropriate modules form a framework for integrating single-user legacy applications into a multi-user cooperative environment. The framework could route information between applications without any 'knowing' it is being used in a new multi-user environment. The COLA approach developed at AT&T has created a systems programming basis for such a framework. Extensions to standard Unix library functions such as open, read, write and close let these be treated like active values, with other actions being triggered whenever they are called (Krell and Krishnamurthy, 1992).

Predictability is delivered in one specific form by percent-done indicators (Myers, 1985) (and the system is more honest as a result). Response-time stability aspects of predictability can be supported by materials that let developers reduce resource usage (paging managers, hypertext pre-fetch code, dynamic linking and indexing code). Such materials can also improve the *pace tolerance* of the system, by reducing adverse system delays.

Pace tolerance is not just concerned with shortest possible response time of a system. Users also need to control the interaction pace, such as specific capabilities for controling mouse acceleration and setting double-click intervals, as well as operations for designers to insert delays. Such specific support currently comes in the form of materials (i.e. library routines) with limited tools (control panels, resource editors). UIMS generally lack time constructs. For example, RAPID only had a time-out construct (Wasser-

man, 1985). Pace tolerance is also delivered by operations for introducing delays into the interaction. Materials that calculate the time needed to read a message before removing it automatically using Bevan's algorithm (Bevan, 1983) would also make a small contribution to pace tolerance.

However, little attention has been paid to *deviation tolerance* when reading set-up files. The simple database manager can detect misnamed parameters and inappropriate value settings, but there is no provision for error recovery by the system or the user (there are not even notifications of problems).

Some programming languages have fail-safe features (e.g. error handlers in Visual Basic; Microsoft, 1993b). Such features assist deviation tolerance, by providing an infrastructure for implementing error detection and recovery. Further support is provided by materials that implement error recovery from either user or system errors. In the case of system errors, re-usable checkpointing and roll-back code can be used. Many database tools can provide these capabilities at the functional level of an interactive system (e.g. the database capabilities of Visual Basic). The most robust tools maintain their checkpointing logs and repositories independently of the system to ensure they will not be contaminated by system failure. After a system failure, the system state can be set to that of a selected backup, or a selected log can be processed to reach the desired system state.

In the case of user errors, re-usable modules that implement undoing capabilities can be used (Yang, 1988). Some implementation frameworks assist with the provision of this feature by providing basic support for undoing. For example, Command objects in the MacApp framework (Schmucker, 1986) can have undo methods associated with them. However, the developer must construct an inverse for each command to take advantage of this. Even so, this is still assistance with the property of deviation tolerance, despite its very basic and specialized nature.

Suite provides automatic support for (multi-user) undoing of user manipulations of (distributed) active values (Dewan and Choudhary, 1995). Any side-effects taken in response to these modifications by the application must be undone by application-defined undo methods. Thus, the responsibility for undo is divided between the generator and the application with the generator undoing its actions and the application undoing the ones it takes.

Construction phase interactions with robustness properties are summarized in Table 5.5. Compared to flexibility, more support takes the form of delivery, but this is again in the form of local specialized solutions that partially deliver a property. Several properties could be assessed during specification, but little can be done to preserve this during construction, other than by formal transformation methods and for the limited solutions provided by specific materials. However, the table does not include properties that can be preserved from specification through the use of model-based UI tools and UIMS. The use of such tools improves support for interac-

Table 5.5 *Construction Interactions between Tools/Materials and Interaction Robustness*

Property	Interaction	Comment
Observability	Assist	Generally, T/Ms supporting representation multiplicity (e.g. view controllers) support observability Also assisted by context-sensitive help and UIMS with Arch/Slinky architecture
Insistence	Deliver	By very specialized materials (e.g. materials for modal dialog boxes or repeated audio replay)
Honesty	Assist	Generally, T/Ms supporting representation multiplicity, response-time stability and pace tolerance support honesty
Predictability	Deliver	Percent-done code delivers partial and very specialized support (response-time conformance, also achievable by reducing resource usage)
Access Control	Deliver	By access control lists
	Assist	By customized overlays as well as by more basic file system features
Pace Tolerance	Deliver	Delay introducing operations (e.g. for reading messages)
Deviation Tolerance	Assist	By 'clean' dialog abstractions that support processes, by constructs for error recovery such as fail-safe programming language features
	Obstruct	By resource managers that silently ignore errors in configuration files (e.g. X Window System)

tion robustness during construction, but only for properties that could be addressed during specification (see Table 5.2). The table also omits interactions with logging code, as that is largely outside the scope of this chapter. However, the logs produced will highlight adverse patterns of user interaction, especially failures in robustness.

5.3.3 Internal Properties

Much support for internal properties comes from the architectural model, but such properties must be preserved in the final architectural refinements and throughout construction.

Construction tools should always improve *development efficiency*. This can be compromised if the underlying configuration or programming language is not well formed and properly specified. For example, a problem with the concrete syntax for subnet traversal in RAPID forced cumbersome 'fixes' when a subnetwork needed to be traversed from more than one point in a calling network (Cockton, 1985).

Instrumentation code profiles the space and time consumption of executing processes. It delivers some *maintainability* (by highlighting adverse resource usage) and some *evaluability*. Its main value is in its assistance for maintainability, where it helps to discover errors (from failures to meet requirements to system crashes), and to locate the cause of the error. Instrumentation code however does not assist with the key step in maintenance, i.e. correcting the cause of the error. Tools for discovering errors include:

- video and audio recording facilities that capture user activities and commentary;
- quality assurance testing procedures which validate that the system meets its requirements;
- debugging tools for conducting tests and experiments to locate errors;
- performance and resource monitoring tools;
- logging and evaluation tools (tools for evaluability).

Tools for locating the cause of the error may include profilers and testers. Tools for correcting the cause of the error include text editors and specification/programming tools to correct the error and regenerate the appropriate materials.

At the same, instrumentation code does obstruct *run time efficiency*. Thus properties need to be traded-off when selecting tools and materials, just as they had to be when selecting architectural models.

Further support for assessment of internal properties such as maintainability and modifiability comes from the use of inspection techniques. Remaining properties are only assisted. For example, *evaluability* is assisted by materials such as code that logs invocation of each event. Tools that analyse such logs belong to the evaluation phase.

Portability is supported by the use of code supporting layers and wrappers around platform dependent features, or the use of emulators and simulators which let systems coded for one specific hardware and/or software environment execute in another one (this approach may obstruct run time efficiency, but it can 'buy time' for a more thorough conversion of the

application). For new applications, virtual toolkits (Retter *et al.*, 1992) deliver portability.

Run time efficiency is supported by *virtual separation* (Shevlin and Neelamkavil, 1991), where any execution inefficiencies due to separating levels of abstraction at design time can be removed by tight physical integration of the run time code for the user interface and the functional core. Virtual separation in this sense is little more than the specialization of compiler optimizing techniques for separable interactive systems development.

User interface integratability has two distinct aspects. On the one hand, the user interfaces of separate architectural components should be consistent and interoperable. On the other hand, it should be possible to compose separate interactive components into a single system.

Consistency and interoperability for interactive components are supported by materials that implement components described in style guides (Microsoft, 1993a, 1995). User interfaces that use common components will be easier to integrate. Some components cover all levels of abstraction in interaction. For example, Visual Basic's common dialog boxes (see Section 5.4.3) are common functions that have been factored out of the individual logical, dialog and functional levels of their interactive behavior. Visual Basic provides such common dialog boxes along with some underlying functionality. Other construction tools provide support for composing dialog boxes and related functionality. Most X toolkits (e.g. Tcl/Tk; Ousterhout, 1994) supports 'superwidgets' that are such compositions.

'Composability' of interactive components requires basic software support in order to address this aspect of user interface integratability. Basic assistance is provided by materials that implement inter-application communication protocols. Users can use these to share data among these applications. For example, users may cut, copy and paste data between applications. Many object-oriented computing environments now support embedded objects, which are constructs that assist user interface integration. For example, Microsoft's OLE (Williams, 1994) lets users of one application invoke functions provided by another.

Another form of support is found in Field (Reiss, 1990), which addresses the composability aspect of user interface integratability. In Field, every user interface broadcasts events of interest and other user interfaces can register interests in them. For instance, a debugger can broadcast the statement being executed and an editor can receive it and then highlight the current line. This has supported integration in programming environments (by putting minimal wrappers around tools). The method is now in commercial use.

Lastly, one may also regard X pseudo servers (Lauwers and Lantz, 1990) as delivering user interface integratability by integrating multiple instances of the same interface without requiring any changes to the user interface.

Construction phase interactions with internal properties are summar-

Table 5.6 *Construction Interactions between Tools/Materials and Internal Properties*

Property	Interaction	Comment
Development Efficiency	Deliver	Well-designed tools and materials should always deliver this property
System Modifiability	Assess	By inspection techniques, but largely an architectural property
Portability	Assist	By virtual toolkits and more generally by layered wrappers or emulations and simulators
Evaluability	Deliver	Instrumentation code
Maintainability	Deliver	Instrumentation code reveals common problems
	Assist	By Inspection Techniques
Run Time Efficiency	Obstruct	By Instrumentation code, layered wrappers and emulations/simulators, which slow things down
	Assist	By virtual separation, which removes layers at run time
User Interface Integratability	Assist	By standardized (style-guide-based) components and other common components By tools such as Visual Basic (Microsoft) and Tcl/Tk By materials such as inter-application communication facilities and Object Linking and Embedding (Microsoft)

ized in Table 5.6. There are no obvious patterns in the table, other than a wide range of forms of interaction. However, the interactions noted here are clearly only a sample of possible ones, since there are many general software tools and module/class libraries that offer favorable interactions with internal properties. The analysis above has thus highlighted the more novel interactions that are especially relevant when constructing interactive systems.

5.4 Commercial Tools

Three commercial tools are now analysed in depth to further validate and extend the analysis from the previous two sections. They are 'commercial' in the sense that they are either products or have a widespread user base.

There are two main uses for a tool study. On the one hand, a full analysis of properties would be a (complete) tool evaluation, but on the other hand a more restricted analysis can confirm existing and expose further interactions between properties. The restricted analysis can also expose the complex ways in which a tool may interact with properties. All examples in the analyses below are chosen with the second use of a tool study in mind. They are not to be taken as complete and balanced evaluations of each tool. Instead, they reflect the interest of the authors in the utility of the properties and architectural analyses developed in earlier chapters. They are thus more evaluations of the value of property profiles and architectural analysis than summative evaluations of the worth of the three example tools, which are TAE+ (Szczur and Sheppard, 1993), Tcl/Tk (Ousterhout, 1994) and Visual Basic Version 3.0 (Microsoft, 1993b).

External properties are visible to the user of a system. This means that users of the interactive system being developed will be aware of them. Likewise, external properties of the tools used during design and development are visible to developers (as tool users). Tools therefore manifest external properties to developers and more or less support internal properties being designed into the system being developed. Each of the following examples looks at these differing aspects of interaction between tools and properties.

5.4.1 TAE Plus

NASA's Goddard Space Flight Center develops and maintains software to provide for control of all NASA's unmanned spacecraft and for the collection and analysis of the resulting scientific data. The Transportable Applications Environment Plus (TAE+) was designed to handle the development of user interfaces and the run time management of systems in this complex, heterogeneous, distributed computing environment. TAE+ is now distributed commercially by Century Computing.

TAE+ supports the Motif (OSF, 1990) look and feel for a wide variety of platforms. Developers use the TAE+ Workbench to specify the layout and dialog of a user interface. The application's windows are constructed from Motif widgets and presentation types that are combinations of Motif widgets. The Workbench generates a resource file and code to implement the user interface in ANSI C, K&R C, C++ or Ada. Developers add functional core routines to complete the application.

A set of application services, the Window Programming Tools (WPTs), provides run time support, managing the user-interface layout and the

dialog that has been specified in the Workbench. In addition, the user interface may be dynamically updated by the application using a run time interface library. Communication between the Workbench and the run time support system occurs via the resource file.

The discussion below evaluates TAE+ very briefly in terms of the internal properties of Chapter 2 and concentrates in more detail on the external properties of Chapter 3. For the most part the evaluation is done in terms of the TAE+ user (an interface designer) as opposed to the end user of an application developed with TAE+.

Modifiability

TAE+ partitions an application into three distinct parts: layout, link-based dialog, and functional core. Each part may be modified separately. For instance, if only layout changes are made, the application may be restarted with the updated resource file. The system does not need to be rebuilt.

Similarly, program code may be attached to the link-based dialog within the Workbench. This code may be modified externally and the changes will be maintained when the system is regenerated from the Workbench.

Portability

TAE+ supports the following platforms: Sun (SunOS and Solaris), Hewlett-Packard 9000 series (HPUX), Silicon Graphics (IRIX), IBM RS/6000 (AIX), Concurrent RT-7000 (RTU), Intel 486-based (SCO Unix and Linux), DEC station (ULTRIX and OSF/1), and DEC VAX (VMS). A user interface generated for one platform is completely transportable to any other.

Evaluability

Two tools help TAE+ usability engineers evaluate the 'goodness' of TAE-produced end-user application interfaces. An adjunct tool, CHIMES, can be used by the user-interface designer to check consistency across windows (e.g. placement of objects) and compliance of layout with usability guidelines (such as number of colors and type fonts). A second (adjunct) tool, the User Action Graphing Effort, uses TAE's Perl scripting capability to capture data used to compare the actions (keystrokes, mouse clicks) taken by a novice in performing a task to those of an expert doing the same task. A graphical display of the actions of the two users and a time-stamping capability enable a usability engineer to identify features of the user interface that need to be made easier to use. TAE+ support for rapid prototyping further improves evaluability when combined with co-operative evaluation (Monk *et al.*, 1993).

Maintainability

The scripting facility described above can be used to develop test scripts and an automatic application test suite. TAE+ thus supports maintainability (and modifiability) by addressing *regression testing* (the repetition of tests passed by code before it was changed). Successful maintenance requires that changed code should pass these tests again.

Should user problems indicate a need to change displays, then layout changes can be made quickly and take effect without rebuilding the system.

Dialog changes that affect only the resource file may also be accomplished without rebuilding the system. TAE+ lets developers make run time changes in the user interface through library calls from the functional core. While this feature extends the range of interfaces that can be developed, it negates the separation of the functional core from the user interface. This may result in increased maintenance costs.

Run time Efficiency

The run time efficiency of TAE+ is dependent on the operating environment. Applications running locally on up-to-date workstations seem 'fast enough'. As with all systems, network delays or out-of-date hardware can cause problems.

User Interface Integratability

TAE+ generates applications with the Motif 'look'. Motif operation guidelines are enforced to the extent that standard widgets (e.g. radio buttons) are used, but other guidelines, such as menu structure, are not enforced. TAE+ provides the flexibility to model a user interface on existing ones, but this flexibility leaves the designer with the responsibility for compliance on style issues. Local standards can be supported because designers can modify the Workbench, changing the set of widgets that are available and customizing property defaults (color, widgets, etc.).

Functional Completeness

TAE+ can be used to create interfaces that look and feel like those created with Motif-based tools. Additionally, TAE+ implements a number of widgets designed for use in control panels. These include: dynamic text whose color and text string are dependent on threshold values of an attribute, a strip chart, a rotator for circular gauges, a discrete widget that displays unique pictures for a finite number of attribute values, and a mover that animates a defined area of a picture in response to changes in attribute values. These widgets can be activated by user inputs as well as internally-generated data.

These somewhat esoteric widgets are essential for application programs

at NASA Goddard Flight Centre. Basic toolkits such as Motif do not provide this functionality, so these TAE+ extensions make it possible to achieve functional completeness.

For the developer, there are some remaining inelegancies that obstruct functional completeness (e.g. support at run time, but not in the Workbench for geometry management and sub-panels of Dialog Boxes etc., see below).

There are several issues associated with the current release, Version 5.3, of TAE+ in terms of functional completeness. Firstly, the widgets are not all simple Motif widgets; some are a combination of Motif widgets and the Dynamic Data Objects that are unique to TAE+. Although the developer can code at the Motif level, it is currently not simple to do so. It is difficult to add widgets because such additions require modifications to both the Workbench and the API.

Secondly, TAE+ does not support geometry management. Therefore, widgets such as the Motif RowColumn widget can only be used by declaring an X Window workspace in TAE+. (This workspace is not managed by TAE+, but by making windowing system function calls.) Third, the Workbench does not allow a designer to create a panel contained inside of another sub-panel. The run time library supports subpanels – there are plans to eliminate these shortcomings in the next release.

Development Efficiency

The TAE API has been shown to be efficient in terms of learning and coding time because the Window Programming Tools operate at a high-level of abstraction. Further development efficiency results from support in the TAE+ Workbench for object re-use through copy and modify.

TAE+ provides a Rehearse function that permits the designer to prototype the user interface and provide clients with an operational prototype without coding. The prototype may be used for design reviews and successive refinement of the interface before commitment to a final design. This has been shown to reduce overall development time. There are several features that contribute here, for example, being able to make layout changes without rebuilding the system. However, if user needs must be addressed by adding new widgets, then development becomes less efficient (see user interface integratability, above).

Flexibility and Robustness

TAE+ supports many internal properties. External properties are also supported for developers (in the Workbench) and for the user (in generated systems). Thus, end user applications can be developed with TAE+ to meet many of the criteria related to flexibility and robustness, although there is

no explicit CSCW support for the properties of *human role multiplicity* and *access control.*

Two flexibility properties associated with planning of user actions (i.e. *reachability*, *multi-threading*) and robustness properties associated with the current state of the system (i.e. *observability*, *insistence* and *honesty*) can be addressed for those aspects of the dialog that are configured as TAE+ *link-based dialogs.*

Links (connections in TAE documents) are a type of event-response rule. The events are limited to things like selections and field completions. Responses can be to alter a panel (window, dialog box) attribute such as visibility and/or to initiate a call-back. They let designers define simple dialogs without writing code, such as popping up windows/dialog boxes or closing them.

Theoretically the link-based dialog can be analysed for *reachability* and *observability*. As long as the designer uses the link-based dialog, it is possible to check that there is a path which will display all relevant data. However, each event that can have a link associated with it can also generate a call-back to the functional core, or perform (hidden) dialog actions. As most applications require some use of the call-back mechanism to complete their dialog definition, this also means that automatic analysis would be incomplete without analysing the code – a nearly impossible task. Assessment of *reachability* and *observability* in the final system is thus only supported for the exclusive use of link-based dialogs and data-driven objects. The latter let designers easily build an object that changes state when the value of a monitored variable changes. For example, a numeric output can be displayed with three colors: red for out-of-range error, yellow for near out-of-range, and black for in-range. *Insistence* is thus addressed by this construct.

TAE+ supports interruptible behavior in normal operation. *Pre-emption* by the functional core is possible (e.g. error conditions). However, functional core initiated states are not represented in the link-based dialog. Thus, while updates initiated asynchronously by the functional core address the temporal requirements for *honesty*, they further obstruct the analysis of *reachability*. The mix of positive and negative interactions is a good example of the complexity that can arise in property-oriented analyses of tools.

TAE+ currently supports keyboard and mouse input and has been instrumented additionally for speech recognition and synthesis, and thus provides moderate *device multiplicity*. *Representation multiplicity* is achieved with a robust library of objects for representing information, both textual and graphical, and additional objects can be created. For *I/O re-use*, the Workbench supports cut/copy/paste of all objects (automatically renaming them). Only the X Window standard cut/copy/paste are supported in applications.

Reconfigurability can be supported (e.g. in the form of feature modifications) by calls from the functional core to the run time user interface. As the Workbench itself is a TAE+ application it can be readily reconfigured by developers.

The general experience with favorable run time efficiency extends to *pace tolerance*.

Deviation tolerance is given basic support for 1-level undoing.* TAE+ supports restricting the range of numeric fields, but generates an out-of-range error at run time, which could obstruct deviation tolerance.

5.4.2 Tcl/Tk

The interface building tool Tcl/Tk consists of a programming language, Tcl (Tool command language) together with its associated X Window toolkit, Tk. Being a full programming language, Tcl itself is inherently neutral with respect to the external properties proposed in Chapter 2. However, using Tcl together with the interface building facilities provided by Tk, it is possible to build complete applications which provide any desired combination of properties. Alternatively, Tcl/Tk can be used to develop tools with which a user may build systems. With this approach system-building tools can be produced which guarantee a particular set of properties for any resultant system. While any of the external properties could be delivered in such systems, the features and facilities provided by Tcl/Tk, particularly those provided to manipulate the Tk widgets, affect the ease with which the developer might achieve certain properties. Overall, tools such as Tcl/Tk do not address as many properties as systematically as do sophisticated tools such as TAE+. The main consequence is a loss of development efficiency for the more demanding aspects of user interface design.

As with other X Window toolkits, *device multiplicity* in Tcl/Tk is restricted to the use of display, mouse and keyboard. *Representation multiplicity* is facilitated by the provision, within Tk, of a variety of basic widgets. As both Tcl and Tk are designed to be extensible, these widgets can be combined or extended, and more complex widgets created. Thus tools or applications can be produced which deliver the required level of representation multiplicity. The selection retrieval mechanism associated with Tk widgets simplifies the delivery of basic *I/O re-use* such as 'cut', 'copy' and 'paste'. In addition any input/output re-use at the physical or functional levels can be delivered with some programming effort. Although Tk provides no explicit support for *human role multiplicity*, several Tcl/Tk applications have been built which deliver this property, including database systems which support the differing roles of Data Manager, Data Provider

* In 1-level undoing, only the last change can be undone, so an undo followed by an undo undoes the undo.

and Data User, and computer assisted learning systems which distinguish between the roles of teacher and student (Newman and Smith, 1995).

Reconfigurability is facilitated for the developer by the ability to set or alter key bindings, mouse button bindings, mouse movement interpretation and defaults for font and colors that can be set for each object class in Tcl (the latter capability also addresses the internal properties of *maintainability* and *modifiability*). Applications or tools can then be built which allow the user to customize any of these features. Tcl/Tk is neutral with respect to the properties of *reachability* and *non-preemptiveness*, while *adaptivity* and *migratability* could only be delivered with considerable programming effort.

Robustness properties are largely design and specification issues, hence construction tools interact with these properties less than do specification tools. Nevertheless, Tcl/Tk provides some features which may support the delivery of some of these properties. Firstly, the availability of modal dialog boxes, flashing icons and window 'grabs' can assist in the delivery of *insistence*. Secondly, the ability to disable and 'grey out' buttons or menu items, and the ability to change the cursor according to the user's context can contribute to *honesty* and *predictability*.

5.4.3 Visual Basic Version 3.0

Visual Basic is the name given by Microsoft Corporation to its development environment for a version of the Basic programming language that exploits capabilities of their Windows operating systems.

Visual Basic determines the appearance and behavior of a user interface in two ways:

- Some features are determined by specifying them interactively.
- Other features are set during the execution of the Basic program.

Visual Basic's documentation calls these *design-time* and *run time* settings respectively. The values that can be set at design-time and run time are not identical, but there is considerable overlap.

Visual Basic combines features of construction and execution tools. The development environment supports design-time creation of *forms*, which can be used as dialog boxes, document windows or application windows.

Controls (the widgets of user interface toolkits) can be placed on forms. There are controls for text entry, value entry (sliders and spin-boxes), value selection (list and check boxes, option buttons) and command initiation (command buttons, drop-down and pop-up menus). Controls have attributes that affect their appearance and behavior.

Attribute values can be set at design-time. Text labels and icons are treated like controls, which lets their attributes be set at design-time.

At run time, controls respond to a fixed set of input and system events.

Handlers for these events are programmed in Basic. Other Basic procedures can be written, and these can be called from event handlers. These may interrogate and alter the values of attributes as required.

There are many capabilities in both the design and run time environments. The analysis of properties which is given here merely addresses some of the most important of them in relation to the architectural model described.

Examples have been chosen to illustrate three uses for tool studies: tool evaluation; to confirm existing and expose further interactions; to expose the complex ways in which a tool may interact with properties. The analysis below is based on reports from a few of the authors about their experiences in using Visual Basic version 3.0, supplemented by an extensive study of the generally candid programmer's guide (Microsoft, 1993b). All page references below of the form (VBPG xxx) refer to this guide.

Flexibility Properties

There is a clear pattern in the support offered by Visual Basic for flexibility properties, since its run time architecture largely addresses the logical interaction component. It thus lacks most of the functional partitions adopted for architectural analysis introduced in Chapter 4. Since properties that concern flexible planning of interaction depend heavily on dialog functions, the absence of a dialog component affects support. Similarly, most properties that concern flexible representation of information rely on several functional partitions. The lack of clear dialog and functional core adapter components means that such properties cannot be systematically addressed (since they involve interactions via the dialog between the logical interaction and the functional core adapter).

Support for flexible interaction planning is thus largely restricted to logical interaction features. The underlying event model makes it very easy to write modeless interfaces, and thus *multi-threading* and *non-preemptiveness* are assisted. However, non-preemptiveness is easily obstructed by poorly written applications.* To achieve full *user-oriented* non-preemptiveness for all applications running in a Windows environment, every application has to regularly surrender control via a *DoEvents* call (VBPG 417). This reveals the lack of process or equivalent constructs. In terms of the interactions between tools and properties introduced earlier in this chapter, multi-threading is assisted rather than addressed, since process constructs have to be built on top of basic events. There is thus a strong risk that extensive multi-threading will not be achieved when developing with Visual Basic.

* The user-oriented use of *non-preemptiveness* is potentially confusing here, as it is used from the user's point of view, whereas in operating systems it is the currently active process that is **not** preempted in a non-preemptive environment.

The control constructs of Visual Basic are restricted to event handlers and procedure calls. This restriction results in a monolithic run time architecture, with no modularisation of processes or threads. The analysis of *reachability* is effectively obstructed by the lack of a central dialog abstraction to analyse. Furthermore, the lack of a functional core adapter rules out coarse-grained reachability analysis. It is thus almost inevitable that proofs of reachability will not be attempted when developing with Visual Basic, unless separate dialog specifications are either prepared in advance or reverse-engineered from the code. The latter approach is difficult and error prone.

The lack of a well structured architecture restricts support for representation multiplicity to piecemeal provision of several specific capabilities. Thus, for example, there are several date formats (VBPG 162) and the icon displayed during dragging can be changed (VBPG 279). More generally, semantic feedback during dragging is greatly assisted by the provision of enter and leave events (as discussed in Chapter 4, page 113). Even so, there is no generalized support for simple user interface animation, which is often obstructed by the very primitive event timing in Visual Basic's kernel. An increasingly common form of user interface representation is thus not well supported.

Support for representation multiplicity covers a broad but uneven spectrum. The lack of a complete software architecture for interactive systems forces compensation to take place as extension to the facilities of Visual Basic – but outside it (VBX files: VBPG 123). VBX files provide a way to add new controls (e.g. graphical command buttons with a 3D look and feel). This (rather indirect) assistance for representation multiplicity has led to a proliferation of third party controls. However, development of new controls within Visual Basic itself is difficult, as there are many graphics primitives and attributes that can only be created/set at run time (e.g. graphics methods for arcs and setting pixels (VBPG 339)).

Representation multiplicity is thus addressed for a few presentation features, but is at best assisted and may be obstructed. VBX files let missing features be added, but they do not let unsuitable ones be fixed. Representation multiplicity is restricted in the Multiple Document Interface, as Document Windows (but not dialog boxes) must go inside the parent application window (VBPG 297). Extensions that overcame this restriction would have to re-implement a major part of Visual Basic itself. Interestingly, the Windows 95 user interface has preserved the Multiple Document Interface – reluctantly, as it appears from the style guide (Microsoft, 1995). This strongly suggests that re-programming the Multiple Document Interface requires resources beyond those available to most application and tool developers.

Support for *reconfigurability* is largely similar to that for representation multiplicity. Features such as multinational data formats address both

properties, but assistance, neutrality or obstruction are more common interactions between Visual Basic and flexibility properties. Very low-level language features assist with reconfiguration (e.g. arrays of controls let controls be added and removed at run time). However, some features can only be set at design time. This obstructs reconfigurability by ruling out run time changes. For example, the multi-line text property and scroll bar properties can only be set at design time (VBPG 40). Also elements of control arrays that were created at design time cannot be removed at run time (VBPG 71). Such *ad hoc* boundaries between design and run time have a negative impact on other properties (see below).

Support for other flexibility properties is limited. *Device multiplicity* is obstructed by the absence of multi-media support. *Human role multiplicity* is not addressed in any way (interactions are thus neutral). However, there is some useful basic assistance for *migratability*, as keystrokes can be passed on to other applications, so agents could be implemented that take responsibility for some tasks. As the receiving application cannot distinguish between user- and application-generated keystrokes (VBPG 521), a requirement for migratability identified in Chapter 3 (page 83) appears to be satisfied, but this requirement is user- rather than system-oriented. In fact, commands that have been migrated do not require presentation (i.e. activating main application window, popping up dialog boxes). However, this will happen when migration is driven at the logical interaction level. Properties such as *pace tolerance* and *honesty* will clearly be obstructed by this approach to migration.

Lastly, *I/O re-use* is given basic assistance by clipboard capabilities (VBPG 405) and the ability to pass on keystrokes to other applications (VBPG 521).

Robustness Properties

As with flexibility properties, robustness properties that depend on several architectural components are not well supported. The lack of clear dialog and functional core adapter components means that *observability*, *insistence* and *honesty* cannot be systematically addressed (since each relates to interactions via the dialog between the logical interaction and the functional core adapter). The result is that interactions with these properties are generally neutral.

The remaining robustness properties of predictability, access control, pace tolerance and deviation tolerance are less dependent on extensive architectural support. Even so, Visual Basic provides limited support.

Access control fares relatively well. Database features address it with the capability to restrict read or write access to data items (VBPG 461). However, file operations provide no such support.

Pace tolerance is generally obstructed. At the logical interaction level,

mouse move events may not be generated for each pixel (VBPG 269). This will cause problems for some fine sketching, drawing, dragging and region selection tasks, since Visual Basic may not be able to keep up with the user. A more general problem arises when Microsoft's DLL (Dynamic Link Library) mechanism is used for integration, because the default time-out for data link accesses is five seconds (VBPG 503)! This suggests that such delays are to be expected. DLL access to functional core values during closed-loop interactions, such as slider manipulation during star field queries (Ahlberg and Schneiderman, 1994) will thus result in pace tolerance problems.

Deviation tolerance is given better support, since the database rollback methods provide some assistance with error recovery (VBPG 478). Similar assistance with error detection is provided by the Data Error event (VBPG 475). However, this focused support for error handling is not matched by non-database features. Visual Basic has an 'on error' construct (VBPG 238), but the assistance provided by this and related constructs – resume construct, null return values (VBPG 164) – are too general to provide effective support for deviation tolerance.

Internal Properties

All things being equal, the design-time capabilities result in *high development efficiency*, especially for systems where the functional core is little more than a database. For example, there is a unified SQL interface for all database systems which are supported (VBPG 483). Other capabilities greatly accelerate the development of a few specific functions. For example, the grid control manages rows and columns for spreadsheet and other tabular presentations. The text box, check box, label, image and picture box controls can all be *bound* to database items (VBPG 462), automating the implementation of dialog links between values in the functional core and the logical interaction. There is also extensive support for later life-cycle activities such as installation (VBPG 573).

When these focused features such as SQL interfaces, grid controls and active data values are inadequate, development efficiency is reduced whenever a key external property is inadequately supported. This problem may be alleviated if there is compensation from third party shareware or commercial custom controls. Thus, development efficiency is reduced when 'graphics with semantic content' (in window graphics) are called for, as these must be written from scratch – jeopardizing *functional completeness* unless appropriate custom controls can be purchased. Where complex dialogs are required, this can easily result in large amounts of spaghetti code, reducing development efficiency and *maintainability*, as well as risking functional completeness due to errors on dialog logic.

Development efficiency is further reduced when run time capabilities are

inaccessible at design time. For example, graphics methods (the procedures called to produce graphics) have more extensive capabilities than design-time graphical controls. Similarly, rapid prototyping is obstructed by the inability to place text in grid cells at design-time, since mock-ups of possible tabular displays must be programmed rather than specified interactively.

Lastly, some language abstractions are too low-level to allow rapid development: items must be added to lists one at a time (VBPG 52), bit fields are used to represent mouse and keyboard status (VBPG 271), and explicit indices are needed to set the 'tab order' for the controls for a form (VBPG 66 – third party tools do support more direct specification of this order at design time).

Maintainability and *modifiability* are addressed by a range of general software techniques: modules (VBPG 126), objects (VBPG 181), generic objects (VBPG 132), and public and private procedures (VBPG 132). Specific Visual Basic features also address modifiability. VBX files have already been mentioned, as have control arrays, which ease modification of the set of controls on dynamic forms. However, some arbitrary restrictions limit the effectiveness of some of the constructs: objects cannot be placed in huge arrays (VBPG 175) or in user-defined types (VBPG 183). More generally, maintainability and modifiability are also hindered because the code is spread out in many procedures for many objects, and it is difficult to have an updated overview of the code in a development.

Features that directly address *run time efficiency* place minimal demands on programmers. For example, bitmaps can be compressed using run-length encoding, which saves storage (VBPG 262) and image box controls allow efficient display of images that do not require the full functionality of picture controls. However, the advice in the chapter on run time efficiency (VBPG Chapter 11) is somewhat piecemeal and does place considerable demands on programmers' memories. Such 'tips and tricks' approaches to run time efficiency must have a negative impact on development efficiency. Furthermore, there are some inefficiencies for which no work-arounds are suggested. For example, it can take 'several seconds' to create an OLE object (VBPG 529), which will be unacceptable in many interactive applications.

User interface integratability is well addressed in Visual Basic, since the Windows environment has directly addressed this property in its provision of DLLs (VBPG 493) and OLE. However, Visual Basic places some limitations on OLE parameters that could limit either user interface integratability or development efficiency (VBPG 554).

Support for *I/O re-use* is also relevant to user interface integratability (clipboard: VBPG 405; sending keystrokes: VBPG 521), and standardization of Windows features supports user interface integratability. Visual Basic provides implementations of common dialogs (open, save as, print, color, font) in the CMDIALOG VBX file (VBPG 103 and 114), although interestingly there are times when Visual Basic 3.0 does not enforce standards

in the Windows 3.x style guide (Microsoft, 1993a). For example, titles for dialog boxes are not required (VBPG 97).

Functional completeness covers the ability to provide functionality at all levels of abstraction required for a system's adopted tasks. Where abstract commands at the functional level of abstraction are largely operations on databases, functional completeness can be readily achieved. There are extensive constructs for information systems (VBPG 453); images can be stored in the database (VBPG 466). There are, however, several features that introduce the risk of losing functional completeness. For example, tasks that require accurate color presentation are obstructed by the use of internal logical palette and system palettes that will produce the 'nearest' match to a color (VBPG 374). This may not be good enough for many applications (not only desk top publishing and image processing, but also Internet applications such as information servers and tele-shopping), even though Windows 3.x itself has extensive palette functions. Some language and environment features can also jeopardize robustness. For example, the DoEvents function must be called to achieve multi-threading, but care must be taken that the procedure which calls it is not called again before the first call returns. If it is called again, then a stack overflow will result (VBPG 417). However unlikely this is, it remains a burden and concern for programmers that would not exist were true multi-threading constructs provided.

The main value of the above analysis is in confirming software architecture as a key determinant of support for properties when developing interactive applications. More classic interactions, e.g. the simplicity–power trade-off between development efficiency and functional completeness, are also exposed by several examples. Functional incompleteness often appears to have been tackled with a local fix that has resulted in inconsistencies between design-time and run time capabilities. The same is true of the equally classic simplicity–efficiency trade-off between development efficiency and run time efficiency, where a chapter of tips and tricks lengthens the developer's coding agenda.

Any global summative evaluation of Visual Basic based on the above property analysis would be misleading. It is hard to trade-off poor support for external properties against, for example, its extensive installation support. There are also clearly development projects where Visual Basic's design-time environment has delivered extensive development efficiency without compromising functional completeness. Even so, it would be a surprise if these developments had particularly adventurous user interfaces, since key external properties are not well supported by Visual Basic.

As with most 'commercial tools', the driving forces in the market concern are internal properties that are foremost in the developer's mind, since all benefits here accrue to the developer. Improvements in external properties

usually accrue to the end-user, with extra costs for the developer that may not be recoverable. This balance of provision for external and internal properties is evident in the site reports in the next section.

5.5 Experiences at Research and Development Sites

The analysis of tools and materials will now be completed by considering broader experiences at four sites, three in Europe, and one in North America. The two development sites develop business critical hardware and software for internal usage and for sale as products. The two research sites develop state-of-the-art prototypes for both internal and external clients.

5.5.1 Development Work at a Large Systems Manufacturer and Integrator

Nature of interfaces

The nature of interfaces designed at this site is not homogeneous: very different kinds of applications are developed. They extend from legacy applications (including the administration of operating systems) to work-flow systems (based on imaging), new PC-based tools, and client–server applications. Thus, some applications have been on the market for many years and now have a large customer base. These applications continue to evolve. Other applications are only bespoke (for a single customer) and may have a relatively short operational period.

Requirements for legacy applications are less demanding with respect to end-user interaction, but complex with respect to *functional completeness*. Requirements from a few other applications, however, have very sophisticated and specific demands concerning user interfaces.

Materials

In such an established development environment, well known, state-of-the-art software engineering materials are used for activities such as problem analysis and requirements/system specification. GUI-related issues have also been addressed. Specific to GUIs are the provision of style guides and of vocabularies for applications, and the integration of 'heuristic evaluations' and walkthroughs into design and quality assurance processes. These address *user interface integratability* and support assessment of robustness properties such as *observability* and *honesty*. For the more specific demands of some applications, the services of a usability laboratory provided by a related research department are available, but are currently given limited use.

Tools

Given the different kinds of applications, a great variety of tools are in use, although evaluation tools are not in noticeable use.

A significantly large number of applications have to run on two target platforms (e.g. Unix and Microsoft Windows) simultaneously. There is a proprietary tool for specification and construction of GUIs which is tailored to this requirement of platform multiplicity. This tool, called DialogBuilder (Siemens Nixdorf, 1994), is used predominantly for most interfaces to new and to legacy applications.

Tools available on the market are used according to individual project needs (target platforms and software to be included, e.g. by other development partners). Visual Basic is used for those applications which have only Microsoft Windows as a target platform and which have demanding interface features not covered by DialogBuilder. In the rare cases when applications require the support of very different platforms, the XVT package (XVT, 1991) is used to address *user interface integratability*. Some interfaces are based on a proprietary alternative to XVT which addressed proprietary legacy GUI platforms very efficiently, especially with respect to run time efficiency.

This site maintains legacy applications originally equipped with complex and rather sophisticated forms-based interfaces that support *reconfigurability* and *access control*. In order to provide GUI versions that are *functionally complete* compared to state-of-the-art interfaces, there is tool support for transforming the original forms definition files into GUI definition files. The resulting GUIs can be reworked manually (if necessary) by the standard tool, DialogBuilder. Such tools improve *development efficiency*.

The deciding factors for selecting tools at this site clearly concern the required multiplicity of platforms. The main goal is reducing *costs for development and maintenance*. Predominant are applications requiring only modest and standardized interface features (e.g. as covered by style guides, allowing for the construction of interfaces by specification). For these applications, *multi-threading* seems to be sufficiently supported by window managers, i.e. between rather than within applications.

Modifiability is often inherited from the original forms-based interfaces although restricted to a pre-defined scope of interface features (e.g. language, novice vs expert). Special requirements have to be met with respect to learnability, and with the coexistence of legacy interface variants with GUIs.

5.5.2 A Campus Research Centre

This site is using Tcl/Tk in conjunction with the TIMES Distributed System Builder (Smith and Newman, 1995) as a 'Rapid Delivery' (RAD)

vehicle for both 'stand-alone' and distributed information systems. Provided that the necessary problem analysis is available with which to 'prime' a system, TIMES and Tcl/Tk can be used to provide rapid implementations of full working systems with approximately a couple of days of effort for a stand-alone system and about a week for a distributed system.

The basic system consists of combinations of 'front-end subsystems' (denoted FESS) and instances of information management subsystems (IMSS). The FESS are mostly written in Tcl/Tk although other available front-ends are also used, for example, browsing tools for the World Wide Web such as Mosaic (Dougherty *et al.*, 1994)) and Netscape (Pfaffenberger, 1995). The IMSS are usually TIMES systems (a program written in C with an attached database) but could also be simple files or commercial data management systems. A complete system consists of at least one FESS with at least one IMSS (Smith and Parks, 1995).

Several link constructs are supported for interconnection between the subsystems. They can be accomplished by the front-end subsystem directly reading the file in which the data is stored; more 'sophisticated' alternatives include socket connections, pipes, e-mail and intermediate files.

As described in Section 5.4.2, Tcl/Tk can be used as an effective tool for building systems which deliver a number of the external properties detailed in Chapter 2. However, the use of TIMES as an IMSS considerably extends the degree to which these external properties (and the internal properties proposed in Chapter 3) can be delivered while also reducing the amount of programmer effort required, thus improving both the quality of the user interfaces and *development efficiency*.

Most of the properties are achieved by adopting suitable design goals and then by ensuring that the delivered system meets the design. The design of the tools assists in achieving effective implementations quickly. For instance, interfaces are designed which let users make use of appropriate representations on appropriate devices for both input and output, and thus provide *device* and *representation multiplicity*. Tcl/Tk assists with *representation multiplicity* by providing appropriate widgets. The TIMES IMSS assists in the acceptance of a variety of input formats and the provision of alternative output formats by providing a rich set of translation/parse/search capabilities that can be accessed from the FESS.

Non-preemptiveness is a goal which can be achieved by always allowing the user a choice of actions in the design (the tools have no direct influence on this except that they do not force pre-emptiveness). Similarly, *multi-threading* is not specifically prohibited by the tools and large scale multi-threading has been chosen as an explicit design decision in the interfaces built. In a particular situation a user can choose to browse or carry out a 'what-if' investigation then return to the situation they were in and continue. However, at most points they only have one window visible and

must deliberately leave this to continue. Multi-threading is thus assisted by a stack rather than directly addressed by switchable threads.

The IMSS also assists greatly in providing *observability* and *honesty* by making it easy and quick to perform various tasks: locating and retrieving information that has been stored; changing storage representations without losing existing information; and selecting subsets of existing information. *Access control* is facilitated by requiring an explicit 'publication' of information before it can be observed. The publication mechanism is role-oriented, and thus addresses *human role multiplicity*. Information is only available for use by users acting in an appropriate role. Information can still be made 'publicly' available by creating a 'public search' role which is given automatically to all users of the system. The ability to quickly add new storage and indexing capabilities (both statically, by changing the FESS, and dynamically on user request in the TIMES IMSS) means that *modifiability*, *migratability* and *reconfigurability* can be readily supported where required. This same ability means that it is easy to record enquiries and to construct a 'frequently asked queries' facility with the corresponding 'frequently required answers', a very good example of *I/O re-use*.

By design, *reachability* has been approached in a rather unusual way. The TIMES IMSS will not permit deletions, thus it is not possible to remove history and all previous states of the systems are *observable*. Conversely, no state of the complete system can be reached which would negate history. However, it is always possible to create a subsystem containing only part of the information in the existing system. The IMSS provides assistance in ensuring that all such 'viewpoints' are internally self-consistent. Having defined a new viewpoint that does not contain the record of a particular event or does not contain some particular pieces of information, it is then possible to carry out 'what-if' scenarios using this as a starting point. The results can then be compared with other subsets without any possibility of overall system inconsistency arising.

The TIMES IMSS has been designed to be *portable* and already runs on most Unix platforms plus MAC and MS/DOS PCs. The ability to use a variety of communications tools also means that a system can be configured to make use of existing facilities without, necessarily, needing to port the IMSS and the possibility of migrating functionality from the FESS to the IMSS means that a lightweight 'native' FESS (for example Xterm, e-mail tool or WWW browser) can be used, greatly enhancing portability.

Evaluability is a major design goal, and both the IMSS and the FESS building tools have been produced with this in mind. The GENIE system (the UK Global Environmental Change Data Network Facility; Newman *et al.*, 1995) developed at this site has been configured both to allow users to supply comments and to record interactions. This permits the actual usage of the system to be reviewed at regular intervals, providing both usage

statistics and the ability to identify problems with the interface and with the user's understanding of the functions provided.

The rapid development concept means that *development efficiency* is not a major issue. However, *run time efficiency*, which is often sacrificed for rapid development, is important. The IMSS is designed to facilitate the achievement of rapid response with low computing resource use. In addition, the ability to configure an appropriate distributed system allows existing subsystems to be re-used, where this would minimize resource usage. Tcl/Tk, being interpreted, is not particularly efficient. However, if resource consumption problems or excessively slow performance are observed, it is possible to migrate the computing requirements to the IMSS.

5.5.3 Research Centre for Large Engineering Company

Apart from normal programming languages and many standard utilities, this site uses a number of tools for testing purposes and for prototyping purposes. A few of these are briefly reviewed in order of priority.

Visual Basic

This tool for PC software running under Microsoft Windows is used because of its often high *development efficiency*. The proliferation of third party shareware and commercial custom controls increases the chance of *functional completeness*. Visual Basic provides excellent support for normal user interfaces with standard graphics. Database access is simple. It is very easy to write modeless 'direct manipulation' (of the interface) interfaces due to its underlying event model. However, as noted in Section 5.4.3, there are problems with graphics with 'semantic content', support for animation and dialog control. At this site, these shortcomings have all impacted *development efficiency*, *maintainability*, and especially *functional completeness*, by forcing changes to prototypes because design decisions could not be implemented.

Toolbook

Toolbook is another PC tool for applications running under Microsoft Windows. For many applications it delivers good *development efficiency*, and *representation multiplicity* is addressed by specific support for animations and multimedia. Dialog control is scattered onto localized scripts and thus Toolbook suffers from similar problems as Visual Basic here (e.g. it obstructs analysis of *reachability*, obstructs provision of *pre-emptiveness* when required by application domain or user expertise, and obstructs *maintainability* and *modifiability*).

StartView

This is an Interface Builder set for PCs running under IBM's OS/2. The tool is primarily used for its CUA conformity aiming at *user interface integratability*, but it also delivers good *development efficiency*, making it suitable as a rapid prototyping tool.

5.5.4 A Telecommunications Company

Many of the interactive systems developed at this site have real time requirements and very often deal with very large systems (customer databases running into scores of million of records). Several of them are front-ends to larger systems and often employ standard GUI packages (e.g. Motif). Others deal with network concentrators or reconfigure problematic digital switches. This site largely serves two sets of customers:

- internal: large group of heavy users who exercise discretion and can also get the best out of challenging tools and materials;
- external: larger group, who generally must work with 'industry standards'.

Non-preemptiveness is often felt to be necessary since craft (operators) are not expected to be sophisticated users of the system – and can often be the 'front line'. *Insistence*, *predictability* and *pace tolerance* are all considered vital for these and other users.

Performance monitoring tools are heavily depended on – this can make or break a product. In both dealing with customers (scores of millions) and calls (120 million/day), this site is constantly confronted with problems of 'scale'. While several systems are of course broken down into smaller pieces, there are systems that have to deal with a large number of entities. Hence the critical role of performance monitoring to ensure *pace tolerance* and *run time efficiency*.

Portability is both vital and largely assured thanks to Unix. Every size of Unix box is deployed at virtually all levels. Thus tools work in many places. The organization is too large to establish which tools are in regular use. Common tools and materials include: Unix, Unix tools, the X Window system, OpenWin (the organizational standard for *user interface integratability*), and Motif (some developers' preference). Several UI builders are in use, but there are too many tiny and big UI builders throughout the organization to get any coherent sense of current trends.

The site is largely tool- and process-driven in approach. Tools at times take precedence over process, but in the milieu of large scale software construction (not just interactive systems) process plays a very strong role. A standard deployment cycle is followed that is very similar to the one presented in Chapter 1. There are on-line methodologies, which are heavily consulted.

Evaluation and re-evaluation are performed at various stages. The local development methodology dictates a variety of things and has an impact on other things (choice of tools for example), largely by guidelines of the form: what to do when X happens during stage Y of a project development.

5.6 Conclusions

Tools and materials for developing interactive systems is a vast topic. Analysis here has been restricted only to tools and materials that have a positive or negative interaction with an external or internal property, and ones that are used or produced during the specification and construction phases of development.

Various forms of interaction arise between tools/materials and properties. These forms have direct implications for development activities, determining the extent of work and expertise required from developers, and the phases of development when properties can be considered. The most favorable interaction is *delivery*, which can occur during specification and require little expertise and no further effort in later phases. Less favorable interactions are *proof*, *addressing* and *assessing*, which tend to only occur during specification, require considerable effort and expertise, and require further attention to properties in later phases. These variations in attractiveness are reflected in the site and tool reports (none of which mention proofs), where delivery of internal properties during construction is the predominant form of interaction.

It is clear that tools and materials that are currently used extensively are largely ones that support internal properties during construction. Few tools address external properties. The causes of this situation cannot be established with confidence from the range of examples above, but these do allow some informed speculation as to why current tools are uneven in their support for external and internal properties.

One likely cause is the limited attention given to the quality of the final systems that are produced by tools or incorporate re-usable materials. However, this limited attention may reflect a deeper cause that lies in the nature of interactions between external properties and tools and materials.

External properties can often be delivered or proved during specification, but they must still be preserved during construction. Similarly, some properties can be assessed during specification, but the property must still be preserved during construction, and then re-tested during evaluation. Thus appropriate specification constructs and assessment tools cannot guarantee satisfaction of properties in the final system. This reduces the attractiveness of tools and materials that only interact at these levels with key properties. In contrast, a construction tool that delivers a property does so with no further effort. This difference in the effectiveness of specification and construction phase interactions may be an important cause of uneven tool and

material support for external and internal properties. It cannot however be the sole cause.

Differences of support between specification and construction are wider than they need to be. Construction tools often only assist in achieving properties, even though the gap between assistance and addressing/delivery may not be particularly great, especially when missing supporting constructs have already been implemented. Thus a construction tool that only assists with a property will require far less effort from developers if a relevant supporting construct is implemented and encapsulated. Such a simple addition of capabilities can be called an 'assistance upgrade', and this may be the obvious way to quickly improve on the state-of-the-art.

For most tools, 'assistance upgrades' should generally succeed. However, few have clearly been attempted in recent years. As such upgrades would be technically straightforward, the root cause of slow improvement of tool support for external properties must lie elsewhere.

The most plausible cause relates to the beneficiaries of 'hard' properties (i.e. ones with which proof, delivery or addressing interactions with tools/materials occur). When a property is satisfied, there are various beneficiaries. The major beneficiary from internal properties is the software developer. However, when external properties can be delivered, the users of a system rather than the developers of the tool or material will be the major beneficiary.

Tools and materials that improve internal properties have an immediate benefit for the software developer. Tools and materials that improve external properties have less immediate benefits. Although customer relationships should be strengthened, and the reputation of the developer for quality development should improve, the actual return to many developers on investment could remain uncertain, if not unclear. However, the customer–contractor relationship in software development is not fixed in stone, and closer, more open and more cooperative relationships are developing, just as they have in many areas of manufacturing. Interestingly, the one report from an in-house development site does identify specific external properties that are required by their operators ('craft' in telecommunications speak).

The apparent root cause of unnecessary differences in tool support for external and internal properties can thus be addressed. The site report from a campus research laboratory shows that external properties can be given focused attention when developing software infrastructure. Furthermore, the facilities in tools like TAE+ that have evolved to meet the needs of a large demanding and varied internal user base can also provide reasonable support for external properties. The obstacles to improving tool support at the specification stage for external properties are not largely due to technical obstacles to realizing some form of interaction, but, as has already been noted, due to the unsatisfactory loose ends that have to be addressed during

subsequent construction and evaluation. For these reasons, model-based UI tools have promise that goes beyond the generally cited improvements in development efficiency. Such tools could also preserve external properties that are proved, delivered, addressed, or assessed during specification.

This concludes the informed speculations on the causes of slow improvement of tool support for external properties. The main value of these conjectures is that they identify possible ways forward, especially the value of model-based development tools.

In summary, interactions with software development that were identified in Chapter 3 have been shown to be substantial, in that examples of different strengths of interaction can be readily found for existing tools and materials, but these interactions are diffuse, diverse and lack coherence. Internal properties are currently covered more comprehensively and coherently. Support for external properties is much more piecemeal, due to the risks of 'property erosion' during construction and evaluation. Model-based tools could be developed to address this 'property erosion'. Thus the predominance of interactions with internal properties reflects not fundamentals, but forces operating within software development. Designs for tools and materials that recognise the nature and degree of these forces are most likely to harness or counteract them.

Tools alone will not solve all problems associated with properties. There will still be trade-offs to be made. As with the last example site, methodology is not irrelevant, and it has some effectiveness even with partial tool support. Thus the proper combination of tools and methodology is also an issue that needs to be resolved.