CHAPTER 2

External Properties: the User's Perspective

2.1 Introduction

The usability of an interactive system is linked to the quality of the dialog, and quality shall here be expressed through a number of measurable properties of the dialog. The aim of this chapter is to identify and define a set of user-centered properties of interactive systems which promote high quality from the perspective of the users. The set must be as complete and mutually independent ('orthogonal') as possible. At the same time these so-called *external properties* must be usable in the software development process as yard-sticks or 'measures' in the quality plan for the development. For a particular system, some of the properties may be absolute requirements (this interactive system *must* have such and such property), while others are desired in a quality plan but are given some 'weight of importance' ($0 \leq w < 1$). Once we understand these properties and their implications, and also the internal properties presented in the next chapter, we will be able to discuss how to construct interactive systems possessing desired and required properties.

Two main approaches are used in this chapter to discuss external properties of interactive systems, informal and formal. The informal discussion, contained in Sections 2.2–2.4, provides a loose characterization of external usability properties as three main principles: task completeness, interaction flexibility and interaction robustness. Flexibility and robustness are further broken down into more basic properties. Each property is defined in natural language and we provide some concrete examples of the property in real systems to help understanding.

Section 2.5 discusses the use of mathematically-based formal models, as an aid towards understanding and distinguishing between various external properties. We do not present a complete formal description of each external property defined in the chapter, and there are no formal proofs of theorems about the properties. Rather, the emphasis is on explaining the kinds of formal models that have influenced our understanding of the external properties and on the distinctions between the various levels of abstraction defined in Chapter 1.

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

2.2 Goal and Task Completeness

The purpose of an interactive system is to allow users to attain their goals within a specific application domain. If users can reach any goal by means of the system we may talk about *goal completeness*. But this is not a measurable property of the system, it is rather a feature of the combination of users and system, since one cannot foresee every goal a user may form.

For the software developer, it is more relevant to consider *task completeness*, i.e. to ask whether the system supports all adopted tasks (tasks for computer support will be adopted during requirements specification). Succesful execution of these tasks when interacting with the system will lead to goal satisfaction. As adopted tasks will only be partially automated, computer support must be shown to meet user needs.

The relevant tasks are found during problem analysis, where future users are studied at their workplace. Descriptions of their work are analysed to isolate common goal states, typical and problematic initial task states, and regular procedures for task execution. This process is an early task analysis, prior to design (but task analyses after design are also important in HCI). It is an essential process in interactive system design and the system should support all of those tasks which have been identified. If we assume an acceptable adoption of identified tasks prior to requirements specification, then task completeness can be defined with respect to the task model. A system is task complete if each task defined in the task model is supported by the system.

Various task analysis methods can be used to generate a task model (Diaper, 1989, and Dix *et al.*, 1993). Tools that are able to generate code directly from a task model, or verify that code actually conforms to such a task model, will increase the likelihood of task completeness.

Note that we do not assume that a user will only ever perform tasks which were predicted by task analysis. Users are far too imaginative and inventive for that assumption to work. As Carroll and Rosson (1991) point out, computer artifacts themselves change the very tasks that users perform. It is therefore impossible to predict all goals or tasks that users may wish that a system supports. This does not, however, reduce the importance of task-driven design for those goals we can predict.

The principle of *task completeness* addresses the question:

- can I do my tasks at all and achieve my goals?

Behind this principle lie the tacit assumptions that users will adapt to the procedures imposed by the system for task execution and that they will make no errors in the process. These oversights can be avoided by asking two further questions about *flexibility* and *robustness*:

- can I do the task my own way? This means that the system should allow user choice during task execution as far as feasible.

- am I supported in doing the task successfully and realizing that I have succeeded? This means that the system should facilitate the user's actions and help the user to recover from mistakes.

These principles are discussed in the next two sections from a task-oriented point of view. The principles do not fully address the complete range of HCI problems. Instead they are a minimal replacement for a universal principle of *scenario completeness*, which addresses the question:

- can all users successfully complete any intended task, regardless of the initial task state (which includes their knowledge and beliefs) and regardless of all expectable events that could arise during task execution?

The principle of scenario completeness has deliberately been left out of consideration because it is not clearly defined and therefore not useful for software engineering. To simplify and even enable subsequent analysis, we have substituted 'adopted goals' for 'any intended task'. The principles of flexibility and robustness address some aspects of typical scenarios, where variations are due to users' preferences, mistakes or slips rather than to the demands of their physical, social and work environments, or to inappropriate beliefs or assumptions.

We have two justifications for this reduced focus:

1. this book reports the first systematic attempt to establish links between what users need and the ways in which software is constructed. It is unreasonable to attempt or expect comprehensiveness in such exploratory work.
2. critiques of task-based design (Bannon and Bødker, 1991, and Benyon, 1992) are still largely polemical, where credible arguments have yet to be backed up by practical consequences. Thus, while we know that learning, users' knowledge, social interaction, working divisions of labour, working practices and situated activity are all relevant to the design of systems, the form of this relevance is not yet clear, and thus we cannot be expected to establish links between what humans need and the ways in which software is constructed.

Our hope is that we have made no commitments to, and depend on no assumptions about theories of human activity that will obstruct future extensions of our framework to incorporate more demanding aspects of human activity.

2.3 Interaction Flexibility

Interaction flexibility refers to the multiplicity of ways in which the user and the system exchange information during task execution. This can apply at all levels of description defined in Chapter 1: functional, dialog, logical and physical levels. Interaction flexibility requires designers to recognize that

people react differently, and designers must respond to user differences and preferences by providing a variety of interaction techniques.

We will list interaction flexibility properties under three subcategories: representation (of information), planning (of task execution) and adaptation (of dialog forms).

Representation of information:

- **F1. Device multiplicity** – the capacity of the system to offer multiple input and output devices for communication. Input devices include microphone, keyboard, mouse, dataglove, video camera, etc. Output devices include screen, loudspeaker, force-feedback joystick, etc.
- **F2. Representation multiplicity** – the capacity of the system to offer alternative representations for both input and output.
- **F3. Input/Output re-use** – the capacity of the system to allow usage of previous input or output as future input.

Planning of task execution:

- **F4. Human role multiplicity** – the capacity of the system to support users with different roles.
- **F5. Multithreading** – users can engage in several tasks which may overlap in time. In these cases, the system can provide support for the simultaneously active task threads.
- **F6. Non-preemptiveness** – Preemption occurs when the system enforces a sequence of interaction that is not necessarily expected by the user. Non-preemptiveness is the absence of preemption.
- **F7. Reachability** – the capacity of the system to allow users to reach any system state, regardless of the current state.

Adaptation of dialog forms:

- **F8. Reconfigurability** – the capacity of the system to support user-initiated customization of the interaction.
- **F9. Adaptivity** – the capacity of the system to initiate customization of the interaction.
- **F10. Migratability** – the capacity of the system to support user- or system-initiated transfer of task responsibility.

Reconfigurability and adaptivity are often dealt with as one property, *customizability*, but the software developer must distinguish between functionality that gives the users some choices (reconfigurability) and features that makes the system take the initiative (adaptivity).

The remainder of this section will further examine each of these properties. These properties are again summarized in Table 2.1 at the end of this section.

2.3.1 Device multiplicity

Device multiplicity means that multiple input and output devices can be used for the dialog. For example, in an airline reservation system, the customers might type or speak their queries in natural language. The logical representation in both cases is the same, natural language, but different input devices are used to articulate the query.

Whereas representation multiplicity refers to the dialog and/or logical levels of interaction, device multiplicity refers to the lowest level of interaction flexibility, the physical level; the word device is preferred to other terms used, such as 'media' and 'modality', because they have several interpretations. Some authors refer to a device as a media type. For others, the word media is not constrained to the physical level, but can be understood at higher levels of abstraction (Blattner and Dannenberg, 1992), and the media is understood as a representational system (Alty, 1991). Others, like Bernsen (1993), use the term modality to refer to a representational system. Given this overlap in interpretation between media and modality (which extends to multimedia and multimodality), we will be explicit about our meanings for these terms. Media refers to the device or physical level, whereas modality refers to all other levels of abstraction. Therefore, multimedia corresponds with device multiplicity, while multimodality is a more complex property corresponding with representation multiplicity, the next flexibility property.

2.3.2 Representation multiplicity

Representation multiplicity concerns flexibility in the rendering of state elements as well as in the articulation of input. Multiple representation covers the variation of information content as well as the presentation of the information. For example, on output a system could support alternative representations of the notion of temperature over a period of time. It can be presented as a thermometer, if the actual numerical value is important, or as a graph if it is important to notice trends. It might even be desirable to make both representations simultaneously available to the user. Each representation provides a perspective on the internal state of the system. At a given time, the user or the system is free to consider the representations that are most suitable for the task. When several renderings are presented simultaneously, it is often called multimodality.

Alternatively, a single output on the screen can represent a synthesized representation of a collection of internal system values. For example, performance meters aggregate all system data concerning resource usage and present that information as a timeplot to indicate overall utilization. A quick glance at such a plot can help a user understand performance trends.

An example of input multiplicity would be in a drawing package where

a user may draw lines by direct manipulation or by specifying the end coordinates numerically in text fields. Depending on the task requirements, either means of specifying the input can be important, and both may need to be equally available. If both line and numeric values are simultaneously presented, we have a particularly rich form of interaction referred to as equal opportunity by Thimbleby (1990).

As well as using different representations alternatively they can also be used simultaneously to achieve a certain effect. For example, a sound effect (a 'bip') may be issued as a new window appears.

Representation multiplicity is related to multi-modality but the latter is a more complex issue. Multi-modality also covers how signals on two or more modalities (communication channels) are combined to form a single message.

2.3.3 Input/Output re-use

It is possible to articulate an input expression by referring to previous input or output expressions. Cut, paste and copy commands are typical examples of input and output re-use (I/O re-use). Another simple example of re-use occurs in command line interfaces in which users can select commands from their previous input and re-issue them to the system.

Two interesting issues arise. First, there are implications for type coercion. The second issue is at what level of abstraction is the re-used information interpreted. Type coercion is necessary when the source and target systems use different data structures. For example, the internal representation of a circle differs between object-based and pixel-based graphical editors. Cut and paste re-use will have to support conversion between these different internal representations. Furthermore, it is desirable that such type coercion be invertible, but this is a complicated issue involving interoperability, which we discuss further in Chapter 3.

The issue with interpretation at different levels has implications on how interaction histories are stored by the system. At the physical level, re-use would mean that actual keystrokes would be recorded. Re-use at the functional level would mean that functional-level information (e.g. the command code and the value of arguments) would be recorded. Re-use at the dialog level is rare, as input re-use generally is implemented at the extremes of keystroke input or command invocation. Indeed, it is difficult to identify advantages for dialog level re-use that are not matched and exceeded at the functional level. Aspects of I/O re-use are further discussed in Section 4.3.

Default behavior is related to I/O re-use. Default values are generated by the system based on prior user interaction; they provide input or output based on prior history. Adjusting default behavior based on the interac-

tion history in this way is similar to system-initiated adaptation, called adaptivity below.

2.3.4 Human role multiplicity

In multi-user systems, different users serve different roles or functions in their interactions with the system and, ultimately, other users. A role in this context refers to users' goals and determines the kinds of tasks that they will want to perform, the methods or commands used to accomplish those tasks, and the system objects that will be necessary to complete their actions. A user role is thus identified by a cluster of goals that is allocated to or adopted by a user within some division of labour. Roles may overlap and change over time.

So, for example, in a multi-party conferencing situation, one participant might play the role of the manager whose responsibility is to explicitly pass floor control to the other participants. The participants can assume the role of either the speaker or a member of the audience. Thus, the conferencing system calls for three different roles: manager, speaker and audience. Human role multiplicity refers to the extent to which the system supports the various roles that users can assume.

This variability of roles is most obvious in multi-user systems, but it is certainly not exclusive to groupware. The major distinction between single-user and multi-user systems is that multi-user systems will have to support multiple human roles simultaneously, but also a single-user system must be able to support the user in different roles, sequentially. A simple example of role multiplicity occurs with Hypercard, which has five pre-defined roles that a single user may switch between — scripting, authoring, painting, typing and browsing. Hypercard uses these role definitions both to ease the novice user into more sophisticated programming tasks and to limit some users from changing secure parts of a Hypercard stack. This last concern raises questions about how the system controls access to various system objects, a topic linked to role multiplicity that we discuss under the property of access control for the interaction robustness principle.

Another issue of concern for a system supporting multiple roles is how users are able to change roles. Changes in role can be either user initiated or system initiated, as discussed below under the headings 'Adaptivity' and 'Reconfigurability'. In the conferencing example, it is the duty of the manager to explicitly assign roles to the conference participants. A member of the audience sends a request to the manager to gain control of the floor, but it is up to the manager to make the appropriate role assignments (e.g. reassigning the current speaker to be a member of the audience and assigning the requestor to be the new speaker). So the manager has the capability to reconfigure the system. An adaptive Hypercard system, on

the other hand, might sense that a novice user needs to alter text in a Hypercard stack, and therefore change her role from 'browsing' to 'editing'.

2.3.5 Multithreading

In cooperative systems with several simultaneous users multithreading is a *sine qua non* because the users want to execute their individual task threads in parallel.

In a single-user environment the need for multithreading is less clear cut, and a number of arguments may be brought into play. The rest of this subsection discusses different ways and different levels of multithreading in a system with one user (or a few users).

Often a user wants to do several things in parallel, and if the system has concurrent capabilities, multithreading may help users to achieve their goals. This contributes toward interaction flexibility since it lets users perform multiple tasks simultaneously or switch freely between them. The user may want parallelism on more than one level of abstraction: (i) at the functional level, where parallel *command execution* is possible; (ii) at the dialog level, where parallel *command specification* is possible; (iii) at the logical interaction level, where parallel formation of *elements* of a command specification is possible; and (iv) at the physical level, where parallel formation of *an element* of a command specification is possible.

Multithreading at one level of abstraction implies nothing about threading at a higher level. We can have multithreading at the physical level (e.g. simultaneous input of keyboard and mouse events), but not at the logical interaction level (when they are formed into a single element, i.e. an edited text field value).

Similarly, we can have multithreading at the logical interaction level (e.g. simultaneous formation of options and arguments respectively by keyboard short cuts and mouse selections), but not at the dialog level (where options and arguments part of the same command specification). Lastly, we can have multithreading at the dialog level (e.g. simultaneous formation of command specification in separate windows), but not at the functional level (where command executions are serialized).

At each level of abstraction, the kind of multithreading possible depends on what kind of parallelism the underlying system supports. A system with *concurrent* multithreading allows simultaneous communication of a set of elements that are fully formed at a higher level of abstraction.

A system having *interleaved* multithreading permits a temporal overlap between articulations or specifications but stipulates that at any given instant, only one element of a fully formed structure is being articulated (below dialog level), specified (at dialog level), or executed (at functional level). Concurrent interleaving at one level of abstraction may thus be replaced by interleaved multithreading at a higher level.

Multithreading may also be replaced by single threading at a lower level, but this imposes extra interaction steps on users. Thus in a typical window system, the interaction is interleaved multithreaded at the dialog level, as the user may communicate interleaved with a number of open windows supporting different tasks, but at the physical level the user interacts with one window at a time in a serial manner, utilizing one mouse and one keyboard only. This imposes explicit changes of focus on users. Similarly, operating system command languages with backgrounding or job control are multithreading at the functional level, but command specification at the dialog level is single-threaded, forcing the sequential articulation of operations, options and parameters. This imposes explicit articulations of backgrounding and foregrounding on the users.

Just as serialization below interleaved multithreading complicates interaction traces (by adding explicit changes of focus), so concurrent multithreading is more simple than interleaved multithreading, as it avoids suspension and resumption of interrupted formations, and avoids explicit backgrounding and foregrounding.

As an example at the physical level, consider two acts for a user of a computer painting application: changing brush width and painting on the canvas. In a single-threaded or interleaved multithreaded-system, users would have to suspend painting acts in order to change brush width. In a concurrent multithreaded system, such as VoicePaint (Gourdol *et al.*, 1992), users could change the size of the brush as they are painting, and thus be able to paint a crescent moon in one uninterrupted stroke of a locator device.

Designers must carefully consider differences in 'threading' between levels of abstraction, since transitions from single to multithreading and from concurrent to interleaved multithreading require users to plan their interactions more carefully. Interaction is both more flexible and more simple when multithreading is concurrent at all levels of abstraction in interactive systems.

2.3.6 Non-preemptiveness

When considering the interaction between user and system as a dialog between partners, it is important to consider which partner has the initiative in the conversation. The system can initiate all dialog, in which case the user simply responds to requests for information or action. We call this type of dialog system-driven because the system more or less decides which action (or actions) the user may perform next. Alternatively, the system might only react to user input, in which case the dialog is called user-driven because the user has more freedom in choosing the next action. A dialog where either the user or the system may have the initiative is called a mixed-initiative dialog.

Non-preemptiveness refers to the degree of freedom the user has in deciding what next action to perform at the interface, and it is one of the key factors contributing to the user's feeling of flexibility in the dialog. System-driven models of interaction tend to be preemptive, they limit the user's choice of next available action, whereas user-driven interaction favors non-preemption.

Preemptive systems limit the user's options for communication. For example, a dialog box may prevent the user from interacting with the system in any way that does not direct input to the box. From the user's perspective, a system-driven interaction hinders flexibility whereas a user-driven interaction favors it. In general, we want to minimize the system's ability to preempt the user although some situations may require it for safety reasons. In situations in which a user error or slip would result in serious damage without a chance for recovery, it is desirable – or even necessary – to limit user freedom.

The task analyst must have a good understanding of the sets of tasks the user is likely to perform with a system and how those tasks are related in order to minimize the likelihood that the users will be prevented from initiating or advancing some task at a time when they would want.

2.3.7 Reachability

Reachability refers to the possibility of navigation through the system states. It can be defined at any level of detail, but in this context only observable states are of interest. Various aspects of reachability have been given formal definitions, but the main notion is whether the user can navigate from any given observable state to any other observable state. From the user's point of view it may be useful to distinguish between backward and forward reachability.

The user may want *backward reachability* in order to get back to some previous state of the interaction, after having made a mistake or realizing a need for some previous information. This type of reachability is covered by the property of recoverability, as it is usually defined, and it requires sufficient history information to be kept by the system.

Forward reachability means that the user is able to proceed to any desired interaction state, independently of previous dialog development.

These coarse definitions say nothing about how difficult it is to go from one state to another. For example, in order to make a text editor fully reachable, we only need to provide the ability to insert letters sequentially and erase the entire contents of the editing buffer. Such an editor will be reachable because any possible buffer contents can be achieved from any other buffer state. However, the only way to effectively delete the last character would be to erase the whole buffer and type in the entire contents again! To get around this over-simplified reachability criterion, we

can invoke Thimbleby's principle (1990) of *commensurate effort* — things that are easy to do should be easy to undo. Since it is easy to mistype a character in a text editor, it should be just as easy to undo that error (e.g. by providing a delete character command).

Therefore, a more general definition of reachability should include some measure of the ease of navigating. A system has good reachability if the user can navigate from one observable state to another with an effort which in some sense is commensurate with the user's expectation.

2.3.8 Reconfigurability

Reconfigurability refers to the user's ability to adjust the form of input and output. This customization may be very limited, with the user only allowed to adjust the position of soft buttons on the screen or redefine command names. The power given to the user can be increased by allowing the definition of macros to speed up the articulation of certain common tasks. In the extreme, the interface can provide the user with programming language capabilities, such as the Unix shell or the script language Hypertalk in Hypercard.

Such user-initiated changes can also have varying periods of duration. For example, changes could be limited to one interaction session or they could be recorded and affect all future sessions (e.g. resource settings in the X Window system).

2.3.9 Adaptivity

Adaptivity is automatic customization of the user interface by the system. Decisions for adaptation can be based on user expertise or observed repetition of certain task sequences. The distinction between adaptivity and reconfigurability is that in a reconfigurable interface the user plays an explicit role in customization, whereas his role in an adaptive interface is more implicit. A system can be trained to recognize the behavior of an expert or novice and accordingly adjust its dialog control or help system automatically to match the needs of the current user. This is in contrast with a system which would require the user to explicitly classify themselves as novice or expert at the beginning of a session (Kuehme *et al.*, 1992).

Automatic macro construction, as proposed in the Eager system (Cypher, 1991), combines reconfigurability with adaptivity in a simple and useful way. Repetitive tasks can be detected by observing user behavior, and macros can be automatically constructed from this observation to perform repetitive tasks automatically.

2.3.10 Migratability

Task migratability concerns the transfer of control for events or execution of tasks between system and user. It should be possible for the user or system to pass the control of a task over to the other or promote the task from a completely internalized one to a shared and co-operative venture. Hence, a task that is internal to one can become internal to the other or shared between the two partners.

Table 2.1 *Summary of interaction flexibility properties. The 'Description' column contains a short description of each property; the 'Related properties' column is a reminder of relationships to other properties mentioned under each property*

Flexibility Property	Description	Related properties
Representation:		
Device multiplicity	More than one way to do something	Multi-media capability
Representation multiplicity	More than one way to present something	I/O multiplicity, equal opportunity, multi-modality
Input/Output re-use	History repeating itself	Use of defaults
Planning:		
Human role multiplicity	Several people doing several things	Access control
Multithreading	One person doing several things	Concurrency, interleaving
Non-preemptiveness	Doing what you want when you want	User-driven dialog, mixed-initiative dialog
Reachability	Getting anywhere from anywhere else	Commensurate effort
Adaptivity:		
Reconfigurability	The user changing the interaction	Programmability of the interface
Adaptivity	The system changing the interaction	Automatic macro construction
Migratability	Transferring control	

As for many other properties, migration can occur at multiple levels of abstraction. At the physical level, the provision of command completion

migrates responsibility for some physical operations (typing) from the user to the system. At the dialog level a system with number input could allow the user to enter not only literals (e.g. 24) but also numerical expressions (e.g. 6×4). Here a step that would be a user calculation in a full cognitive task description is migrated to the system. At the functional level, file saving is an example of a migratable task: one user may do it explicitly, and another user may want automatic file saving. Interestingly, the effect of migration at the functional level is to reduce the number of commands and objects that users are effectively in contact with during subsequent interactions.

2.4 Interaction Robustness

An interactive system is called robust if it supports a user in performing a chosen task without irreversible mistakes, and if it gives users a correct and complete picture of task progress. Thus, interaction robustness covers all those properties that minimize the risk of task failure. We identify seven properties that contribute to the principle of interaction robustness. The first three properties ensure a correct and complete picture of the system, while the last four properties lessen the risk and cost of mistakes.

Correct picture:

R1. Observability – the system makes all relevant information potentially available to the user.

R2. Insistence – the dialog structure ensures that necessary information is perceived.

R3. Honesty – the dialog structure ensures that users correctly interpret perceived information.

Few mistakes (no irreversible ones):

R4. Predictability – users can predict future states and system response time from the current and prior observable states.

R5. Access control – the system allows for defining control policy and availability for information access.

R6. Pace tolerance – the system allows users to control the pace of interaction.

R7. Deviation tolerance – the system supports users' correction of slips and errors.

The first four of these properties – Observability, Insistence, Honesty and Predictability – are very user-dependent; they can only be validated by user testing. The last three – Access control, Pace tolerance and Deviation tolerance – are less user-dependent and can be validated reasonably within

a system (either by analysis of specifications or by expert walkthrough of an implementation).

The remainder of this section will further examine each of these properties. These properties are again summarized in Table 2.2 at the end of the section.

2.4.1 Observability

A system is observable if it allows users to inspect all information relevant to their tasks. This does not necessarily mean that all relevant data are presented at once. A typical screen-based computer system can only render a small amount of the total information on an output device. Hence there must be some browsing function allowing the user to inspect the information in stages.

An important part of observability is to restrict it to all *relevant* information. It could be argued that one would like to be able to view all of the system state, but even if this were possible the user might suffer information overload. So, in large industrial control rooms and aircraft cockpits where vast numbers of dials have traditionally displayed all of the state, there is an emphasis on glass displays which focus the operator's attention on parts of the available information.

We want the immediately perceivable information to be relevant and sufficient for the user's current tasks. The task model should contribute to this design issue, by identifying those elements of the system state which are most critical at each interaction point. Having identified these elements, lower levels of design can ensure that the identified elements are indeed rendered at the required times. In a constrained process this task-based identification may be sufficient, but in most systems the user will have some control as to which elements are displayed, for example in choosing the positioning and visibility of windows.

Clearly in most situations only the most critical information can be immediately available. However, the user should be able to access all relevant information eventually. This kind of observability – often called *browsability* – is based on the general principle of allowing the users to perceive anything they can name, i.e. anything he or she can provide a description for. If the description is not ambiguous, then the required information should be provided; if it is ambiguous, the user should be given sufficient information about possible choices in the present state. This means that the system must present the information and the possible actions which seem to be useful. Furthermore, this browsing of information should be possible without modifying the system state (other than the form and contents of the current presentation). This requires some kind of multithreading allowing the user effortless return to the state from which the browsing was started.

It is particularly important that a user is aware of the effect of the last

action. That is, that there is effective feedback. Ideally, all changes should be immediately perceivable, but where this is not possible (for example, in a document-wide search/replace), at least some indication of the effects should be given (for example, displaying the number of changes that resulted from the search/replace). In addition, a user ought to be able to browse the effects of the previous action, but this is frequently inadequately supported. For example, after a search/replace the user ought to be able to find out precisely which words were affected, but this is usually impossible.

In an open system like a CSCW system,* the user will also want to be aware of the actions of other users and of external processes. This observability of other users' actions has been referred to as *feedthrough* (Dix, 1994). In practice, we may accept weaker feedthrough mechanisms than those for feedback. For example, in a shared editor feedback of a user's own typing must be virtually instantaneous, whereas network delays of a few seconds may be acceptable for feedthrough. However, as the user did not initiate the actions, merely seeing the effect may be insufficient; it is frequently helpful for feedthrough to identify both the action which caused the change and the user who initiated it. This can be very important in allowing users to interpret the intention behind other users' actions.

The observability of a system is – like the predictability discussed below – tied to the user's understanding of the system and their expectations of its behavior.

2.4.2 Insistence

Just because information is available at the interface, it is not necessarily the case that the user will notice it. For example, one persistent problem in windowed systems is the situation where the user mistakes which window is selected and then directs text at the wrong window. All window systems give some clue as to the active window, often by highlighting the window's name, or emphasizing its border. However, if the user is looking at the content of the window, these borders may be insufficiently salient.

Software designers should do their best to ensure that critical information is not only available, but is actually *perceived* by the user. In addition, the system must ensure that events are reported and noticed at the appropriate time. In particular, if an event indicator is ephemeral (e.g. a buzzer) and the user is temporarily absent, then the user may never realize that the event has occurred.

Insistence can be achieved by various means: by increasing the visual salience, by interrupting the user with pre-emptive dialogs, by using aural signals or by leaving persistent event indicators. The choice of mechanism

* CSCW = Computer Supported Cooperative Work.

is again finely dependent on the plausible cognitive task description, which should identify two aspects:

- where the user's attention is likely to be (in order to assess the salience of a particular feature);
- the required timeliness and salience of different system elements.

Different interface widgets have different salience and timeliness properties. For example, a buzzer may demand instant attention whereas the appearance of a new icon may be eventually noticed after a few minutes. Too many over-salient features will lead to a noisy and unpleasant interface and furthermore will hide the features which are really important. The designer should therefore attempt to match the properties of the interface widgets to the required salience and timeliness required by the user's task.

In an open or multi-user system it is also important that the user is made aware of appropriate events generated by other users and the environment. For critical events this should be very salient, but it is also useful to generate a low-level, ambient indication of external activity. This is called *awareness* in CSCW systems. Experiments have shown that this can be very important in giving users a sense of working together and in diagnosing changes in the world (Gaver and Smith, 1990).

2.4.3 Honesty

Designers need to ensure that users interpret the symbols at the interface in the way that they are intended. In Norman's terms (1988), this is called the 'gulf of evaluation'. Maybe a system is observable and insistent, but if users misinterpret the information, there is bound to be trouble. Honest systems strive to achieve a match between the user's interpretation and the designer's intended interpretation of the interface. This requires that (i) the observable state conforms with and represents the relevant features of the system state, and (ii) the user interprets the rendered information correctly. The second part of this can only be validated through user testing, but the first part is the responsibility of the designer.

Various heuristics exist for promoting the honesty of a system. The notion of affordance from psychology deals with how artifacts in the real world (including a computer's interface) suggest the correct mode of operation. For example, buttons should suggest pushing them to get some operation. Mouse cursors should suggest pointing or dragging. Metaphors are frequently used in an interface to borrow from a user's previous knowledge of how things behave in the world. If a designer structures the interaction consistently according to some clear metaphor to the user, the user's familiarity with the metaphor will assist them in guessing the correct behavior. The potential danger with metaphors in a computer interface is that they might suggest operations which are not possible, or they might suggest the

wrong interpretation of an action (e.g. dragging the icon for a floppy disk, or the hard disk, to the trash icon in the Macintosh desktop metaphor).

Observability and insistence are robustness properties that support honesty, but also flexibility properties may improve the honesty of a system. As an example, if a system is reconfigurable (the user may change the interaction), users can make changes that make the system more honest to them; and an adaptive system (the system changing the interaction) may adapt to a user's habits and thereby be perceived as more honest.

2.4.4 Predictability

The above discussions of observability and insistence focus on the extent to which the system provides enough information for the user to know how past actions have affected the present state. Honesty refers to how this information is correctly comprehended by the user. Predictability concerns the future, that is, to what extent information in the past and present can help the user determine the outcome of future interactions. Like the first three robustness properties predictability depends not only on the system itself but also on the actual user knowledge and expectations. But the designer can do much to further (or impede) the predictability as perceived by the users.

Roughly speaking, predictability means that a user's knowledge of the past interactions and current observable state are sufficient to determine future behavior of the system. The system must be designed to provide state and action information in a reasonably complete, systematic and consistent way. Finer measures of predictability are concerned with just how much of the past is necessary and to what extent behavior is predictable based on immediately observable information.

Consistency is one heuristic that is often applied to increase the predictability of an interface. Consistency allows the user to generalize from specific situations to similar situations. But it is difficult at times to determine, at design time, which situations a user will consider similar or dissimilar.

Predictability is about actions as well as effects. Users need to know not only what will happen when they issue a command, they need to know what commands are available to them at any point. Several properties we have discussed already are related to this property of operation visibility. Affordance means that the operations which are suggested to a user are the ones that are available, that is, affordance means honesty with respect to operation visibility. If the user sees a button that says cancel, is it really the operation that is available? And what will be cancelled? This may depend on the user's actual role, as discussed in Section 2.4.5; when users play limited roles they should know that their set of possible actions is limited.

Users also expect the system to be stable with respect to response times.

A predictable system must therefore exhibit *temporal stability*: if the same action – for example, opening a new file – is executed several times, the response time should be the same each time. Furthermore, execution times for similar tasks (i.e. tasks which the user perceives as similar) must be close to each other.

2.4.5 Access control

Access control mechanisms restrict those parts of a system which a user can view or alter. This is particularly important in systems where the users can assume multiple roles. In a groupware application supporting synchronous editing, certain users could be designated as the editors of parts of the text. Users who are not editors of part of a text should not be allowed to make changes to that text, though they may be allowed to view the text.

Access control addresses the robustness concerns brought about by human role multiplicity. In a single-user context, the actions of users in some roles might be restricted in order to prevent damage to the system. Users in other roles might have very few restrictions placed on their interaction. Though there are good reasons to allow such free interaction, it is not without risk. For example, a Unix super-user has free reign, which means that most operating system safeguards which apply to normal users (e.g. file ownership) are circumvented. Whereas the power of the super-user is necessary for some tasks, it means that the system is less forgiving of slips at the interface.

A simple example of an access control mechanism arises in most modern operating systems, in which protection schemes are used to tailor access to files and control who can alter file contents. In a multi-user text editor, users in the role of commentator are prevented from changing the text owned by some author, though their comments are made visible to editors and authors.

Access control is not always motivated by the desire to prevent users from doing bad things. The training wheels metaphor (Carroll and Carrithers, 1984) in which a system adjusts the availability of functionality based on a user's level of expertise, is intended to ease the learning burden for the novice. Rather than risk exposing a new user to the daunting variety of potential functions, the system begins by offering a minimal set of functions, gradually revealing more functions as user experience grows. This gradual revelation of system functionality does limit the user's access to the system but with the intention of decreasing anxiety and increasing learnability.

In multi-user systems, it is important that users be aware of other users' activity, even if they cannot affect their actions directly. We saw an example of this earlier in the multi-user editor with the visibility of comments. This is related to feedthrough (see Section 2.4.1). In a shared graphical editor, each user may be limited to actions on their own private painting layer.

The system superimposes all layers to reveal to each user what the overall shared picture looks like. Though individual users cannot alter any drawing on another user's layer, they can still see what the others are doing and can coordinate their drawing activity accordingly.

When a user's actions are limited by some access control mechanism, it is also important that they understand the scope of the access control. Again, in the multi-user editor a single author not only wants to know what parts of a document are available to them for editing but also wants to know what other parts of the document are open to others for editing, indicating which parts of the text are liable to change. For example, consider the design of media spaces (i.e. computer mediated video for computer-supported co-operative work), where users can generally make arbitrary live video connections with other users. Privacy issues demand that a user be able to prevent arbitrary intrusion. When someone wants to make a connection with another user and that connection is denied, the system could indicate whether the person is busy with another video connection or is restricting access to their office for the time being.

2.4.6 Pace tolerance

Interaction take place in the real world. There the time it takes for things to occur matters. It takes time for both the user and the system to react to changes in the world. A pace-tolerant system considers the timing match between user expectation and system demands. For example, it takes time for a user to read a message, so sending numerous error messages to screens which scroll away faster than they can be read is a timing mismatch. Designers of such systems fail to realize that what is a quick task for the system is not so quick a task for the user.

As technology improves, we may be led to believe that the system will always be able to keep up with the user. Dix (1991) points out that adherence to this myth of the infinitely fast machine would lead us to conclude that we only ever have to worry about slowing the system down to the user's level. No matter how our technology progresses, it will always be the case that things take time. External factors (network latency, hardware failures, etc.) and increasing system demands (load the entire Oxford English Dictionary when doing a spell check!) will always mean that we should design the system knowing that delays will be present. Pace-tolerant design is conscious of how to meet the user's expectations both when the system is too fast and too slow. A standard concession to the 'faster' user is to provide type-ahead; while the system is busy doing some other task, the user is allowed to provide input to the system that will be interpreted once the system is available. Type-ahead acknowledges that there are situations in which the user works faster than the system.

How to interpret typed- (or clicked-)ahead commands is a tricky issue,

especially in a graphical user interface and it points out the criticality of pace-tolerant design. It is very easy for a user to click at several soft buttons quicker than the system can respond to them. Suppose a user sends a command to launch an application in a new window. While waiting for the new application to initialize and appear on the screen, the user decides to send some commands to other windows. The system stores the user actions to be interpreted once it has finished the task of launching the new application. In what context are those actions interpreted once the system is ready to respond to them? The reasonable answer is that they should be interpreted in the context in which they were issued, but if all that is stored in the type-ahead buffer are physical actions (keystrokes, mouse clicks and mouse positions), then it is virtually impossible to guarantee that the actions will be interpreted in the correct context. Pace-tolerant type-ahead is not a solution that can be implemented at the physical level alone, but involves higher levels of abstraction as well.

When interface behavior is time-dependent and the user is not aware of the dependency, the interface will be hard to learn. How difficult is it for users to understand the difference between a double-click of the mouse and two separate mouse clicks? And once they understand the difference, will they expect a different system interpretation for two mouse clicks than for a single one? If users don't expect the speed of their interactions to affect the interpretation, then they probably won't learn on an interface that relies on it.

2.4.7 Deviation tolerance

No matter how well a system has been designed, users will commit errors from which they will want to recover. Deviation-tolerant systems may support (i) detection of error states or 'dangerous' states, (ii) prevention from getting into error states and (iii) correction of slips and errors. When users can rely on being warned against dangerous actions and being assisted in recovering from small errors, then they will feel more free to explore an interface without worry, i.e. the user will experiment with the interface with the expectation of being able to undo some operation once he or she has learned how it works.

Even in a system with a very careful design of mechanisms for detecting error states and for prevention, the users will commit errors. Therefore the most important aspect of deviation tolerance is the provision of recovery procedures for the user. Recovery is a special form of reachability; the user wants to get from some state of interaction to another. The initial state for recovery is an error state (an unwanted state) and the final state is the corrected state. There can be two different strategies for recovery – backward and forward. A backward recovery strategy, such as undo, has a previously attained state as the corrected state. A forward recovery strategy, such as

Table 2.2 *Summary of interaction robustness properties. In the second column, a '+' indicates the property is tied to user expectation and that validation depends on user testing. The 'Description' column contains a short description of each property. The 'Related properties' column is a reminder of relationships to other properties mentioned under each property*

Robustness Property	User dep.	Description	Related properties
Observability	+	The user may perceive	Immediacy, browsability, feedback, feedthrough
Insistence	+	The user will perceive	Salience, timeliness, persistence, awareness
Honesty	++	The user correctly comprehends	Affordance, familiarity, suggestiveness, guessability
Predictability	+	Understanding how the system will react	Observability, consistency, affordance, response time stability
Access control		Role-sensitive restriction of information availability	Human role multiplicity, feedthrough, awareness, visibility, privacy
Pace tolerance		Response times match user's expectations	Timeliness, adaptivity, migratability
Deviation tolerance		User's recovery intentions are supported	Forward/backward recoverability, commensurate effort, pre-emptiveness

negotiation in a cooperative system, selects a previously unattained state as the corrected state. In some cases only one of the strategies will be available. When both strategies are available, it is the user who decides which to adopt. Recovery should be viewed, therefore, as a user intention, not as a function provided by the system. The designer must make the system deviation tolerant by supplying understandable and easy 'escape routes' from anticipated unwanted states, not by building one recovery function.

Thimbleby's notion (1990) of commensurate effort is again important here. In this situation it means that if an error is easy to commit, its effect must be easy to recover from.

The robustness property of deviation tolerance must be balanced against the flexibility property of non-preemptiveness. A non-preemptive system 'allows the user to do anything in any case', while a deviation-tolerant system 'guides the user away from dangerous slips and errors' and in case of a

slip 'guides the user towards safe recovery'. For example, a non-preemptive power station control system would allow the operator to close a cooling water pump without further ado, but a deviation-tolerant system would react with a modal warning saying 'you can't do this unless you also...'.

2.5 Formal Modeling of External Properties

The external properties cover so wide a spectrum that no single formal model can be used to define and discuss all of them. Some of the properties are tied intimately to the users' perception and behavior, henceforth they are difficult or impossible to formalize.

In this section, we present some simple formal modeling activities which lead to a clearer understanding of some of the external properties associated with usability. But our formal model does not contain real time, and therefore the temporal aspects of a dialog are not formalized below.

In Chapter 1 we introduced four different levels of abstraction for an interactive system: the functional, the dialog, the logical and the physical level. At each of these levels we describe the interactive system as a deterministic state machine, or labeled transition system, consisting of a set of states (system states), a set of events and a function which relates events to state transitions. A state machine, $\mathcal{M}$, is a 3-tuple

$$\mathcal{M} = (S, E, \rightsquigarrow),$$

where

S	=	the set of possible states the machine can assume;
E	=	the set of events or operations the machine can engage in, sometimes referred to as the alphabet;
$\rightsquigarrow$	=	the transition function, which maps events in E to transitions on S. The signature is: $\rightsquigarrow : S \times E \mapsto S$.

We are thus introducing four state machines, one for each level of abstraction, and below we shall further introduce the set *Obs* of observable states, i.e. the states which the user can perceive and distinguish.

But first we use the general machine model $\mathcal{M}$ to illustrate the difference between some of the properties described informally in Sections 2.2–2.4. For example, we can distinguish between reachability and non-preemptiveness. To make this distinction, we will appeal to a graphic depiction of a state machine, or state transition diagram, like the example shown in Figure 2.1.

2.5.1 Flexibility properties formalized

Reachability refers to the connectedness of the state transition graph. Initial reachability means that the user can get from the initial state to any other state in the system. In the example in Figure 2.1, with $S1$ as initial state, the

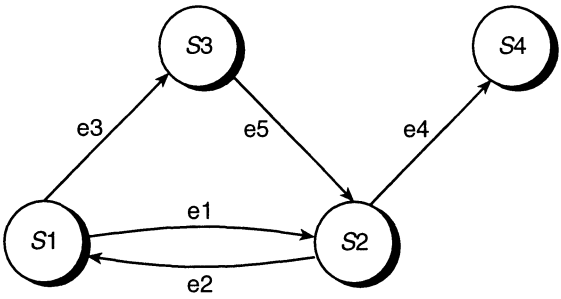

Figure 2.1 *A transition diagram for a simple dialog with four observable states. S1 is the initial state, and e1–e5 are events triggered by user actions.*

dialog has this initial reachability property. In the more general formulation of reachability we ask whether it is possible to get from any state to any other state. In the example, this form of reachability is not satisfied, since there is no event transition from state $S4$ to any other state in the system.

Non-preemptiveness, on the other hand, is not just asking about connectedness, but about the shortest paths between states. It can be defined more or less fine-grained. Complete non-preemptiveness means that the transition diagram is fully connected, i.e. there is a direct path from any state to any other state. Such a requirement does not make sense in a real system of any complexity, and therefore a system design must include decisions on the degree of non-preemptiveness required.

Whereas there is a path from state $S3$ to $S1$ in Figure 2.1, it is only possible to get to $S1$ by first going through $S2$. This intermediary state is pre-emptive. This example demonstrates that non-preemptiveness is relative; it depends on what actions or tasks the user wants to perform. If the user never would want to get to $S1$ from $S3$, then this pre-emption is acceptable (and maybe even helpful by preventing unwanted actions).

Multithreadedness refers to the possibility of having independent threads of activity going on at the same time, and it can only be expressed here by having several state machines acting in parallel. A formalism capturing this must involve synchronization mechanisms or real time in some form (as for instance CSP* or Petrinet). But at the functional level, we can identify the tasks that the user can be performing, and from this functional perspective a multithreaded system will allow interleaving of those functional tasks.

Input/output re-use can be described in the state transition model at the dialog or logical level. Re-use is possible if there is a transition from a state rendering some input/output value to a state using that same value as input for another task.

* CSP = Communicating Sequential Processes, a language for parallel programming.

2.5.2 Robustness properties formalized

Deviation tolerance, or recoverability, is in state model closely related to reachability (discussed in the previous subsection) and the connectedness of the state graph. Deviation tolerance is examined by determining which states in S are error states and whether there are 'undo' paths back to 'normal' states.

Observability can be discussed using the different levels of state machines and the concept of interaction points. The machines form a hierarchy, where the functional level machine $\mathcal{M}_F$ is considered an abstraction of the dialog-level machine $\mathcal{M}_D$, which is an abstraction of the logical-level machine $\mathcal{M}_L$, etc.

So we have abstraction functions between the states of each machine:

$$Abstr_P^L : S_P \rightarrow S_L \qquad Abstr_L^D : S_L \rightarrow S_D \qquad Abstr_D^F : S_D \rightarrow S_F$$

and sequences of events (transitions) at one level can be interpreted as a single event (transition) at the level above. Figure 2.2 illustrates two of these levels of abstraction.

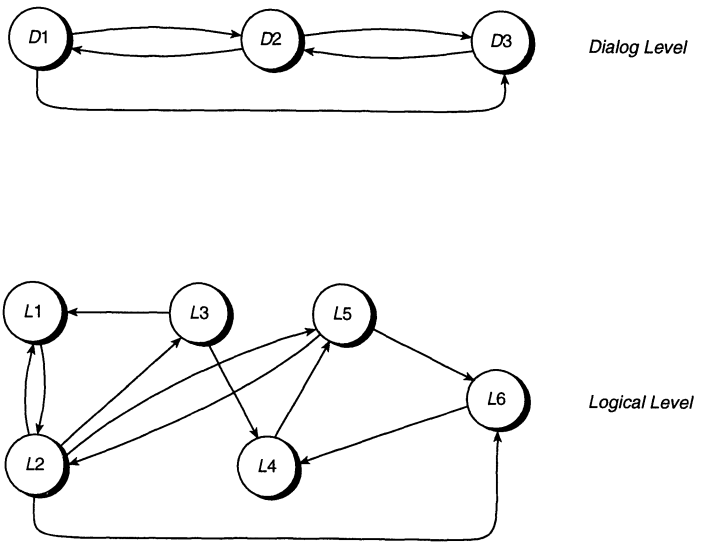

Figure 2.2 *Two levels of state machines. The dialog-level machine has three states, D1, D2 and D3, and the corresponding refined logical-level machine has six states, where D1 is the abstraction of (L1,L2), and D2 is the abstraction of (L3,L4,L5).*

A specific dialog can be described by its state trajectory, i.e. the sequence of states activated during the dialog. An *interaction point* is defined as the point at which a sequence of events at one level can be interpreted as an

event at the next higher level. (At the highest level, the functional level, all states represent interaction points.) Events between interaction points at one level do not cause state transitions (are not 'seen') at the next higher level. Figure 2.3 depicts this relationship between states and events at different levels of abstraction.

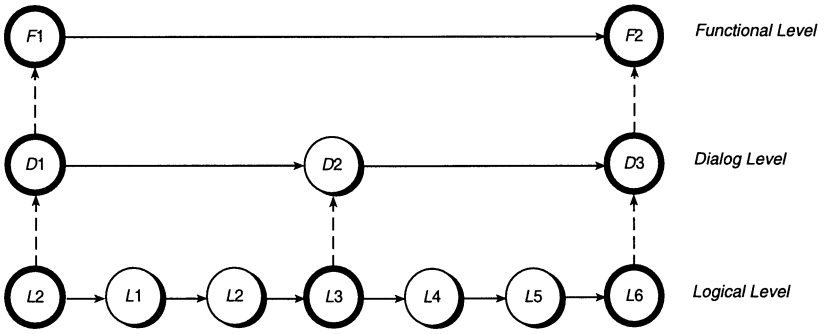

Figure 2.3 *Three levels of state machine trajectories. The interaction points are marked with thick circles.*

The user observes and interacts directly with the physical-level machine, at the 'keystroke level'. But usually only a part of the internal system state is rendered to the user. What the user can perceive is some projection, or rendering, of the physical system states. This may be described by a rendering function from the set of physical states to the set of observable states:

$$render : S_P \rightarrow Obs$$

We can now formulate *observability* properties in this multi-level model. The user only perceives information rendered at the physical level. This level contains information about the higher levels (by means of abstraction/refinement). Therefore, we can ask to what extent the observable information covers all of the dialog or functional information. If the rendering doesn't reveal all information of higher levels, we can ask whether it is possible to browse at the physical or logical level, a strategy in which physical events (such as scrolling a window) provide different observations of the logical state without changing the corresponding dialog or functional states. That is, a passive browsing strategy is one in which the logical level activity occurs between interaction points of the higher levels.

Even if the property of *predictability* is very dependent on the user's perception, we may be able to say something about it in terms of the multi-level state machine model. It is a deterministic model where, given a state $s \in S$, any event $e \in E$ leads to a unique new state $s' \in S$. (This corresponds to modeling $\rightsquigarrow$ as a function.) But the user does not always

know or understand how the system will behave. The uncertainty – or nondeterminism – arises because the system cannot tell the user everything about its state. The *render* function above defines the observable state space *Obs*, and predictability must be formulated with respect to what the user can observe.

A necessary condition for the system to be predictable from the user's perspective is that there is a function which maps events to deterministic transitions on the observable state. It is important to note here that this formulation of predictability is a necessary condition, but not a sufficient one. That the observable effects of events are deterministic does not guarantee that the users will actually perceive the determinism. The formalism can only suggest that the definition of the observable state space *Obs* be such that we can guarantee that users perceive it. Here the property of *insistence* and the persistence of information in *Obs* will play a role and influence how predictable the system occurs to the users.

2.5.3 Summary of formal model of external properties

This section has demonstrated how some of the external properties presented in this section can be better understood by attempts to model them mathematically. We do not claim to be able to provide such models for all of the properties given in this chapter, nor would such an exercise necessarily be beneficial. The formal models are only useful to the extent that they make clear distinctions between properties or suggest new properties, or if they can be used in verification of a system.

2.6 Conclusions

In this chapter, we have presented a catalog of external properties of interactive systems which characterize usability, and which can be useful in the software development process as yardsticks for quality. To summarize, the external properties fall into three categories:

Goal and Task completeness – you can do what you thought of doing.

Flexibility – you can do things in several ways.

Robustness – you can avoid doing things you wish you hadn't done.

The last two principles attempt to compensate for obvious limitations of Goal and Task Completeness, by considering user preferences and planning, their need to understand the state of a system, and likely sources of error and frustration. These extensions still fail to cover the full spectrum of requirements of the more demanding principle of 'scenario completeness' (page 27). However, there are also principles that could be adequately addressed without covering all the requirements of scenario completeness. Some notable examples of omissions are:

Learnability – the ease with which novice users achieve competent performance with new systems.

User satisfaction – how a system makes the user feel in terms of sense of accomplishment or excitement.

Rather than suggest that we could present a complete catalog of external properties to support usability, we consider the properties discussed to represent (some aspects of learning and user satisfaction apart) the current state-of-the-art. The properties are defined to form as complete and 'orthogonal' a space as possible, but a number of interdependencies and trade-offs will show up in Chapter 6 when discussing concrete examples. We have introduced a finite-state machine model to illustrate how at least some of the properties may be formalized, but formalization has not been carried through, because it is still difficult to see how to utilize it effectively in the development process.

We hope that the efforts to provide a systematic catalog that has both formal and informal rationale will encourage researchers in the area to add to and improve upon the properties identified in this chapter.

For the remainder of this book, however, we will assume only the external properties defined here and try to demonstrate how systems should be built which satisfy those properties where appropriate.