

Challenges and Solutions for Screen Reader / I.T. Interoperability

Kip Harris

IBM Human Ability and Accessibility Center

IBM Research

Abstract

The combination of Assistive Technology (AT) and Information Technology (IT) sometimes delivers a flawed end user experience. This is particularly disappointing to technologically savvy users who expect a seamless IT integration experience. The author draws from his experience as both AT and IT practitioner to present his view of the interoperability challenge within the screen reading problem domain.

Introduction

Most of us expect a “plug and play” experience when we install new application programs or devices onto our computer workstation. While such chores are usually uneventful using current technology, a Plug and play experience is not yet realized by screen reader users. Plugging a screen reader (SR) together with information technology (IT) too often results in a barely usable, or even unusable, user interface. Another aspect of this problem is the instability of many SR + IT applications when the workstation configuration changes. Seemingly small updates in one component or another (the SR, the IT, or the base operating system) can wreak havoc on a previously working application.

Colleagues have frequently asked me why SR/IT interoperability is so problematic. In my role as a professional developer of assistive technology (AT), I’ve gained much first hand insight into the technical issues in this area. I share my views of the root problems in this paper, as well as options for making progress. I believe that a better level of interoperability can be achieved than what we have today. However, I will also argue that the most interesting and important applications cannot be made interoperable without specific, custom programming to bridge AT and the IT application. The final part of this paper will discuss several options for satisfying this need.

Issues in known art

Broadly speaking, interoperability problems originate from two sources. There are engineering deficiencies in handling things that we should know how to do (“known art”), and secondly, there is a need for solutions to problems that we don’t know how to solve. We’ll begin our discussion by considering the first of these, the problems in known art.

Platform architecture and innovation

An IT application and a SR communicate through the interfaces defined by an accessibility architecture. These architectures are specific to each platform and can be considered as part of the platform API. Consequently, there is one architecture for

Windows, another for Linux, and another for Java. Each architecture defines a formal protocol that specifies the scope and the interpretation of the information that is exchanged. The only information that an assistive technology can obtain about an application is the information which fits into the protocol¹.

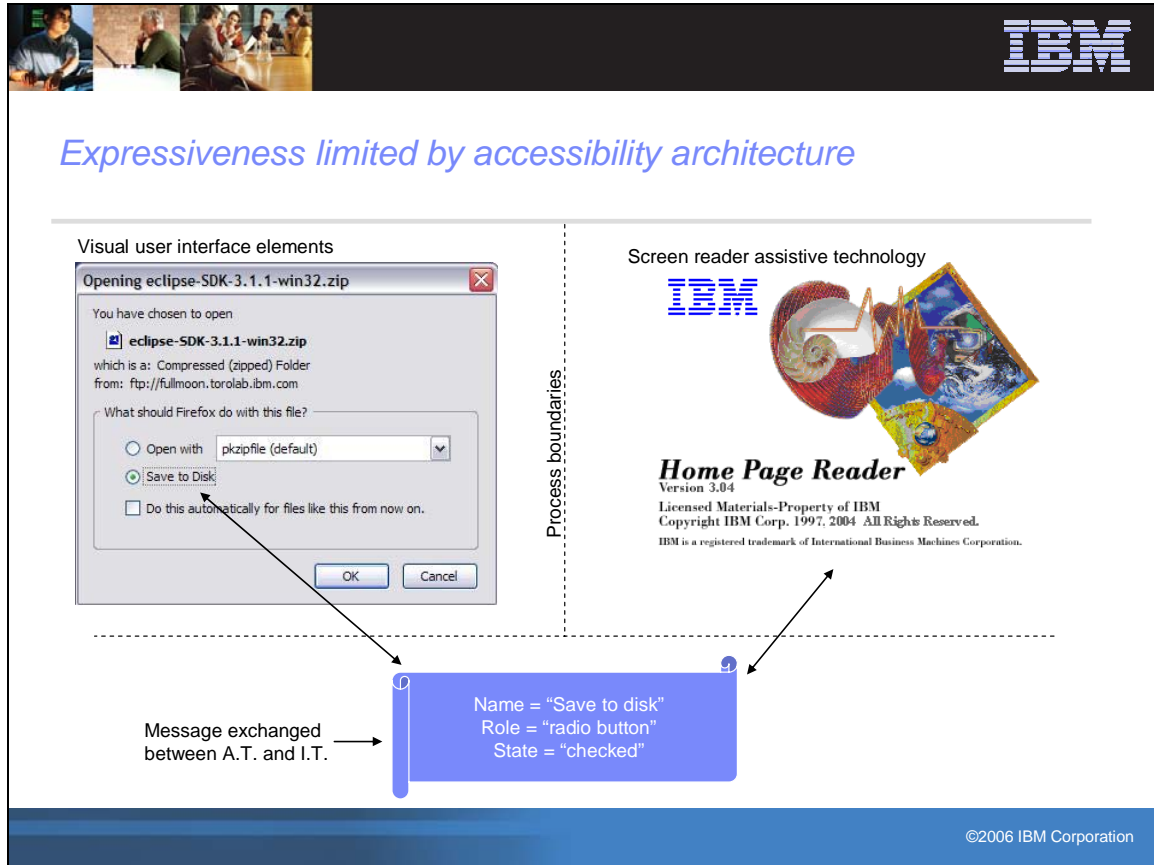


Figure 1: The screen reader and application are well isolated. The SR can only see information that is delivered within the protocol defined by the accessibility architecture.

Because the SR is limited to describing the application using only information that is available in the protocol, the architecture limits the capabilities of the SR. Figure 1 illustrates this point. A “file download” panel is shown on the left, a screen reader is drawn on the right, and the two are separated by a process boundary. The accessibility architecture will provide the screen reader with a message describing the currently focused UI element, which in this case, is a “Save to disk” radio button selection. The screen reader uses this information to tell the end user about the UI element’s name and type. However, because the color of the push button is not part of the architecture, the screen reader cannot present this information.

¹ An AT can sometimes discover additional information about a running application in the presence of a weak security framework.

Microsoft's Active Accessibility specification, which defines the accessibility architecture for Windows, provides a second illustration. The protocol designers did a good job of accommodating descriptions of simple UI elements such as checkboxes and push buttons, but did not make sufficient provisions initially for describing tables. Consequently, SR's sometimes don't recognize tables in application user interfaces.

A well designed architecture will meet most of the needs of commonly used UI's. Meeting all current needs, on the other hand, is probably not achievable, due to vast diversity of user interfaces in more sophisticated applications. Furthermore, new innovation in IT creates new UI paradigms which existing architectures are not prepared to handle. For example, consider the difficulties associated with presenting a flow chart, or a complex mathematical expression, or a logic circuit, or music. The data fields and events defined by today's accessibility architectures may simply not accommodate the information that must be represented in a new subject matter area.

The end result is that interoperability problems appear with the passage of time, and will frequently appear whenever SR's attempt to interpret more sophisticated applications. Such applications are optimized for specific subject matter areas, and frequently employ unique UI paradigms that cannot be expressed in the vocabulary provided by the platform architecture.

Compatibility of implementations

The accessibility architecture is a specification. Working software must be implemented from the specification, by each AT vendor and each application². This task is analogous to the challenge faced by vendors who create networking software: each vendor constructs a unique implementation of a published specification. No matter how well the specification is written, each disparate implementation team will interpret certain operational descriptions in slightly different ways. Interoperability problems result when two such differing interpretations attempt to interoperate.

For example, in the Windows MSAA environment, the text object has a name and a value field. Some application vendors expose static text in the name field, while others use value for the same purpose. Likewise, UI objects are responsible for firing MSAA events when something changes. We have found pervasive problems with inconsistent event behaviors among the UI elements delivered by various vendors.

Other software technologies solve this class of problems by providing more guidance in the architecture / protocol specification, and by devoting significant resources to interoperability testing. Constructing a reference implementation is a formidable task indeed. The correct behavior of at least tens, and perhaps hundreds of common UI elements would need to be documented and standardized. Creative UI designers often create complex widgets by composing simpler widgets into more complex objects³. Such UI elements would need to be included in the effort, and consequently, the scope of the

² It is more accurate to say that the specification must be implemented by each UI element author, rather than by each application. Examples: custom controls, PDF, Flash, SWT.

³ A "calendar" widget is one example of a complex UI object which is composed of simpler, more primitive UI elements

problem is potentially unbounded. Nonetheless, this effort might well be manageable for the small subset of the most commonly used, simple UI elements.

Fragile heuristics

SR's perform their task by monitoring the visual UI and then creating textual descriptions for the end user. SR's rely heavily on heuristic algorithms to infer information that is missing from the architected accessibility interfaces, or where the information in the interfaces might be ambiguous. These heuristics are frequently complex: A representative example of a rule which is associated with a "value change" event might look like this:

```
If
  (a Value Change event has just been delivered) and
  (the event was generated by a text box control) and
  (this is a single line text box) and
  (the text box is associated with a "spinner" control) and
  (a Key Up or Key Down keystroke
      was the last keystroke observed) and
  (the Key Up/Down timestamp is within 150 milliseconds
      of this time stamp)
Then
```

It is not unusual for heuristics to be dependent on a particular sequence of events, or particular spatial arrangements. I use the term event signature to describe the collection of events and attributes that a SR can observe, in order to recognize that a particular action has occurred on a specific type of UI element. The example just presented is the event signature of an option selection change in a Combo Box. An enormous amount of development time is typically spent in discovering the signatures and heuristic rules that will make a SR "work" for a given use case in a given application.

Heuristics are "fragile" because they are commonly based on the incidental, undocumented, internal behavior of particular software components at a given point in time. When a component is patched at a later point in time, the internal behavior frequently changes due to arbitrary internal implementation decisions. However, if a SR's heuristic is an observer of such a component, it will break. Consequently, we frequently see SR / IT solutions break when patches or point releases are introduced into any of the components of the solution – operating system, IT application, or SR.

Applying best practices within the known art

I've described three troublesome areas which cause SR/IT interoperability problems. The root causes of these problems are not unique to the Assistive Technology domain; in fact, these are well known engineering problems in other software disciplines. I believe that the solutions to these classes of problems are known art, and if so, it suggests that we look to existing best practices for guidance.

The first problem area is limitations in the expressiveness of the platform architecture. This is not entirely a problem in known art, and we will return to this issue in great depth

in the latter part of this paper. However, there certainly are many things in this area that we know how to do better. In particular, when a new technology is created, the architects must include a programming model for accessibility [Brunet 2005]. Even in recent times, new technologies are sometimes introduced (e.g. Flash) without any platform architecture support in their initial releases. Once an architecture is defined, the technology owners must remain actively engaged, and evolve the architecture to accommodate community needs and innovation. For example, where an existing architecture is missing an important UI paradigm, the technology owners need to take remedial action and extend the architecture.

We looked at problems which originate from differing interpretations or incompatible implementations of the architectural specification. This is a classic and very well understood IT problem. In other software engineering disciplines, this class of problem is addressed by providing more detailed specification guidance, as well as the development of reference implementations and tools for more comprehensive interoperability testing.

The need for heuristics has its roots in both of these first two problem areas. The problems that result from the use of heuristics might be mitigated by the practices which address architectural shortcomings (which would reduce reliance on heuristics).

Current best practices require a manual test of an IT application with the complete IT+SR solution. All use cases must be exercised by a person who does not have access the video display, and the application must be operated exclusively through the UI provided by the SR. This level of development investment often results in a positive end user experience.

This level of test can be expensive. Furthermore, each IT+SR pair is a unique combination from a verification perspective. A successful test of one IT+SR configuration assures us that we've met a level of compliance. However, it does not assure us that a similar solution which substitutes a comparable product for either the IT or the SR will perform equally well.

Obviously, it would be far preferable to develop and test just a single IT+SR combination. Additional guidance in specification implementation, reference implementations, and some tools for interoperability testing may promote this desirable "loose coupling" between IT and SR components. Furthermore, the existing bits and pieces of insight regarding best practices implementation, which might improve the correct and expected use of the data fields and events, is currently distributed among the disparate parties (platform vendors, AT vendors, and application developers). Discovering these low level insights is an ad-hoc activity which is repeated by each development team. Concerted industry effort and leaderships by technology owners is needed to make progress toward more systematic engineering practices.

Interoperability for unique application semantics

The problems that we've explored so far are primarily engineering issues that can perhaps be corrected with better designs, implementations, and practices. We'll now turn our attention to a final problem area that is beyond the realm of known art, and is much more difficult to remedy. This is the realm of human perception and cognitive processing of visual information, which is central to human-computer interaction (HCI).

A visual UI presents visual elements on a display to convey some mental model to an end user. The UI elements themselves have little meaning; we depend on human perception to interpret the visual scene and infer the meaning. The intended meaning of a visual presentation is called the *semantics*.

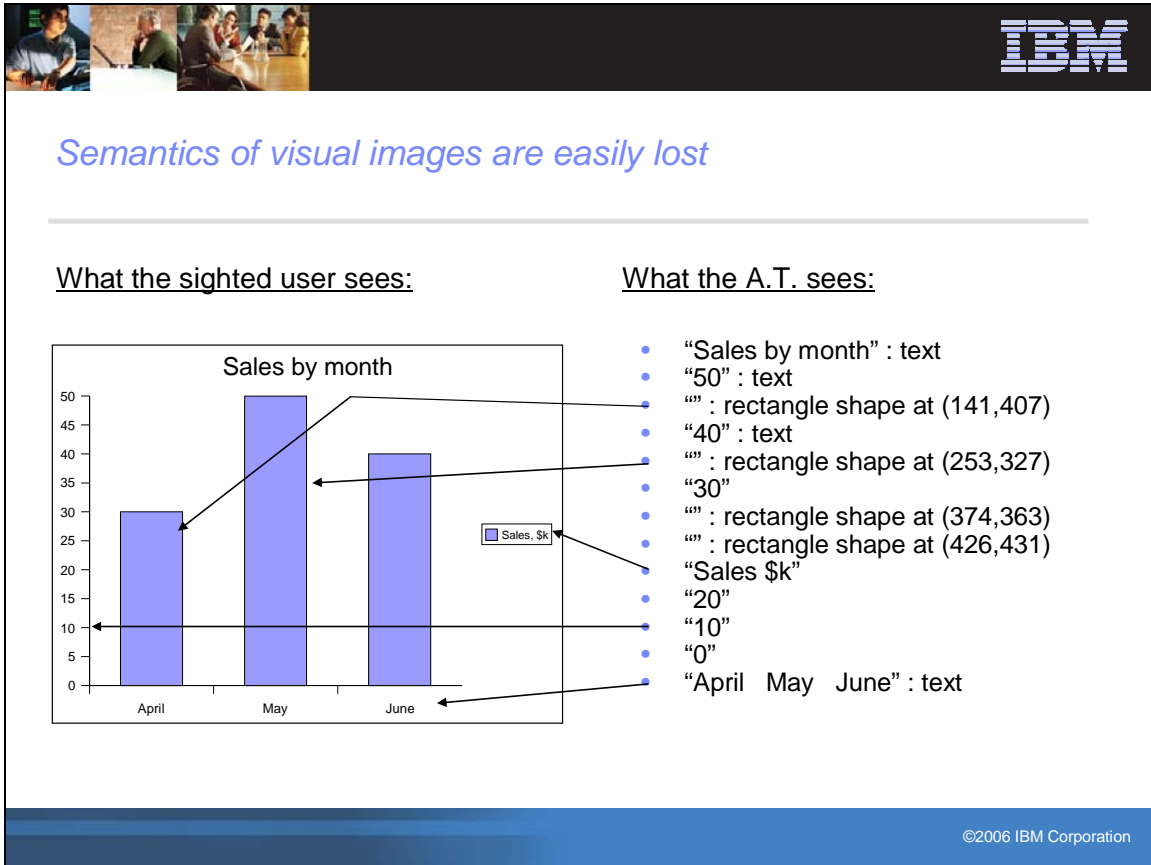


Figure 2: A picture which depicts a common bar chart. The visual appearance of the bar chart is shown on the left hand side, while the "view" of the bar chart which is available to an AT is on the right. Arrows indicate the correspondence between messages seen by the AT and visual UI elements.

For example, consider a graphical user interface that presents a bar-chart from a spreadsheet application, such as the graphic depicted in Figure 2. The visual display is an arrangement of shapes and labels. An SR might inspect this scene, but can only describe the locations and dimensions of the rectangular figures. The SR cannot construct a meaningful text description without some built-in understanding of the semantics of a bar-chart. Access to the information in the spreadsheet behind the bar chart is needed by the SR to construct a truly usable, non-visual UI.

Programs such as Microsoft Excel are well known to ATs, and some have specific algorithms that apply Excel's semantics. However, in the general case, the AT does not have built-in knowledge about an arbitrary application (imagine a web based spreadsheet

program), and must instead interpret the visual scene from only the visual elements on the display.

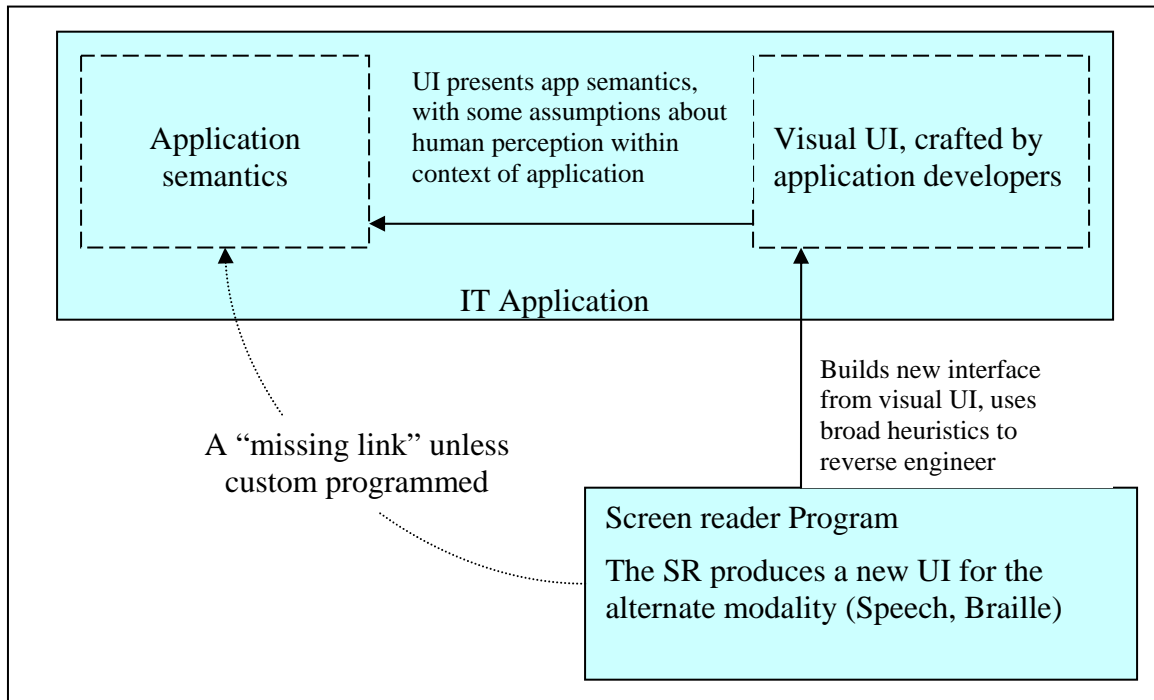


Figure 3: AT’s attempt to infer semantics from UI elements. The SR does not have a direct connection to application model unless “bridge” code is specifically developed for that application. This is highly error prone where no such custom connection is provided.

Figure 3 illustrates the situation. Application developers, with deep understanding of the application semantics, craft a visual UI. After the application is deployed, the end user with sight develops an understanding of the underlying semantics through human perception and interpretation of the visual scene.

The AT, on the other hand, uses computer algorithms to create a new UI, based on a machine scan of the visual elements. These algorithms perform a modality transformation from two dimensional visual scenes to one dimensional text strings. There is no “understanding” of the underlying application semantics⁴. In short, we have the AT describing the video scene, rather than the AT describing the application. It should be no surprise that the SR’s machine translation loses much of the fidelity to the original application semantics⁵.

⁴ A SR may recognize and understand Excel and a number of other well known programs, but in the general case, the AT knows nothing about the semantics of an arbitrarily chosen application. Instead, the AT is constructing a UI “on the fly” using only the arrangement of visual elements on the screen.

⁵ The problem of capturing the underlying semantics is made even more difficult when one considers that both application UI and human interpretation are sensitive to the context of the problem domain. For example, UI elements are arranged (and interpreted) quite differently in a spreadsheet, as compared to how a UI elements are interpreted in flowchart. Product developers have spent millions of man-hours creating

Toward the encoding of application semantics

I have just argued that the most fundamental problem in SR / IT compatibility is the loss of application semantics in the course of machine translation. Better capture and translation of application semantics will require that they be encoded in machine readable syntax. If such an encoding is achieved, the SR can interpret and translate the information while preserving the meaning that was intended by the original application authors.

I will present several alternatives for accomplishing exactly this. Each approach represents a different mix of prerequisite innovation, cost, and reusability of solution. In particular, we will consider:

- How many UI's must be authored? Traditionally, all applications write a visual UI. Must I write one additional UI for text? One additional UI for each SR?
- Who creates the non-visual UI's? What skills are needed?
- Must I, as an application vendor, interlock with each SR vendor as I develop a new product?

A definition language for user interface components

Today, if an application presents a unique UI that does not conform to common usage patterns, the application vendor must arrange for the development of SR updates to accommodate the application's uniqueness. These updates will understand how to deal with the application's unique semantics. The end user must install these SR customizations before he can operate the new application. Clearly, this is a burden on all stakeholders: application developer, SR vendors, and end user.

Ideally, the application developer would write a single user interface, and release the product without any interlock with SR vendors. The SR would connect to the new application, automatically inspect the application for any special instructions about handling unique semantics, and then present a textual version of the user interface which satisfies the end user. How might we programmatically describe a UI so that an SR could automatically interpret it?

Recent developments in Web UI's show us one possible approach. Some Web applications are using innovative UI widgets which are built from DHTML. These widgets look and feel like familiar controls, but are internally composed from simple graphical elements which are meaningless to SR's. A semantic definition language has been proposed by the W3 Protocols and Formats working group to remedy the problem [3]. The application encodes the intended interpretation of the new widget using XML-based Resource Definition Framework (RDF) technology [4]. When both application and

visual presentations which are customized for the semantics of a particular problem domain; this results in the superior "ease of use" which distinguishes one product from another. The visual presentation works only because there is a certain amount of human interpretation of the visual arrangement. We humans interpret a visual presentation in the context of the current subject matter, which to some degree relies on real-world experience, judgment, and common sense. These interpretative abilities are absent from the general purpose, machine executable algorithms that the SR executes.

SR use this technology, a SR can build a textual user interface for an otherwise unfamiliar UI.

The DHTML+RDF proposal is sufficiently descriptive to identify well-known controls. For example, a DHTML-based tree view control can be described in this scheme⁶. Can the RDF approach be extended to provide for SR / IT interoperability when the control (or panel) is not well-known? Might some grammar be developed that is analogous to the Web Services Definition Language (WSDL) [5] in the Web services domain, where the description language is sufficiently rich to describe services that have not yet been invented? In the Web services domain, the scope of the problem is limited to the description of message structures. In the Human-Computer Interface domain, there are far more demanding requirements on the description language, because it must be able to describe both behavior and intended meaning, in addition to structure.

The designers of such a “UI Description Language” will have a difficult challenge indeed. The language would have to describe the structure, behavior, and intended interpretation of innovative user interfaces which do not fit within well known paradigms. Because this task is essentially an encoding of arbitrary semantic meaning within an unconstrained problem domain, I will conjecture that this is the same class of problem as automatic machine translation of natural language.

An enormous amount of research and development has been expended on natural language translation, and as anyone who has used such a service knows, much remains to be done. Likewise, a UI Description Language which accommodates non-well known UI patterns is a significant research problem and far beyond known art. However, if some innovation could solve this problem, it would be possible to author a single UI in the UI Description Language, and then generate multiple interfaces for the various modalities from that single definition.

Screen reader customization and scripting

Self-describing user interfaces, through either RDF or some other future user interface definition language, is a future technology. Current technology depends on “scripting” features which are available in the higher performing SR products. Scripting gives us a mechanism for extending the capabilities of the SR by plugging additional algorithms into the Reader. We encode an application’s unique semantics in the scripts which we write.

Ideally, script development would be performed by the same UI team that develops the visual interface. This is the team that has the best understanding of the application model and of any UI innovations. It is the team with the best understanding of the technical aspects of the behavior of the new interface. If the application team writes all the interfaces (visual, aural, and Braille), then we eliminate the dependencies and need for coordination between separate IT and AT commercial enterprises. Finally, both visual and non-visual versions of the UI are then simultaneously available, and deployed by the application developer.

⁶ It is the widget-implementer’s responsibility to emulate both the structure and behavior of the known control, so we are presuming that both of these are well-known, if not well-documented.

Alternatively, one could point out that extending the SR requires specialized, scarce skills. Also, if the SR does not provide sufficient features for scripting a certain use case, then the customization must be done by the SR vendor himself in the base product.

In summary, under this approach, either the application developer or his proxy must write a plug-in module for the unique UI components of his application. This is only possible when the scripting provides sufficient features to support the innovation. Otherwise, the SR itself must be enhanced to support the new UI component. Either way, an additional, unique UI is being authored for each SR that supports the application: we write “1 + n” unique user interfaces.

Self-voicing

A third option to incorporating application semantics into additional user interfaces is to integrate SR features into the application. This technique is called self-voicing, which alludes to the fact that the application typically provides synthesized speech as an alternate output modality. With traditional screen readers, the technology which provides the alternate input or output modes is bundled into the screen reader. In a self-voicing environment, the I/O mechanisms for the alternate modality (typically speech) are integrated into the application, so the alternate I/O modality is available even if an SR product is not installed. There is no need to integrate the application with other products.

Self-voicing technology has been developed for Java applications, and is perhaps most visible today in some of the newer pervasive devices which exploit speech-enabled browsers. The X+V technology, in particular, provides an environment in which both visually rendered HTML and aurally rendered VHTML can provide a “side by side” interface: one visual and tactile, the other aural (using synthesized speech and voice recognition). The visual and aural interfaces are each a distinct development effort to implement, but if done properly, there may not be a need for further AT integration. So we write twice (once for each modality).

Summary and conclusion

I’ve argued that interoperability problems are either engineering issues in known art, or alternatively, result from the inherent fallibility of machine translation from a visual mode to text, in absence of some encoding of the application’s unique semantics. Engineering issues can be addressed by well known processes which are actively employed in other IT disciplines. In particular, the industry would benefit from availability of reference implementations, improved guidance on specification implementation, and interoperability testing.

The issues associated with machine translation of unique application semantics, where the semantics are inferred by the SR based on visual arrangements of UI elements, is much more difficult to solve.

The premier solution would provide for a “write once” UI which could be interpreted by an SR without requiring any coordination between IT and AT enterprises. A UI definition language would be needed, in which we could encode application-unique structure, behavior, and intended meaning of a UI object. I’ve conjectured that creating such a

language is roughly as difficult as machine translation of natural languages and is far beyond known art.

A more immediate, tactical solution is found in the scripting capabilities provided by some SR's for extending the reader's capabilities. When this is available, we can manually encode application semantics into the scripts. These capabilities are available today, and are the most practical avenue for solving interoperability problems. That is, recognize that you must create multiple user interfaces for the multiple modalities, and plan for it. However, in taking this approach, you accept that you write an additional UI for each AT with which an IT solution must interoperate. This approach is not an option where the AT is not available for modification, or where existing customization facilities are inadequate.

Self-voiced applications can drive both visual and aural modalities without any need for a SR. However, this too has its drawbacks. Each application is re-implementing the same functions, in keystroke command definitions, audio controls, and preferences settings. Each application vendor would likely implement these features in a different way, which greatly diminishes overall usability. There is also the problem of resource sharing and coordination among the multiple ATs. From an application implementation perspective, we are writing twice, plus the additional burden of coordinating all the various ATs.

Emergence of pervasive technologies which embrace multimodal interfaces may provide the most promising long-term approach. New usage patterns involving mobile users and pervasive devices are making hands-free / eyes-free / multi-modal operation a requirement of the mainstream. New technologies such as X+V, voice-enabled browsers such as Opera and NetFront, and the speech capabilities which are expected to be available in the next major Windows client release, will encourage the construction of user interfaces that can be designed to be equally usable from various I/O modalities. However, even here we must note a caveat. Pervasive applications are usually designed without a requirement for complete operation through just one modality. Instead, there is typically an assumption, for example, that the user can see the device's display.

The need for unique UI development for each modality is common to all approaches. Product developers will continue to be required to deliver usable UI for each I/O modality, and for this, there is no "silver bullet."

References

1. Brunet, P.B., et al. "Accessibility requirements for systems design to accommodate people with vision impairments". IBM Systems Journal, Vol. 44, No. 3, 2005. On the Web at <http://www.research.ibm.com/journal/sj/443/brunet.html>.
2. Microsoft Active Accessibility Version 2.0. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/msaa/msaastart_9w2t.asp
3. W3 Protocols and Formats, 2005 "Dynamic Accessible Web Content Roadmap". <http://www.w3.org/WAI/PF/roadmap/>.
4. W3 Resource Description Framework (RDF). <http://www.w3.org/RDF/>.
5. W3 Web Services Description Language (WSDL) 1.1. <http://www.w3.org/TR/wsdl>.