COMPUTING PRACTICES

Building Task-Based User Interfaces with ANSI/CEA-2018

Charles Rich, *Worcester Polytechnic Institute*

The recently approved ANSI/CEA-2018 standard is motivated by the current usability crisis in computer-controlled electronic products. The standard facilitates a new user interface design methodology that uses a task model at runtime to guide users.

According to a recent study,[1] half of all reportedly malfunctioning consumer electronics (CE) products returned to stores are in full working order—customers just couldn't figure out how to operate them. The trouble started in the 1980s with the infamous blinking VCR clocks and has gotten steadily worse as virtually everything we buy these days has a computer chip controlling it.

The current usability crisis has at least two aspects: *complexity* and *inconsistency*. First, computer control has made it easy—perhaps too easy—to add features to products, and their resulting complexity has exceeded the capacity of current user interface (UI) designs for users to operate them intuitively. Second, there is little or no UI consistency, either between devices with similar functions or between devices from the same manufacturer.

Standardization would appear to be a logical solution to the inconsistency problem. Unfortunately, CE manufacturers adamantly resist any attempt to standardize UIs across devices with similar functions. They believe that their UIs' appearance and operational details are crucial to brand identification and product differentiation, and fear that UI standardization is the first step toward commoditization, which may be good for consumers but drives down profit margins.

Standardization of UIs across devices from the same manufacturer doesn't meet with this resistance but is much more limited in its potential advantages. For example, a typical consumer only owns one model of radio and thus wouldn't benefit from consistency across all models from a given manufacturer, and there is limited opportunity for interface consistency between a radio and, say, a microwave oven.

The recently approved ANSI/CEA-2018 standard[2] aims to directly address the complexity problem by facilitating a new kind of user interaction without trying to standardize the appearance of the UI per se. This tricky, but necessary, strategy could significantly improve the usability of computer-controlled CE products and software interfaces in general.

TASK-BASED USER INTERFACES

One way to deal with device complexity is to eliminate as many features as possible. Apple has been notable for doing this. However, some advanced features enabled by computer control, such as increased customization and programmability, are truly useful. But they're also inherently complicated, and few users read or can thoroughly understand the relevant manuals or documentation—if they can even find them!

This unavoidable complexity demands a shift in product design methodology. In addition to performing its primary function, such as playing a DVD or heating food, a computer-controlled CE device should also actively help the user learn how to operate it via a task-based UI.

Architecture

Figure 1 shows the architecture of a task-based UI.

The architecture's most important feature is that it uses a task model description at runtime to guide the user. In traditional UI design, formal task models are used only at design time, if at all, and then discarded. (For an interesting experiment in combining model-based design with task-based user interfaces, see the BATS[3] system.)

The next most important architectural feature is the decomposition of the task-based UI into two components: a generic task engine and an application-specific UI. (The task model description is also, of course, application specific.) It is this decomposition that has mitigated industry resistance to ANSI/CEA-2018, because it standardizes only the task engine and the task model description language, not the UI. Another advantage is that the effort of building a task engine can be amortized over many different applications.

The task engine's basic functions are to load and validate a task model description, and to maintain a representation of the current status of the user's tasks. The task engine doesn't interact directly with the user. If it needs information from the user, it sends a request to the UI. How this request is presented to the user depends on the specific UI—for example, it may be graphical, textual, spoken, and so on.

Communication between the task-based UI and the controlled device(s), called *grounding*, is implemented in ANSI/CEA-2018 using JavaScript. Because it has been implemented on a wide range of platforms, JavaScript provides maximum flexibility for grounding to different networks and other infrastructure technologies.

The architecture also allows for the possibility that the user will perform manual operations on the device, such as loading a DVD. In fact, some devices may only be manually operable—for example, because they're not connected to the network—in which case the UI would simply provide instructions to the user indicating what to do.

Finally, the architecture is functional, not physical. For example, the task-based UI may run as a computational process on the same hardware as the device or have separate hardware. The UI display, if any, may use the device hardware—for example, if the device is a TV—or it may use a shared remote display accessed through the network. Similarly, the task model description may be uploaded from the device, downloaded from the Internet, provided on a USB stick that comes with the device, and so on.

Task guidance

One way to think about the guidance that a task-based UI provides is in terms of the questions a user can ask the system, such as
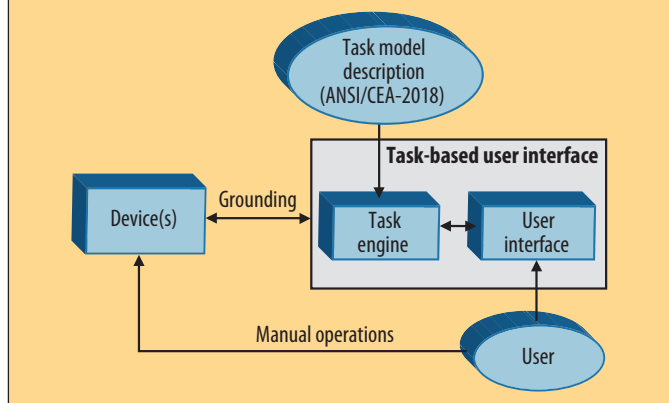
- What can/should I do next?
- How do I do <task>?

Figure 1. Functional architecture for a task-based user interface and its relationship to device(s) and user. Unlike traditional UIs, in which formal task models are used only at design time, if at all, and then discarded, a task-based UI uses a task model description at runtime to guide the user. ANSI/CEA-2018 standardizes only the task engine and the task model description language, not the UI.

Figure 2. Screenshot of a task-based UI for a DVD player built according to the architecture in Figure 1. The top half of the screen is a generic "chat window" for guidance; the bottom half is an application-specific direct-manipulation GUI.

- When should I do <task>?
- Why did you do <task>?
- What are the inputs/outputs of <task>?
- Did <task> succeed?

The question of what to do next is at the heart of task guidance. The signature experience of the usability crisis is facing a bewildering array of buttons, sliders, and so on and not knowing how to respond. In fact, a "What next?" capability should be considered as indispensable to good UI design as the "undo" capability currently is.[4]

A system implemented according to the architecture in Figure 1 can provide the answers to these user questions. Figure 2 is a screenshot of DiamondHelp, a DVD recorder application[5] built according to this architecture

```
<taskModel about="urn:computer.org:cetask:library"          <task id="LookupInCatalog">
        xmlns="http://ce.org/cea-2018"                        <input name="book" type="Book"/>
                                                               <output name="location" type="string"/>
  <task id="Borrow">                                           <postcondition>
    <input name="book" type="Book"/>                             $this.location != undefined
                                                               </postcondition>
    <subtasks id="borrowing">                                  <script>
      <step name="go" task="GoToLibrary"/>                       $this.location = lookup($this.book);
      <step name="choose" task="ChooseBook"/>                  </script>
      <step name="check" task="CheckOut"/>                    </task>
      <binding slot="$choose.input"
              value="$this.book"/>                           <task id="TakeFromShelf">
      <binding slot="$check.book"                              <input name="book" type="Book"/>
              value="$choose.output"/>                         <input name="location" type="string"/>
    </subtasks>                                               </task>
  </task>
                                                             <task id="UseSearchEngine">
  <task id="GoToLibrary"/>                                     <input name="query" type="string"/>
                                                               <output name="book" type="Book"/>
  <task id="ChooseBook">                                       <output name="location" type="string"/>
    <input name="input" type="Book"/>                          <postcondition>
    <output name="output" type="Book"/>                          $this.book != undefined
                                                               </postcondition>
    <subtasks id="initial">                                    <script>
      <step name="lookup" task="LookupInCatalog"/>               $this.book = search($this.query);
      <step name="take" task="TakeFromShelf"/>                   if ( $this.book != undefined )
      <binding slot="$lookup.book"                                 $this.location = lookup($this.book);
              value="$this.input"/>                            </script>
      <binding slot="$take.book"                              </task>
              value="$this.input"/>
      <binding slot="$take.location"                          <task id="CheckOut">
              value="$lookup.location"/>                        <input name="book" type="Book"/>
      <binding slot="$this.output"                              <script>
              value="$this.input"/>                              print("["+$this.book+" checked out!]");
    </subtasks>                                                 </script>
                                                              </task>
    <subtasks id="alternative">
      <step name="search" task="UseSearchEngine"/>            <script init="true">
      <step name="take" task="TakeFromShelf"/>                  <!- insert JavaScript from Figure 5 ->
      <applicable>                                             </script>
        $this.success == false
      </applicable>                                          </taskModel>
      <binding slot="$take.book"
              value="$search.book"/>
      <binding slot="$take.location"
              value="$search.location"/>
      <binding slot="$this.output"
              value="$search.book"/>
    </subtasks>
  </task>
```

Figure 3. Complete ANSI/CEA-2018 task model description for borrowing a book from the library. This XML document defines seven task classes. The top-level task, Borrow, is decomposed into subtasks GoToLibrary, ChooseBook, and CheckOut. ChooseBook is further decomposed either into LookupInCatalog followed by TakeFromShelf or UseSearchEngine followed by TakeFromShelf. All the other task classes are primitive.

using the Collagen task engine[4] and its associated task model description language, which preceded and inspired ANSI/CEA-2018. The top half of the screen is a generic "chat window" for guidance, while the bottom half is an application-specific direct-manipulation GUI.

TASK MODELING

Task modeling—the process of developing a task model description for a particular domain—is a well-known technique in both UI design and artificial intelligence (AI).

A reasonable question then is why yet another task model formalism is needed.

The first reason is that ANSI/CEA-2018 is a standard, which makes it possible for devices from different manufacturers to interoperate. In modern offices, factories, laboratories, and homes, computer-controlled devices are, to a rapidly increasing extent, connected via networks. While standards, such as universal plug and play (UPnP; www.upnp.org) for CE devices and the Laboratory Equipment Control Interface Specification (www.lecis.org) for

laboratory instruments, already exist for remote network control of individual devices, the real payoff of networking lies in supporting high-level integrated services that involve multiple steps on multiple devices, such as gathering, analyzing, and storing data in a laboratory or copying a movie from videotape to DVD in a home entertainment center.

From the user's point of view, each of these examples is conceptually a single high-level task. Unfortunately, especially if the devices involved are from different manufacturers, users currently need to learn the different operational details of each device to carry out the whole task. A single standard spanning from high-level tasks down to the device level is needed for unified support of such multidevice tasks.

The second reason for ANSI/CEA-2018 is that it distills the key features of task models in a way that enables practical runtime use in CE and similar low-cost applications. In task modeling, as in all formalisms, there is a tradeoff between expressive power and computational tractability.

The graphical task analysis formalisms commonly used in UI design, such as ConcurTaskTrees,[6] and the hierarchical task network formalisms commonly used in AI, such as SIPE,[7] are more expressive than ANSI/CEA-2018. For example, ANSI/CEA-2018 doesn't explicitly represent parallelism or include a rich model of time intervals.

However, these more expressive graphical and AI task modeling formalisms are also more computationally expensive to reason about automatically. This isn't a problem for UI design tools, because the automatic reasoning occurs only at design time. Similarly, this isn't a problem in AI, because the computing resources typically available for AI applications are much greater than those currently available in CE devices.

Finally, compared to a very broad model-based UI design formalism such as the User Interface Extensible Markup Language (www.usixml.org), which is intended to cover everything from layout to dialog control and includes a task-modeling component based on ConcurTaskTrees, ANSI/CEA-2018 has a much narrower and more limited focus.

TASK MODEL DESCRIPTION

Figure 3 shows a complete, self-contained ANSI/CEA-2018 task model description for a simple example task:
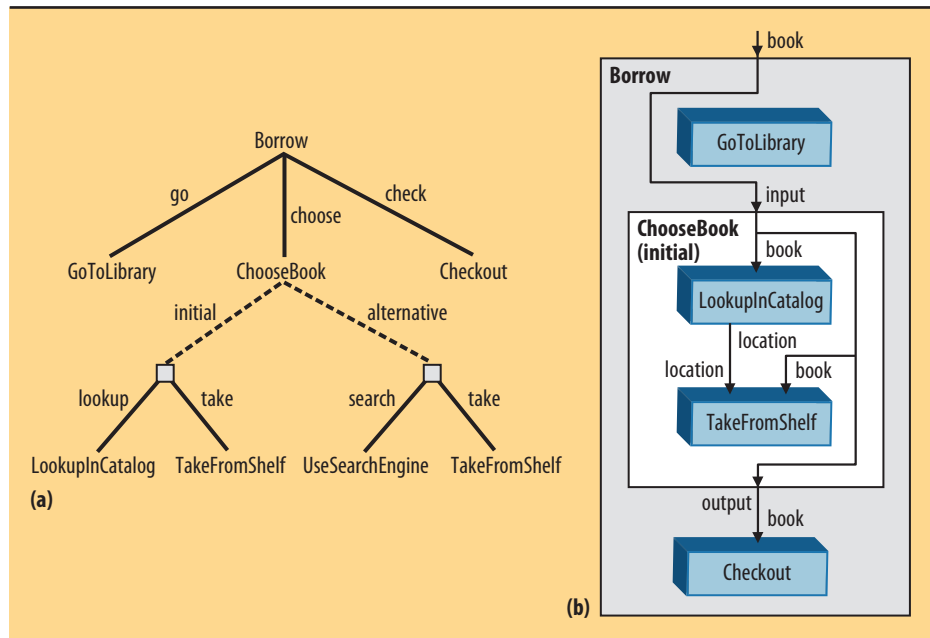

Figure 4. Informal graphical presentations of the task model structure in Figure 3. (a) Task decomposition tree, with dotted lines indicating decomposition choices. (b) Data flow between task inputs and outputs follows selection of initial decomposition.

borrowing a book from the library. The first thing to observe about ANSI/CEA-2018 is that it isn't a graphical task modeling formalism. The primary purpose of the standard isn't to help human designers visualize and formalize the task structure of a new domain, but rather to specify the syntax and semantics of an XML document that a device will interpret at runtime to guide the user.

This isn't to say that graphical visualization is unimportant—quite the opposite. Humans can't use complex formalisms without making diagrams. For example, Figure 4 contains helpful graphical presentations of the task model description in Figure 3. However, these diagrams aren't a formal part of the standard; they're just an informal aid to understanding. In the future, it may be useful to develop a graphical tool for ANSI/CEA-2018 based on similar diagrams.

The standard's key expressive features include tasks, input and output parameters, preconditions and postconditions, grounding, task decomposition, temporal order, data flow, and applicability conditions.

Tasks

The concept of tasks, which might also be called activities, goals, jobs, or actions, is at the heart of the ANSI/CEA-2018 standard. Task examples in the CE domain include copying a videotape to a DVD, watching a recorded TV episode, and turning off the room lights.

Task characteristics. Tasks vary widely in their time extent: Some occur over minutes or hours—for example, watching a recorded TV episode; some are effectively

```
function Book (author, title) {              var database = [
    this.author = author;                        { query: "Heinlein", book: stranger },
    this.title = title;                          { query: "Sawyer", book: mindscan },
}                                                { query: "Vinge", book: fire } ];

Book.prototype.toString =                    function lookup (book) {
    function () { return this.title; }          for (i = 0; i < 3; i++)
                                                    if (catalog[i].book.author == book.author
var stranger = new Book("Heinlein",                      && catalog[i].book.title == book.title)
            "Stranger in a Strange Land"),              return catalog[i].location;
    fire = newBook("Vinge",                  }
          "A Fire Upon the Deep"),
    mindscan = newBook("Sawyer","Mindscan"); function search (query) {
                                                for (i = 0; i < 3; i++)
var catalog = [                                    if (database[i].query == query)
    { book: stranger, location: "Shelf1" },            return database[i].book;
    { book: mindscan, location: "Shelf2" },  }
    { book: fire, location: "Shelf3" } ];
```

Figure 5. Initialization JavaScript to be inserted into the task model description of Figure 3. This code defines the Book data type and associated functions used in the task model's conditions and grounding scripts. First, a constructor and printing function for Book are defined. Next, three global variables are initialized to implement a tiny card catalog and searchable database for testing purposes. Finally, lookup and search functions are provided for the catalog and the database, respectively.

instantaneous—for example, turning off the room lights; and some have an unbounded time extent—for example, a weekly teleconference.

Tasks typically involve both human participants—as requesters, beneficiaries, or performers of the task—and electronic devices. Some tasks, such as providing a fingerprint for identification, can be performed only by a human being; others, such as displaying a video, can be performed only by an electronic device; and yet others, such as opening a DVD drawer, can be performed by either depending on the circumstances.

Tasks also vary along an abstraction spectrum from high-level—closer to the user's intent and natural way of communicating—to low-level—closer to the primitive controls of a particular device. Watching a recorded TV episode is a fairly high-level task, while pressing the power button on a DVD player is a very low-level task. Tasks are also more or less abstract by virtue of being parameterized.

Deciding on the appropriate task granularity and parameterization is a key part of the modeling process and depends on both the application and the desired level of task guidance. Further, whereas some other formalisms use different representations for high-level tasks (goals) versus low-level tasks (actions), ANSI/CEA-2018 uses a single uniform task representation at all levels of abstraction, which provides more flexibility to adjust the level of granularity in developing models.

Task classes and instances. A task model defines task classes. A task instance corresponds to an actual or hypothetical occurrence of a task. Pressing the power button on a DVD player is an example of a task class. Parameters of this class might include who pressed the button, which DVD player was involved, and when the action occurred. Thus, David Smith pressing the power button on the DVD

player in his living room at 3:15 pm on 1 January 2006 is an instance of this class. A task engine manipulates both classes and instances.

The task model description in Figure 3 defines seven task classes, from the high-level task, Borrow (borrowing a book from the library), to low-level tasks, such as Take-FromShelf (taking a book from a shelf). Obviously, what is high- and low-level is relative to the overall model's level of detail, and is an important task-modeling decision.

Input and output parameters

The *input parameters* of a task class should include all data that affects the execution of task instances, while the *output parameters* should include all data that is modified or created during execution of task instances. For example, the LookupInCatalog task takes a book as input and returns a location string as output. Input and output parameter types may include new application-specific data types defined in JavaScript, such as Book.

Pre- and postconditions

A task's *precondition* is a partial Boolean function that tests whether it's appropriate to perform the task at the moment. A task's *postcondition* is a partial Boolean function that tests whether a just-executed task was successful. Both preconditions and postconditions default to unknown.

Pre- and postconditions are defined using JavaScript expressions. The environment in which these expressions are evaluated includes all the functions and variables defined in the task model initialization script, like that shown in Figure 5, plus a special binding of the variable $this to the current task instance. For example, the postcondition of LookupInCatalog specifies that the task succeeds if and only if the location output is defined.

Grounding

Primitive task types—those without decompositions—may be associated with a grounding script, which is a JavaScript program evaluated in the same environment as conditions. These programs typically connect to an underlying device, cause it to perform the appropriate action, and then report the results by setting the output slots of the current task instance. For example, the grounding script for LookupInCatalog sets the location output to the result of calling the lookup function. (In this simple example, there is no real device; all of the state is stored in the JavaScript environment itself.)

Task decomposition

Task models are hierarchical. Accomplishing high-level tasks usually requires repeatedly decomposing them into increasingly lower-level tasks, or subtasks. This decomposition can sometimes be achieved entirely automatically, while at other times a collaboration between the system and user is required.

For example, the high-level Borrow task is decomposed into three subtasks: GoToLibrary, ChooseBook, and Checkout. By default, the *temporal order* between these steps is linear (totally ordered), but ANSI/CEA-2018 also supports the specification of partial orders. The binding elements in the subtasks definition specify the *data flow* between these steps, which is shown graphically in Figure 4b.

Finally, every decomposition may optionally include an *applicability condition*, which can help the system choose the appropriate decomposition when there is more than one. For example, the applicability condition for the second decomposition of ChooseBook guarantees it will only be chosen when the first decomposition fails. Like pre- and postconditions, applicability conditions are defined using Boolean-valued JavaScript expressions.

REFERENCE IMPLEMENTATION

A reference implementation of ANSI/CEA-2018 under a Berkeley Software Distribution (BSD) open source license is available by contacting the author. It's written in Java and includes both a task engine and a generic UI—a simple command shell that is useful for exploring and debugging task models. In a realistic application, an appropriate graphical, speech, gesture, or other interface would replace this command shell.

Students at Worcester Polytechnic Institute used this reference implementation to build 10 task-based UIs as graduate projects in spring 2008 (www.cs.wpi.edu/~rich/courses/cs525u-s08/projects) and will use it again in 2009. These projects, such as the example in Figure 6, demonstrate that the application of task-based UIs isn't limited to CE.
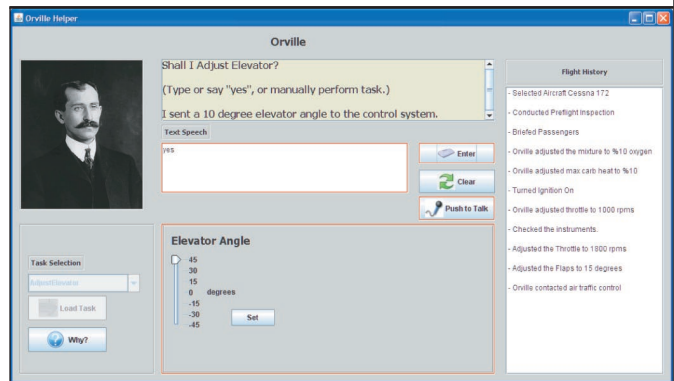

Figure 6. Screenshot of example student-built task-based UI using the ANSI/CEA-2018 reference implementation. The application is an aircraft flight plan assistant.

```
Borrow@format = borrow %s
GoToLibrary@format = go to the library
ChooseBook@format = choose %s
ChooseBook.input@definite = the book you want
LookupInCatalog@format = look %s up in the catalog
TakeFromShelf@format = take %s from %s
UseSearchEngine@format = use the search engine
CheckOut@format = check out %s
```

Figure 7. Optional property file used by reference implementation for formatted printing, with inputs and outputs substituted for each %s by the Java String.format() method.

Northeastern University's Relational Agents Group is also using the reference implementation to guide task-based dialogs (www.ccs.neu.edu/research/rag/research/r_ontology.html), and the Intuitive Interaction for Everyone with Home Appliances Based on Industry Standards project (www.i2home.org) is using it to investigate how to build task-based UIs for UPnP devices.[8]

To demonstrate the reference implementation, a complete annotated transcript of a session using the command shell interface follows. User input appears after the CE prompt.

```
    Welcome to CE Task Engine!
CE> load models/Library.xml
```

First, the user loads the XML document shown in Figure 3.

```
CE> task Borrow
    Let's start to borrow a book.
```

Next, the user creates a new top-level instance of the Borrow task. The system responds by saying "Let's start to" followed by the formatted printing of the current top-level instance (see Figure 7). All of the following system responses similarly use formatted printing.

```
CE> next
    Please go to the library.
```

The user asks the system what to do next, and the system answers based on the task model and current execution state.

```
CE> done
    Ok.
```

Going to the library is a good example of a task that only the user can perform. The user informs the system that this task has been done.

```
CE> status

    Borrow by borrowing [live] [<-focus]
        GoToLibrary [done]
        ChooseBook [live]
        CheckOut
```

Here we see a printout of the current task tree in human-readable form. Note that the task engine keeps track of which tasks have been done, which are live (ready to be done), and which, like CheckOut, haven't been done but aren't yet live due to temporal or other constraints. The reference implementation also includes a rudimentary dialog focus mechanism.

```
CE> next
    What is the book you want?
```

The next step, ChooseBook, can't be performed because it has an unbound input parameter (the book). The system therefore queries the user for a value. Note that the system doesn't ask for a value until it's actually needed.

```
CE> task / new Book("Sawyer","Mindscan")
    Ok.
```

In this generic development/debugging interface, the user provides the input value by evaluating a JavaScript expression.

```
CE> next
    Shall I look Mindscan up in the
    catalog?
```

Looking up a book in the card catalog is a task that either the user or the system could perform, so the system asks for permission.

```
CE> yes
```

```
    Ok.
    Done look Mindscan up in the catalog.
CE> next
    Please take Mindscan from Shelf 2.
```

The system found in the catalog that the desired book is on shelf 2 and asks the user to take it from the shelf.

```
CE> done
    Ok.
    Shall I check out Mindscan?
```

Once informed that the user has taken the book from the shelf, the system immediately offers to perform the electronic checkout, since this is the only subtask remaining and only the system can perform it.

```
CE> yes
    Ok.
    Done check out Mindscan.
    Completed borrow Mindscan.
```

> **Multidevice networked configurations require standard libraries that define high-level tasks and alternative decompositions depending on the types of devices available.**

This completes the top-level task. The current reference implementation doesn't include commands to ask when, how, or why questions, but the information to answer them does exist in its data structures (and several of the student project UIs provide this information to the user).

FUTURE WORK
Much work remains to be done before ANSI/CEA-2018 and task-based UIs are likely to have a noticeable impact on the usability crisis.

In the standards arena, the next hurdle is to develop standard libraries of task models for a variety of domains—what are sometimes called *profiles*. ANSI/CEA-2018 is only a language for writing task models in general. To fully support multimanufacturer, multidevice networked configurations, we need standard libraries that define high-level tasks and alternative decompositions depending on the types of devices available. Some standard profiles already exist for low-level tasks in the CE domain in the form of the UPnP device control protocols defined by the Digital Living Network Alliance (www.dlna.org). However, UPnP isn't an

adequate general formalism for task-based UIs because it has no task decomposition hierarchy—it can only define one level of task.

As researchers begin to develop libraries for numerous devices and manufacturers, many scaling challenges will undoubtedly emerge and must be addressed, such as how to index and retrieve appropriate models, how to factor models to best capture similarities, and so on.

On the tools side, the reference implementation is only a start. Hopefully, through the open source process, it will become both more efficient and more powerful. For example, the task engine would benefit greatly from the addition of a truth-maintenance system[9] such as the one Collagen used. Collagen also included a plan recognition component[10]—given an observed sequence of primitive actions and a task model, it could infer which high-level task (goal) was being performed, including which decomposition choices, if any, had been made. The ANSI/CEA-2018 formalism would support a similar algorithm.

As anyone who has tried it can tell you, developing task models is at least as hard as writing a well-structured object-oriented program. Visualization, debugging, and other tools specifically designed or adapted for ANSI/CEA-2018 task models would therefore be a great help.

Finally, there is a lot of room for creativity and experimentation in designing the interface part of task-based UIs—that is, what the user actually sees and hears. In particular, the availability of the task model at runtime provides a good semantic underpinning for developing natural-language and speech interfaces.

I encourage readers to experiment with the ANSI/CEA-2018 standard and the task-based UI methodology to develop more effective CE applications in particular and software interfaces generally. Meanwhile, the technical and market forces driving this work continue to intensify. Home networking, though still a niche market, will inevitably become commonplace. This means that manufacturers will eventually be forced to make all their products remotely operable, which in turn will make it possible for third parties to use the new standard to develop more usable high-level interfaces. Further, with Moore's law showing no signs of slowing down, future products will be able to support ever-more features. **C**

Acknowledgments

References

1. E. Den Ouden, "Development of a Design Analysis Model for Consumer Complaints: Revealing a New Class of Quality Failures," PhD dissertation, Eindhoven Univ. of Technology, The Netherlands, 2006.

2. Consumer Electronics Assoc., *Task Model Description* (CE Task 1.0), ANSI/CEA-2018, Mar. 2008; http://ce.org/cea-2018.

3. J. Eisenstein and C. Rich, "Agents and GUIs from Task Models," *Proc. 7th Int'l Conf. Intelligent User Interfaces* (IUI 02), ACM Press, 2002, pp. 47-54.

4. C. Rich, C.L. Sidner, and N. Lesh, "Collagen: Applying Collaborative Discourse Theory to Human-Computer Interaction," special issue on intelligent user interfaces, *AI Magazine*, vol. 22, no. 4, 2001, pp. 15-25.

5. C. Rich and C.L. Sidner, "DiamondHelp: A Generic Collaborative Task Guidance System," special issue on mixed-initiative assistants, *AI Magazine*, vol. 28, no. 2, 2007, pp. 33-46.

6. F. Paterno, "ConcurTaskTrees: An Engineering Notation for Task Models," *The Handbook of Task Analysis for Human-Computer Interaction*, D. Diaper and N. Stanton, eds., Lawrence Erlbaum Assoc., 2004, pp. 483-500.

7. D.E. Wilkins, "Using the SIPE-2 Planning System: A Manual for Version 6.0," tech. report, SRI Int'l Artificial Intelligence Center, Menlo Park, Calif., 1999.

8. G. Zimmermann, "Open User Interface Standards—Towards Coherent, Task-Oriented and Scalable User Interfaces in Home Environments," *Proc. 3rd IET Int'l Conf. Intelligent Environments* (IE 07), IEEE Press, 2007, pp. 36-38.

9. D. McAllester, "Truth Maintenance," *Proc. 8th Nat'l Conf. Artificial Intelligence* (AAAI 90), AAAI/MIT Press, 1990, pp. 1109-1116.

10. N. Lesh, C. Rich, and C.L. Sidner, "Using Plan Recognition in Human-Computer Collaboration," *Proc. 7th Int'l Conf. User Modeling* (UM 99), Springer-Verlag, 1999, pp. 23-32.

Charles Rich *is a professor of computer science at Worcester Polytechnic Institute. While at Mitsubishi Electric Research Laboratories, he cochaired the CEA working group that developed ANSI/CEA-2018. His research interests include artificial intelligence, human-robot interaction, intelligent user interfaces, and interactive media and game development. Rich received a PhD in artificial intelligence from the Massachusetts Institute of Technology. He is a senior member of the IEEE, a member of the ACM, and a fellow of the Association for the Advancement of Artificial Intelligence. Contact him at rich@wpi.edu.*

For more information on this and other computing topics, please visit our Digital Library at http://computer.org/csdl.