Available online at www.sciencedirect.com

SCIENCE DIRECT®

Cognitive Systems Research 6 (2005) 71–88

Cognitive Systems RESEARCH

www.elsevier.com/locate/cogsys

Specifying ACT-R models of user interaction with a GOMS language

Action editor: Christian Schunn

Robert St. Amant [a,*,1], Andrew R. Freed [2], Frank E. Ritter [b,3]

[a] Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
[b] School of Information Sciences and Technology, The Pennsylvania State University, University Park, PA 16802, USA

Received 16 September 2004; accepted 16 September 2004

Abstract

We describe a system, G2A, that produces ACT-R models from GOMS models. The GOMS models can contain hierarchical methods, visual and memory stores, and control constructs. G2A allows ACT-R models to be built much more quickly, in hours rather than weeks. Because GOMS is a more abstract formalism than ACT-R, most GOMS operators can be plausibly translated in different ways into ACT-R productions (e.g., a GOMS Look-for operator can be carried out by different visual search strategies in ACT-R). Given a GOMS model, G2A generates and evaluates alternative ACT-R models by systematically varying the mapping of GOMS operators to ACT-R productions. In experiments with a text editing task, G2A produces ACT-R models whose predictions are within 5% of GOMS model predictions. In the same domain, G2A also generates ACT-R models that give better predictions than GOMS, providing good predictions of overall task duration for actual users (within 2%), though the models are less accurate at a detailed level. In a separate experiment with a mouse-driven telephone dialing task, G2A produces models that do a better job of distinguishing between competing interfaces than a Fitts' law model or an ACT-R model built by hand. G2A starts to describe the relationship between two major theories of cognition. This may have appeared a simple relationship, but the complexity of the translation illustrates why this was not done before. G2A shows a way forward for cognitive models, that of higher level languages that compile into more detailed specifications.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Cognitive models; GOMS; ACT-R; Search

* Corresponding author.
E-mail addresses: stamant@csc.ncsu.edu (R.St. Amant), arfreed@nc.rr.com (A.R. Freed), frank.ritter@psu.edu (F.E. Ritter).
[1] Supported by the National Science Foundation (IIS-0083281).
[2] Present address: IBM, 4401 Silicon Drive, Building 675, Office D210 Durham, NC 27703, USA.
[3] Supported by the Office of Naval Research (N00014-02-1-0021).

1389-0417/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.cogsys.2004.09.008

1. Introduction

The automatic generation of cognitive models from high-level specifications has been of continuing interest in cognitive modeling for human-computer interaction. Cognitive models such as ACT-R (Anderson & Lebiere, 1998) can give important insights into user behavior, but building detailed models is usually demanding of time and modeling expertise. If cognitive modeling in support of user interface development or analysis could be made a routine, low-cost activity, this would provide a powerful tool to improve user interface designs.

Unfortunately, many of the cognitive architectures in use today are hard to learn, difficult to program, and a challenge to debug – even as models increase in complexity (Pew & Mavor, 1998). Ritter et al. (2003) note that usability is one of the difficulties that preclude wider use of cognitive models. In order for cognitive architectures to be truly useful, they must become easy to use by a population with a range of backgrounds, including user interface developers.

Recently, significant progress has been made toward making models easier to apply to design problems. Our work is inspired by research on high-level cognitive modeling languages, such as ACT-Simple and its conceptual relatives (e.g., Freed, Matessa, Remington, & Vera, 2003; Lewis, Vera, & Howes, 2004; John, Vera, Matessa, Freed, & Remington, 2002; Jones & Wray, 2004; Yost, 1993).

Salvucci and Lee (2003) developed the ACT-Simple system, which automatically generates ACT-R models from a language similar to KLM-GOMS (Card, Moran, & Newell, 1983). John, Prevas, Salvucci, and Koedinger (2004) described techniques for automatically building ACT-R models (via KLM and ACT-Simple) from actions demonstrated by interface designers. In the domain of interaction modeling for cellular telephones, St. Amant, Horton, & Ritter (2004) developed automated techniques for generating partial ACT-R models based on the specification of a keypad and menu hierarchy.

ACT-Simple demonstrates the feasibility of automatically translating a high-level specification language into detailed ACT-R models – an exciting and significant achievement. ACT-Simple can be viewed as a high-level programming language for cognitive modeling. It is deliberately simple in order to encourage wide adoption by interface developers, but at the cost of producing a very restricted set of ACT-R models.

The ACT-Simple translation process and its results have strong limitations. Each operator in the source model is transformed into one or two ACT-R productions in a static translation process. These productions are chained together in a mostly linear fashion. The productions make almost no use of the environment (e.g., all visual processing is represented as shifts in attention between two fixed locations). Essentially, the generated ACT-R models have relatively little that is "cognitive" about them: there is no input of information from the environment; there are no memory retrievals for information processing; there little or no decision-making. This is not to slight ACT-Simple – these are mainly constraints dictated by KLM-GOMS, which is intended to provide a simple, largely external description of expert performance.

We have built a system, G2A, [4] that translates models from GOMSL (Kieras, 1999), an abstract but rich modeling language within the GOMS family, into models in ACT-R. For a brief contrast with KLM-GOMS, GOMSL allows representation of mental objects, working memory storage, primitive internal and external operators, composite methods, and various flow-of-control constructs. G2A supports all of these capabilities in its translation process. Our goal is to allow the expression of models as simple as KLM-GOMS models, but to let modelers add more complexity as needed.

This article has four main parts. In Section 2, we discuss related literature in GOMS and cognitive modeling in HCI, focusing on the potential benefits that improved modeling techniques can bring to user interface development. In Section 3 we describe the G2A system and how it converts GOMSL models into ACT-R models. Our evaluation of G2A is in two parts. The first part, in Section 4, is a proof of

[4] The system is available at www4.ncsu.edu/~stamant/G2A.

concept that compares model performance between GOMSL and G2A models. The most complex model given in Kieras' (1999) GOMSL manual represents a text-editing task. It takes up about four pages – 190 lines of code containing 15 mental object definitions, 11 methods, and two sets of selection rules. We compare the predictions of this GOMSL model and its G2A translation against each other and against user performance from a small empirical study. The second part of the evaluation, in Section 5, applies G2A to a different task domain, telephone dialing. We compare G2A model predictions with those of a Fitts' law model and an ACT-R model generated by hand. In both parts of our evaluation, G2A performs well, supporting our claims for the value of automated or partially automated modeling techniques.

2. Motivation

It has been argued that detailed task analysis and cognitive modeling techniques require too much effort to be practical in developing user interfaces. The problem has been the difficulty in creating models, not in their use or applicability. The techniques have made clear contributions to HCI, producing both theories of how interaction can be improved (Byrne & Gray, 2003) and practical suggestions (Gray, John, & Atwood, 1993). Further, the work cited in the introduction, especially the efforts of John et al. (2004) to allow designers to build models without modeling knowledge, points toward solutions that make model construction a natural, unobtrusive part of the design process.

Scientific and engineering reasons both encourage this work. The engineering reasons suggest that the use of a higher-level language would be helpful when creating cognitive models of users. The scientific reasons suggest that higher-level languages can help explore the relationships between theories. We take these two types up in turn.

The engineering reasons to use a higher-level language are based on making model building easier in several ways. Large models will use the same constructs in many places (e.g., objects and their relationships). A compiler can create these constructs in a uniform way, and when changes to the model occur, the compiler can ensure that the code remains consistent. Model compilers can help produce models that do not violate expectations about how the architecture will be programmed, both explicitly by checking for violations as well as implicitly by only providing constructs that are appropriate.

Most modeling communities have not seen reuse as it was originally envisioned by Newell (1990), but most modeling communities have used what can now be seen as relatively low-level components. A compiler should also support reusable components because the components are defined more regularly, and at a higher level.

Finally, higher-level languages offer the ability to create models more quickly, which has been noted as a problem by several authors (e.g., Pew & Mavor, 1998). GOMS, for example, can be perhaps taught in one to ten hours, and has a 100 page manual. ACT-R, on the other hand, has an introductory course for graduate students (the ACT-R Summer School) that relies on a tutorial manual half again as large and forty hours of instruction. ACT-R would be useful if more models could be created using GOMS or a GOMS-like language because models could be created more easily, at least initial versions. Indeed, Brooks (1975) notes that the number of lines of code per programmer per day does not seem to vary across languages, so why not use a high level cognitive modeling language? [5]

The scientific reasons for a higher level language include making the relationships between major theories more explicit. Previously, ACT-R and GOMS were seen as related in a straightforward way, with GOMS being a higher level of representation that could be related directly and simply to ACT-R. The translation presented here suggests that this view is true but too simplistic.

The use of a GOMS-to-ACT-R compiler will allow concepts from ACT-R to be included in GOMS models. ACT-R models incorporate learning; they include more accurate perceptual and

[5] We thanks Randy Jones, Bob Wray, and Jacob Crossman for useful discussions in this area.

motor components; they produce response times with variance; they incorporate memory effects, including priming and errors; they support a wider range of applications such as agents in simulations. With expanded use and a way to automatically translate GOMS concepts into ACT-R concepts, it may be possible to unify these two theories as different levels of human behavior.

3. GOMSL translation

GOMSL syntax is comparable to that of a procedural programming language Fig. 1 provides an example. G2A begins by parsing a GOMSL model into an intermediate representation [6] using a simple tokenizer and a grammar that we developed, shown in Fig. 2. In this grammar, symbols in SMALL CAPITAL letters represent literals, and the remainder represent variables.

The result of the parsing process is a hierarchical representation of a GOMSL model. The top-level object in this hierarchy is a model that contains a starting goal, a set of methods, a set of selection rules, and a set of data objects. Methods and selection rules break down into steps, each of which breaks down further into operators. We discuss each of these types of objects in the subsections below.

3.1. Methods and flow of control

The most convenient place to start in describing the translation between GOMSL and ACT-R is with top-level flow of control. [7] A GOMSL method is a sequence of steps, where each step can contain one or more operators. For example, in Fig. 1 the first step in the method `Edit Document` is the `Store` operator. The translation process follows

```
LTM item: Cut Command
  Name is Cut.
  Containing Menu is Edit.
  Menu Item Label is Cut.
  Accelerator Key is COMMAND-X.
Method for goal: Edit Document
  Step. Store First under <current task name>.
  Step Check for done.
  Decide: If <current task name> is None, Then
    Delete <current task>;
    Delete <current task name>;
    Return with goal accomplished.
  Step. Get task item whose
    Name is <current task name>
    and store under <current task>.
  Step. Accomplish goal: Perform Unit task.
  Step. Store Next of <current task> under <current task name>;
    Goto Check for done.
```

Fig. 1. GOMSL object and top-level method as examples.

the hierarchical model representation: GOMSL methods expand to steps that expand to operators that are translated into ACT-R productions. A method thus translates into a sequence of productions (with a few complications, as described below).

Every production generated by the translation process begins, like all ACT-R productions, by retrieving a `Goal` chunk. The most important pattern associated with this retrieval is the `%state` slot. (The names of slots that are manipulated by the translation process are prefixed by "%" to avoid possible conflicts with names that a modeler might use.) This is the same scheme used in ACT-Simple in its representation of control: every production contains a condition test of the `%state` slot of a `Goal` chunk and an action to update that slot. By manipulating the pattern associated with the `%state` slots of a set of productions, a sequential ordering can easily be imposed on the set. The productions in Fig. 3 show the basic mechanism.

Step execution, however, is not always sequential within a method. `Decide` statements support branching within a method based on predicate tests, comparable to `if` statements in a standard programming language. If the predicate in a `Decide` statement succeeds, its sequence of operators is executed in normal fashion. If the predicate fails, then execution falls through to the next step in the method. For a `Decide` state-

[6] For this we adapted an off-the-shelf LALR parser, written by Mark Johnson, from the online AI Repository at CMU: http://www-cgi.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/code/parsing/lalr/lalr.cl.

[7] The models produced by G2A include a small number of ACT-R 4 constructs (Anderson & Lebiere, 1998) that are deprecated in ACT-R 5; we are working to make the system ACT-R 5/6 compliant (Anderson et al., 2004).

```
    —– Top level —–
S → form | form rest-forms
form → DEFINE_MODEL ':' id GOAL IS goal '.' | object-type ':' id properties |
    METHOD_FOR_GOAL ':' goal-spec steps |
    SELECTION_RULES ':' goal branches RGA '.'
properties → id IS id '.' | id IS id '.' properties
    —– Selection rules —–
branches → branch | branch branches
branch → IF access comp access ',' THEN AG-form '.'
AG-form → AG ':' goal-spec
    —– Method header —–
goal-spec → goal | goal USING using-list
goal → id id
using-list → access | access ',' using-list | access ',' AND using-list
    —– Method steps —–
steps → step-form | step-form steps
step-form → step-header '.' step-stmts '.' | step-header '.' DECIDE : if-then-form '.'
step-header → STEP id | STEP
step-stmts → step-stmt | step-stmt ';' step-stmts
step-stmt → AG ':' goal-spec | id access | STORE access UNDER TAG |
    DELETE access | test/store predicates AND_STORE_UNDER id | action
    —– Method decide forms —–
if-then-form → if then | if then else
if → IF predicates
then → THEN actions
else → ELSE actions
predicates → predicate | predicate ',' | predicate ',' predicates |
    predicate ',' AND predicates | predicate AND predicates
predicate → access comp access
access → id | id OF id
    —– Method actions —–
actions → action | action ';' | action ';' actions
action → id access | RGA | HOME_TO | KEYSTROKE | TYPE_IN | CLICK |
    DOUBLE_CLICK | POINT_TO | VERIFY | SPEAK
    —– Lexicon —–
object-type → TASK_ITEM | VISUAL_ITEM | LTM_ITEM | AUDITORY_ENCODING
comp → IS | IS_NOT | '>' | '<' | ...
test/store → GET_TASK_ITEM_WHOSE | RECALL_LTM_ITEM_WHOSE |
    LOOK_FOR_OBJECT_WHOSE | WAIT_FOR_VISUAL_OBJECT_WHOSE |
    WAIT_FOR_AUDITORY_OBJECT_WHOSE
```

Fig. 2. Abbreviated GOMSL grammar, with RGA: return goal accomplished, AG: accomplish goal.

ment the translation process generates auxiliary productions that test for predicate failure. The `%state` values in the conditions of these productions are set, in a postprocessing phase, to fall through to the `%state` value associated with the first production for the next step.

```
;;; Get-task-item-whose ((is name [current-task-name])) [current-task]
(P production86
   =goal>
      isa                     goal
      [current-task-name]     =[current-task-name]
      %state                  edit-document-3
   =match88>
      isa                     task-item
      %id                     =temp87
      name                    =[current-task-name]
==>
   =goal>
      [current-task]          =temp87
      %state                  edit-document-4)

(P ag89
   =goal>
      isa                     goal
      %state                  edit-document-4
      . . .
==>
   =goal>
      %state                  edit-document-5
      . . . )
```

Fig. 3. Sample generated productions.

Goto statements are another type of control structure in GOMSL, allowing arbitrary transfer of control to labeled steps. Their translation is a variation on the above process. When a Goto statement is encountered, the translation process searches for a step whose label matches the target of the Goto statement, and the first production for this step is used to set the %state slot in the conditions of the Goto production.

Each GOMSL method satisfies a goal; instead of calling operators, a method can "invoke" another method by an Accomplish-goal (AG) statement. This flow of control translates to an ACT-R production that creates a subgoal to be pushed on the goal stack. When a method completes, it ends with a Return-with-goal-accomplished (RGA) statement; this becomes a pop action in ACT-R. Selection rules, which govern the choice of methods when more than one can accomplish a given goal, are translated similarly to Decide statements.

3.2. Objects and working memory

GOMSL supports declarative object representations, including objects in long term memory (LTM-items), objects available through visual processing (Visual-items), and task descriptions (Task-items). Each object is a named collection of property-value pairs. GOMSL objects are translated directly into ACT-R chunks. Each generated chunk is given a unique identifier for reference.

Objects in GOMSL can be brought into working memory in the form of named tags, such as <current_task_name> in Fig. 1. Properties of an object stored in a tag are also immediately available for processing. G2A captures this functionality via a slot in the Goal chunk for each tag in a GOMSL program. Changes to the contents of a tag by a GOMSL operator translate to updates of the corresponding slot in a Goal chunk.

For convenience, GOMSL allows method arguments to be defined. For example, a method might be defined as follows:

Method_for_goal: Issue Command using <command_name>

with the method being activated by a statement in another method:

Step 2. Accomplish_goal: Issue Command using Cut.

In other words, when the method for issuing a command is executed, it should use "Cut" as the name of the command. The argument <command_name> here is a pseudo-parameter, rather than a true parameter in a programming language sense, because the GOMSL architecture supports only a single working memory system (as might be expected in a system that strives for cognitive plausibility). Instead, named tags are global in scope. Method calls that involve arguments must first place all the relevant values into slots associated with the current Goal object, for retrieval by the productions of the method. This means that when a production creates a subgoal, it copies tag values from the current goal to the subgoal before pushing it on the stack.

3.3. Operators

GOMSL defines a number of primitive operators that carry the basic load of modeling performance. In addition to the control-flow forms shown in Fig. 2, G2A includes the following actions Keystroke, Type-in, Click, Double-click, Hold-down, Release, Point-to, Home-to, Speak, Look-for-object-whose, Get-task-item-whose, Store, Delete, Recall-LTM-item-whose, Verify, and Think-of, encompassing all GOMSL operators except for Wait-for-object-whose and Wait-for-auditory-object-whose, which we consider minor gaps in coverage from an HCI perspective. (Work addressing these gaps is in progress.)

G2A contains two classes of translations to generate ACT-R productions. A *G*-type translation for a given GOMSL operator creates an equivalent ACT-R production whose duration is fixed to a value specified by GOMSL, as shown in Table 1. There will usually be only one *G* translation for any GOMSL operator, and it will result in a rule that has its performance set explicitly.

A-type translations generate productions that follow conventional ACT-R idioms, with durations determined by the ACT-R architecture. There may be several different *A* translations for a single GOMSL operator if different idioms are appropriate.

The *A*-type translations in G2A are defined as follows.

- Home-to takes an argument (mouse or keyboard). Both actions have a direct translation to individual ACT-R productions.

Table 1
Translated GOMSL operator durations, in seconds

GOMSL operator	Duration
Keystroke	0.280
Type-in	*N keys* ∗ 0.050
Click	0.200
Double-click	0.400
Hold-down	0.100
Release	0.100
Point-to	1.100
Home-to	0.400
Look-for-object-whose	1.200
Get-task-item-whose	1.200
Recall-LTM-item-whose	1.200
Store	0.0 or 0.050
Delete	0.0 or 0.050
Verify	1.200
Speak	*N syllables* ∗ 0.150
AG	0.050
RGA	0.050

Times are taken from (Kieras, 1999).

- Keystroke takes a key as input and translates to two productions. The first moves the right hand to the keyboard and the second causes the key to be pressed.
- Type-in takes character string as input. It translates to a sequence of Keystroke operators, one for each character.
- Click translates to two productions, the first for hand movement to the mouse, the second for clicking the mouse.
- Double-click is like click, except it has two click productions.
- Double-click/A, Hold-down/A, and Release/A are translated into ACT-R Mouse-click actions. ACT-R lacks a direct implementation of these as single actions, which has some implications for low-level user modeling, as described in Section 4.3.
- Point-to takes a target of either a literal symbol, an object stored in a tag, or a literal or object stored in the property of a tag. It expands to a Hand-to-mouse production followed by a Move-cursor production, imposing appropriate conditions on the manual state. The G2A translation process generates a location for the object or literal (if one was not provided), which is stored for later pointing actions to the same target. This location is used in the Move-cursor production. (Numerical screen coordinates can be used if provided in the GOMSL model, but this requires specialized task-specific code.)
- Recall-LTM-item-whose and Get-task-item-whose both take a list of predicates and a store tag as arguments. They translate to a production in which the identifier of a retrieved object that meets the predicate tests is stored in the given tag. Fig. 3 provides an example.
  A predicate is a comparison between operands. Comparisons in G2A are limited to tests of equality or non-equality of symbols. The first operand to the comparison is a property of the object to be retrieved. The second is either a literal, a literal stored in a tag, or a property associated with an object stored in a tag. In the first two cases, the production that is generated includes a pattern that makes a direct comparison between the object

property and the literal or tag. In the third case, an additional buffer is added to the production that allows the retrieval of the chunk corresponding to the identifier stored in the tag, and access to its properties. Predicates are processed one at a time until the list of predicates is exhausted. The last action carried out by the production is to store the identifier of the retrieved object in the tag slot (labeled `[current-task]` in Fig. 3) of the current `Goal` chunk.

- `Look-for-object-whose/search`. There are three translations available for this operator. The modeler chooses depending on the subject's expertise and the object location. The first *A*-type translation for this operator, takes a list of predicates and a store tag as arguments. The translation generates productions that search through the set of visual objects present until the desired object is found. These productions use the standard ACT-R idiom of `find/attend/harvest` for visual acquisition. The `harvest` production is augmented via the same predicate processing as with `Recall-LTM-item-whose`.
  If the visual module could be guaranteed to find an object that meets all of the predicate tests on its first try, then the translation would be similar to `Recall-LTM-item-whose`. Lacking this guarantee, however, productions must be generated for each of the predicates that may fail. These secondary productions cause the `find` production to be fired again, to visit another not-yet-attended visual location. When a visual object is found that passes all the predicate tests, its identifier is recorded in the tag slot of the current `Goal` chunk.
- `Look-for-object-whose/direct` is an alternative *A*-type translation. In many situations, exhaustive visual search is not carried out because the user either knows the location of a visual object already or because the predicates rely on pop-out properties of the object. For such situations the translation generates a production that simply moves attention to a specific visual location (randomly generated, but recorded for later use, as with `Point-to`).
- `Look-for-object-whose/none` is another *A*-type translation, intended to handle situations where performance is so practiced that shifts of visual attention are unnecessary. This translation generates no production and takes no time.
- `Store` generates a production that records a given value (either a literal value, the contents of a tag, or the property of the contents of a tag) in a target tag, that is, in the appropriate slot of a `Goal` chunk. There is one special case for translation of `Store` (and `Delete`, below), as dictated by GOMSL. When these operators occur in the same step as other operators, their action is merged with the other operators, which means that they have no independent duration.
- `Delete` generates a production that sets the contents of a given tag to a null (`Empty`) value.
- `Verify/last` generates a production that moves attention to the visual location that was most recently visited by a `Look-for-object-whose` or a `Point-to` operator.
  `Verify/none` generates no production, taking no time.
- `Speak` generates a production to utter the argument.

The translations given above have been tested using 34 examples taken from the GOMSL manual (mostly verbatim, though some correction of minor syntax errors was necessary.) When the examples were sufficiently detailed to be executed, G2A generated executable ACT-R models using either G- or A-translations.

4. Evaluation: G2A for user modeling

This first part of our evaluation constitutes a proof of concept of G2A's functionality. We show that a model automatically generated by G2A corresponds well to an extended GOMSL model describing a text-editing task, taken from the GOMSL manual. This is not enough, however; even if for a given task G2A can produce a model at a more detailed level than an existing GOMSL model, we still want to establish some resulting benefit of the additional effort. To demonstrate this, we describe a search process that chooses between different translations for the same operators, producing models with different characteristics. In a small user study, G2A produces models that perform considerably better than the GOMSL model in predicting user performance for the text editing task with respect to reaction time.

4.1. Comparing GOMSL and ACT-R models

We can test the accuracy of the GOMSL to ACT-R translation directly by choosing a GOMSL model, translating it using *G* translations for all its operators, and running a timing comparison.

For our comparison we used the `Edit Document` model from the GOMSL manual (Kieras, 1999). It includes four top-level methods describing activities in a mouse-based word processing environment. `Copy word` involves selecting and copying a single word and pasting it elsewhere in the document. `Copy arbitrary` is a comparable task for a sequence of words. `Delete word` involves selecting a word and pressing the delete key. `Move arbitrary` is similar to `Copy arbitrary`. All editing actions are through selections from a pull-down menu. These lower-level activities are accomplished by the methods given in the *Method* column of Table 2.

Table 2
Model predictions for the `Edit Document` task, in seconds

Method	GOMSL (s)	G2A$_G$ (s)	Error (%)
`Select-insertion-point`	3.60	4.14	15
`Select-word`	4.40	4.71	7
`Erase-text`	6.40	7.25	13
`Select-arbitrary-text`	6.70	6.52	−3
`Issue-command`	9.05	8.55	−6
`Paste-selection`	12.80	12.84	0
`Copy-selection`	14.85	14.36	−3
`Cut-selection`	16.20	15.79	−3
`Copy-text`	28.85	28.55	−1
`Move-text`	30.80	29.98	−3
`Edit-document`	101.20	100.63	−1

To match this model, we set G2A to produce an ACT-R model using only *G*-type translations. The generated model, G2A$_G$, contains five chunk definitions, 23 chunks, 79 productions, and various auxiliary constructs, over 1,500 lines of formatted model code in all, an expansion of 750%.

The *GOMSL* column in Table 2 shows the durations per method as given by the GOMSL model (Kieras & Meyer, 1997, p. 58), and the *G2A$_G$* column shows the corresponding values for the generated ACT-R model. The ACT-R numbers are averaged over 20 runs, to account for variation due to random placement of objects in the environment. The fourth column, labelled Error, shows that the GOMSL and ACT-R model predictions are very close, $r = 0.999$, with small discrepancies due to parallel execution of visual and motor actions in ACT-R, plus the automatic addition to the ACT-R model of hand movements between keyboard and mouse that are implicit in the GOMSL model (specifically, in `Select-word` and `Select-insertion-point`). On average, the predictions of *G2A$_G$* are within 6% of the GOMSL method durations, and the overall task duration prediction is within 1%.

The discrepancies between the two models are largest in the `Select-insertion-point` and `Select-word` methods. We attribute this to the greater run-time robustness of GOMSL models. In the GOMSL model, even if movements of the hand between the keyboard and mouse to prepare for key presses or mouse actions are not modeled explicitly, the GOMSL model will still generate appropriate predictions. This is the case in the `Edit Document` task, which contains no `Home-to` operators. In ACT-R, in contrast, if these preparatory actions are not explicit in a model, incorrect execution will result. G2A's limited compile-time processing means that the *G* translations for keyboard and mouse actions may generate superfluous `hand-to-home` and `hand-to-mouse` productions, even if the hand is already correctly placed.

Table 2 thus presents a straightforward but non-trivial result: it shows that the hierarchical relationships between GOMSL methods, flow of control within the methods, and transfer of information between methods are captured in the productions generated from the ACT-R model created by G2A, even if production durations are fixed.

4.2. Predicting user performance

An obvious next step, staying within the same task domain, is to see how well these models predict actual user performance. If we time users carrying out the `Edit Document` task in a standard word processing application, we find that it takes only about 18 or 20 s, far less than the 100 s predicted by the GOMSL and $G2A_G$ models. We can make educated guesses about why these models perform poorly, but if we believe that they nevertheless correctly represent the basic structure of the task, we can develop explanations for observed behavior in a more systematic way with G2A.

The alternative ACT-R (type "A") translations in the previous section can be thought of as ACT-R idioms for representing particular activities, which means that G2A faces the same issues as a human cognitive modeler: how should one design an accurate model? G2A can build models by using the GOMSL model as a stand-in for domain knowledge and user data, if available, to guide its decisions. In other words, G2A can carry out a model-fitting process, automatically evaluating which assumptions must be changed to determine which strategies are represented in a model.

To do this, G2A treats the alternative translations for GOMSL methods as a search space. By varying the translations that are activated in generating ACT-R models, G2A explores this search space, using hill-climbing to identify the best translation. In hill-climbing, an evaluation function *f* is applied to a current state *s*. The function *f* is applied to each of the states neighboring *s*, and if any of these successor states produces a better value, the best of them becomes the new current state. The process repeats until no successor state produces a better value than *f*(*s*).

A state for the G2A search is a set of translations, one for each method: `Look-for-object-whose/direct` + `Verify/last` + `Click/G` + ⋯. Each translation set produces a unique model. Successors to a translation set are those that differ in the translation of one method. For *f*, G2A executes the model corresponding to the current translation set 20 times, collects predictions of the total duration of the `Edit Document` task, and computes the difference from a target duration. (Notice that we cannot directly compare method execution times, because method boundaries are only implicit in user behavior – all we have access to is external events.)

To test this idea in practice, we conducted a small user study. We implemented a simple, instrumented text editing application comparable to Microsoft Notepad, tailored to the `Edit Document` tasks described in the GOMSL manual. A pilot subject ran ten trials, following the same sequence of `Copy word`, `Copy arbitrary`, `Delete word`, and `Move arbitrary` tasks. We then ran six users through the same procedure as the pilot subject.

The mean duration of the pilot user's trials was 16.30 s, which we used as a target for the G2A search evaluation function. For the search process, we used both *A*- and *G*-type translations, to identify possible areas where one type of translation is superior to another (e.g., a given *A*-type translation of a mental operator might neglect cognitive processing that is accounted for by a longer *G*-type translation.) Fig. 4 shows the distribution of execution durations for a sampling of 1,000 different models in the G2A search space (as a lower bound, there are 17 operators with at least two translations, or 131,072 possibilities). The fastest model completes in about 10 s, the slowest in about 105 s. [8]

G2A evaluated only about 50 models in its hill-climbing search. The model with predictions closest to the pilot data, as determined by G2A's evaluation function, shows no surprises: $G2A_1$ relies on *A*-type translations for all mental and visual activities; it perform no visual search, directing attention to known locations; it does no verification; and it uses *A*-type translations for all motor actions except `Hold-down`, for which a *G*-type

[8] The peaks in this distribution contain on the order of 100 models that differ from each other internally and yet produce very similar overall predictions. This indicates the importance of domain knowledge in automatically generated models, an issue not addressed here.

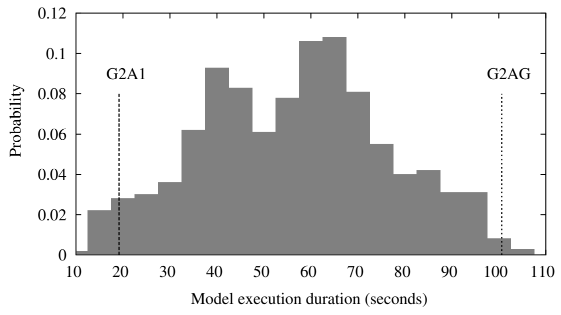


Fig. 4. Distribution of generated model execution duration, with $G2A_1$ and $G2A_G$ times indicated.

translation is used. When we compare $G2A_1$'s predictions of total duration with the performance of the six users in the study, we find that it also gives good results. The grand mean duration over the six users was 18.85 s, with a standard deviation of 1.82 over the six user means. $G2A_1$'s prediction of 19.18 s gives an error of about 2%.

This is also a result worth noting: the ACT-R model automatically produced by G2A's optimization process reflects decisions that a human modeler might make to capture human performance in this domain, and its prediction is much more accurate (and obtained with little additional modeling effort) than that of the original GOMSL model. $G2A_1$ has about the same predictive power as comparable automatically generated ACT-R models for different domains described in the literature (Salvucci & Lee, 2003; John et al., 2004).

4.3. Refining a model for more detail

$G2A_1$ has obvious built-in limitations. In particular, the duration of its mouse movements and visual attention shifts are based on random locations, rather than an actual environment, and thus its detailed predictions of user actions are artificial and unlikely to be veridical. Using the data from the pilot subject, however, it is possible to carry out a more detailed analysis refining $G2A_1$.

We defined a new search evaluation function that measures the intervals between keyboard/mouse events (i.e., stripping out the cumulative duration of each event), and computes the mean squared error with the corresponding interval durations for our pilot user. We also altered the model generation process to use actual locations of objects, as measured in our instrumented application.

The best model produced by the search process for this set of assumptions, $G2A_2$, is almost identical to $G2A_1$: it does no visual search and no verification, and it relies on A translations for all mental operators. It varies only in that `Double-click` is implemented by a single action translation, rather than a sequence of mouse clicks.

$G2A_2$'s prediction of overall task duration is about the same as that of $G2A_1$, 19.03 s. The more detailed predictions of $G2A_2$ are shown in Table 3, along with the mean intervals between mouse click and keystroke events for the six users in the study. Unfortunately these predictions of $G2A_2$ are not as good as we might have hoped for. The correlation between user intervals and $G2A_2$ predictions is 0.754, and the average error in the predictions is 35%. There are two values that are much larger than the others in Table 3, for the first instance of selecting the buffer insertion point and for pressing the Delete key. The `Keypress` translation generates two productions, one that homes the hand to the keyboard, and another that presses Delete, based on the distance between the fifth finger in its home row position and the position of the

Table 3
User performance and model predictions for the `Edit Document` task, separated by visible actions, in seconds

Action	User	$G2A_2$	Error
`Double-click`	0.25	0.20	−20% (0.05 s)
`Select Copy`	1.48	1.40	−5% (0.08 s)
`Set insertion`	0.53	1.01	+91% (0.48 s)
`Select Paste`	1.28	1.18	−8% (0.10 s)
`Select sentence`	2.16	2.02	−6% (0.14 s)
`Select Copy`	1.81	1.16	−36% (0.65 s)
`Set insertion`	0.76	1.01	+33% (0.25 s)
`Select Paste`	1.62	1.18	−27% (0.44 s)
`Double-click`	1.35	1.74	+29% (0.39 s)
`Press Delete`	0.64	1.39	+117% (0.75 s)
`Select sentence`	2.92	2.00	−32% (0.92 s)
`Select Cut`	1.82	1.13	−38% (0.69 s)
`Set insertion`	0.74	0.96	+30% (0.22 s)
`Select Paste`	1.49	1.18	−21% (0.31 s)

target key. This overestimation is not surprising; users only pressed a single key during their trials, and this key is in the top right corner of the keyboard; their movement during this task subsequence was much more streamlined. Some details are obviously not being captured by the model. If a refined model were able to reduce the two largest overestimates, this would bring the model's error down to 24% – still high, but more respectable.

There are two ways we might change $G2A_2$ to produce such an improvement. First, for consistency with the GOMSL model, we specified user tasks in the simulation interface to use the same low-level mouse actions, in particular `Double-click`, `Hold-down`, and `Release`, for which our ACT-R translations are known to be inaccurate (all are implemented as `mouse-click` actions). We expect that once the missing actions are developed, validated, and added to the ACT-R architecture, we will see better results. Second, it is possible that the structure imposed on ACT-R productions by GOMSL forms (e.g., GOMSL methods entail some overhead due to `AG` and `RGA` operators) degrades the accuracy of predictions of a generated model, compared with the predictions of a "native" ACT-R model. Different translations that combine or parallelize GOMSL operators more effectively may help. Both of these efforts remain for future research and development.

Despite the limited predictive power of $G2A_2$ at a detailed level, it constitutes another interesting result. $G2A_2$ was produced as a refinement of $G2A_1$, in a largely automated process that called for only the small effort of supplying detailed environment information. Otherwise the process depended mainly on the GOMSL model for domain knowledge and pilot user data to guide model construction. Existing work on model translation has not tested model predictions at the same level of detail as $G2A_2$ (e.g., Salvucci & Lee, 2003; John et al., 2004), and given this lack of bench marks we find its performance surprisingly good. The next step will be to apply the techniques described in this article to other domains to test the generality of the approach. We discuss just such an exploration in the next section.

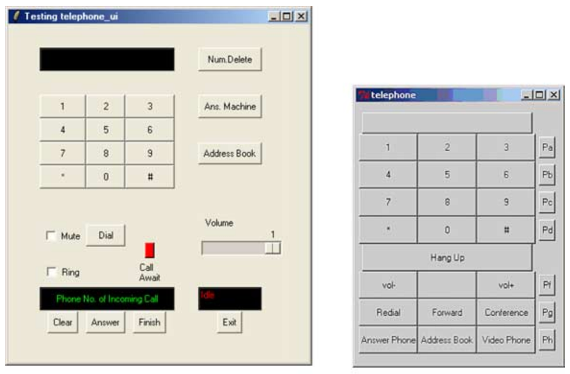
5. Evaluation: Using G2A to compare interfaces

In this section, we describe the application of G2A to a different task, dialing telephones on a desktop computer platform. The goal of applying modeling techniques to the dialing task is to compare the relative efficiency of different telephone layouts. We draw here on results from Freed (2003). We focus on the portion of his work that deals with the duration of the dialing task.

5.1. The experiment environment

The dialing task is simple: pressing a sequence of ten digits with the mouse on an on-screen telephone interface. Ten different interfaces were used during the experiment, drawn from a collection of 100 telephone interfaces developed by students for a user interface course. The ten interfaces were chosen so that they would have a variety of button sizes and shapes, colors, numbers of extra features (in addition to the keypad), and levels of feedback. A sample of the interfaces, on the same 50% scale, is shown in Fig. 5.

Nine participants were recruited for the experiment, between the ages of 21 and 45. The dialing portion of the experiment took about thirty minutes for each subject. The experiment environment consisted of a full-screen application shown on a seventeen-inch monitor with 1024 × 768 pixels. A trial began with a single button centered on the neutral gray screen, labeled "Go". When the user



Telephone interface #3 Telephone interface #5

Fig. 5. Sample telephone interfaces.

pressed the button, a new telephone interface was presented and the user dialed a number. User actions were timestamped and logged. Subjects were asked to dial ten different numbers on each of the ten phone interfaces, for a total of 100 numbers dialed. Trials containing errors were removed from the data during analysis.

The telephone numbers dialed during the experiment are given in Table 4. We performed a statistical analysis of distances between randomly chosen keys and selected ten numbers that provide a reasonable range and distribution of movement distances.

5.2. A Fitts' law model

The simplest theoretical analysis for an interface can be performed using Fitts' law (Fitts, 1954), which relates target size and distance to movement time. As established in extensive empirical research (MacKenzie, Sellen, & Buxton, 1991; MacKenzie & Buxton, 1994; MacKenzie, 1995; MacKenzie, 2003), Fitts' law governs mouse movement time, relating it to the distance to the target and the width of the target along the axis of movement. We use a common formulation of Fitts' law, $\mathrm{MT} = a + b * \log_2(2 * D/W)$, where a is a reaction time constant, b is a movement time multiplier, D is the distance to the target, W is the width of the target along the axis of movement, and MT is the total predicted movement time. In our model, $a = 0.100$ s and $b = 0.150$ s. D and W are determined empirically by measurement.

To apply the Fitts' law model to the dialing task, we need to include the time to press the mouse button, in addition to movement time. This number varies across modeling paradigms, but we adopt the value of 0.200 s, as used in the Keystroke Level Model (Card et al., 1983, p. 264), CMN-GOMS (Card et al., 1983; John, 2003), and GOMSL (Kieras & Meyer, 1997).

5.3. A native ACT-R model

An ACT-R model developed by Freed (2003) predicts not only dialing duration but also eye fixations during dialing. The model incorporates relatively detailed productions that control visual search. For example, the model focuses on a key (a digit in the number to dial), moves the mouse pointer to click on it, and then performs an explicit attention shift left, right, up or down, depending on the digit to be dialed. The model contains seven chunk types, 24 chunks, and 67 productions. For the purposes of comparison, this model took approximately three weeks to write and test.

The model is designed to use its visual and motor subsystems in parallel. While the motor component of the model moves the mouse and clicks on a button, the visual component searches the interface for the next button to dial. Dialing is broken down into three components: the area code (three digits), the exchange (three digits), and the extension (four digits). The model starts by randomly searching for the keypad, and then systematically searching within the keypad for each of the buttons it needs to dial. The model never looks ahead more than one digit at a time. While the model searches for the last digit of the area code on the telephone, it retrieves the exchange from memory, and when it searches for the last component of the exchange, it retrieves the extension.

5.4. A G2A model

To build a G2A model to perform the dialing task, we began by developing a simple GOMSL model, which took less than half an hour to write. The bulk of the model is taken up by the task specification, the order in which the keys in a telephone number are pressed. There are only two methods in the model, as shown in Fig. 6. `Dial Number` retrieves the digits in the number to be dialed,

Table 4
Phone numbers dialed

Numbers dialed				
814-866-5000	215-654-5785	123-654-7890	814-234-9657	814-863-5000
740-611-9273	412-268-3000	101-010-1010	606-193-3012	103-273-1029

```
// Task items are specific to a single telephone number
Task_item: T1
    Name is T1.
    Digit is "8".
    Next is T2.
Visual_object: first_digit
    Content is "8".
...
Method_for_goal: Dial Number
    Step. Store T1 under <current_task_name>.
    Step Check_for_done.
        Decide: If <current_task_name> is T11, Then
        // T11 is a dummy value for the Next property of the last task item.
            Delete <current_task>;
            Delete <current_task_name>;
            Return_with_goal_accomplished.
    Step. Get_task_item_whose Name is <current_task_name>
        and_store_under <current_task>.
    Step. Accomplish_goal: Dial Digit.
    Step. Store Next of <current_task> under <current_task_name>;
    Goto Check_for_done.
Method_for_goal: Dial Digit
    Step. Look_for_object_whose Content is Digit of <current_task>
        and_store_under <target>.
    Step. Point_to <target>; Delete <target>.
    Step. Click mouse_button.
    Step. Verify "Correct digit pressed".
    Step. Return_with_goal_accomplished.
```

Fig. 6. Abbreviated GOMSL specification of the telephone dialing task.

iterating over them in the order the keys are pressed. `Dial Digit` moves the mouse cursor to a specific key and clicks the mouse.

Lacking pilot user results as in Section 4, we treated the user duration for the first telephone interface as the target for our G2A analysis. We also gave G2A information concerning the locations and sizes of the keys in the interface. The best model returned by G2A has a duration well above 9 s, much longer than the observed user value of 6.42 s as well as the duration of the Fitts' law model, and about the same as the median duration of the native model. The G2A model does no visual search, no verification, and in general is as fast as possible using the operator translations described in Section 3.3.

Although G2A in its current form cannot produce models that match user performance for the target telephone interface, we have an interesting recourse – we can extend G2A's set of translations. The first changes we made were straightforward. The only motor operators in the model are `Click` and `Point-to`. By default, mouse operators generated by G2A include a `Hand-to-mouse` production in order to avoid potential inconsistencies during model execution. Because the dialing interfaces in this experiment are completely mouse-driven, however, no movements are needed between the mouse and keyboard. We modified the translations of these operators to remove hand movement to the mouse and thus reduce execution time.

The second set of changes was more involved, drawing on basic techniques in compiler optimization: loop unrolling and function inlining. Consider a simple loop in a procedural programming language, in which a variable j iterates from 1 to n over the body of the loop. The variable j is tested and updated n times, and control is transferred from the end of the loop to its beginning n times. This overhead can be reduced by unrolling the loop. For example, if we include two copies of the body inside the unrolled loop, incrementing j appropriately between the copies, then we can cut the overhead for testing and transferring control in half. If we include n copies of the body of the unrolled loop, we can dispense with all overhead aside from updating j, at the cost of requiring more space for a larger program. We can apply this technique to the translations in G2A. It is not straightforward, because loops are only implicit in GOMSL models, but we have developed a loop-unrolling translation in G2A for specialized iteration patterns in GOMSL models.

The other optimization technique we applied is function inlining. In a procedural programming language, the statements of a function are executed in sequence. When function a calls function b, a new context is created by pushing appropriate information related to b onto the program stack. When b returns, the stack is popped and a resumes execution. Inlining a function removes this overhead by directly including the statements of function b in the body of function a. Inlining a function saves the cost of the function call and return. As with loop unrolling, function inlining trades space for execution time. G2A's translation of `AG` and `RGA` operators follows essentially the same function calling scheme as described above, and inlining GOMSL methods produces comparable savings as a result.

Because our loop unrolling and function inlining translations are not yet completely general, they are under the modeler's control rather than available to the automated search component. In the case of the dialing task, however, there are only two possibilities for their application. The loop in `Dial Number` can be unrolled, and the method `Dial Digit` can be inlined. When G2A performs both optimizations, the resulting model gives a duration of 8.02 s for the "pilot" telephone interface with which we are working – still above user performance of 6.42 s, but as fast as we can expect to make an ACT-R model without tweaking its internal parameters or applying a learning compilation procedure, which are still exploratory.

5.5. Model performance

User performance and model predictions for the experiment are shown in Fig. 7. The lines connecting the model predictions are not meaningful, but are provided so that general patterns can be seen more easily.

None of the models predict user dialing time especially well. The Fitts' law model predictions, however, track changes in dialing time across telephones; if we were to increase the constant duration of a mouse click in the model to 0.365 s, the predictions would overlay the user data almost exactly. This suggests that physical motor actions dominate the telephone dialing task, which is borne out by research that examines dialing tasks on other platforms, such as mobile telephones (Silfverberg, MacKenzie, & Korhonen, 2000). Though it is less obvious by visual analysis, the G2A model also tracks changes in dialing time across telephones well, even if the magnitude of differences in its predictions is much smaller.

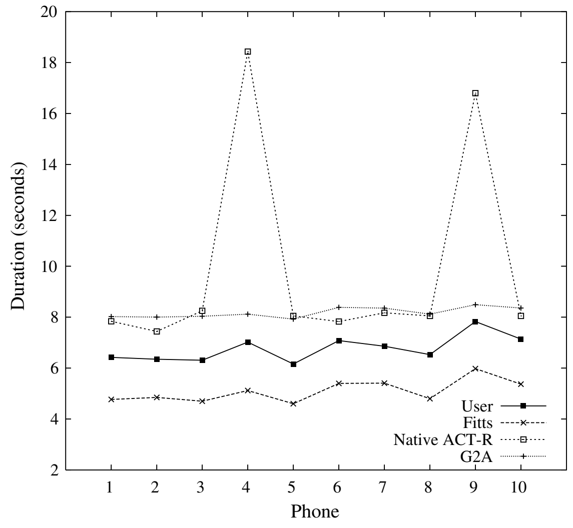


Fig. 7. User performance and model predictions for the dialing task.

Although the native ACT-R model is not as good as the Fitts' law and G2A models, [9] it makes important qualitative predictions that the others do not. The native ACT-R model makes extreme predictions for two telephone interfaces for a specific reason: the model is designed for a button layout in which the top row contains the numbers 1 through 3, the next row 4 through 6, and so forth. Interfaces #4 and #9 reverse this ordering of rows, in the fashion of a calculator rather than a standard telephone. When the native ACT-R model fails to find a specific button in the expected location, its directed visual search degrades to a much less efficient traversal of the display. Neither the Fitts' law model nor the G2A model, both of which ignore patterns in button layout, predicts a higher dialing duration specifically due to the reversed layout, but the increase in dialing duration in the user data for the two interfaces is partially explained by the behavior of the native ACT-R model, and it illustrates the problem solving behavior such models can resort to with difficult interfaces.

For a direct comparison between the models, we have two plausible measures, as shown in Table 5. Of the two, the rank-order correlation is

[9] The poor performance of the native ACT-R model can partly be explained by the difficulties of interacting with real interfaces, rather than simulations or specifications as with the other two models. The native model based its duration computations on the observed size of the numbers to be selected in the display, rather than more appropriately on the size of the buttons that contained the numbers. Its predictions are thus dominated by the distance between targets, neglecting their sizes. As a result, our comparison focuses more closely on differences between the G2A and Fitts' law models.

Table 5
Comparative model performance measures for the dialing task

Performance measure	Fitts' law	Native ACT-R	G2A
Rank-order correlation (Spearman's ρ)	0.879	0.916	0.264
Correlation (Pearson's r)	0.966	0.905	0.638

the most important: it allows us to decide which model gives the most accurate results in predicting the efficiency of one telephone interface to another. That is, the more accurately that a model can predict the relative efficiency of different interfaces, the more effective it will be as an analysis tool for choosing between them. For this measure, the G2A model is superior to either of the others, with a rank order correlation (ρ) of 0.916. The Fitts' law model is close in this regard, with $\rho = 0.879$. The least-squares correlations for both models are also comparable, at $r = 0.966$ for the Fitts' law model and $r = 0.905$ for the G2A model. [10] The native ACT-R model does not perform as well.

A skeptical modeler might argue at this point that what we have done in developing our G2A model is to strip an ACT-R model down until it is as close as possible to a Fitts' law model, but this is not quite right: the G2A model actually produces better rank-order predictions than the Fitts' law model, which we attribute to the greater detail in which interleaved visual and motor operations are represented. The G2A model has further advantages, in that as an ACT-R model it can be extended to explore cognitive issues at lower or higher levels of detail, for example, whether its visual search strategy matches user behavior, or how its processing can be integrated with other tasks or influenced by behavioral moderators (Ritter, Reifers, Klein, Quigley, & Schoelles, 2004). Integrating such factors into the modeling process without the support of a unified cognitive architecture is problematic.

[10] In abstract terms, the duration of the dialing task in a Fitts' law model consists of a variable duration for each movement plus a constant duration for each button press. A GOMSL model breaks down similarly, with identical movement durations but larger constant durations for memory processing in addition to button presses. The GOMSL model will thus perform the same as the Fitt's law model with respect to our two correlation measures.

6. Conclusion

To summarize, G2A takes a GOMSL model as input and generates an ACT-R model that reflects the GOMSL model's flow of control and information processing, generating almost identical predictions. G2A can generate alternative models through an automated search process, which give good predictions of user performance at a high level of abstraction and still reasonable predictions at a more detailed level. We have tested the capabilities of the system in two domains. The resulting comparisons demonstrate the promise of the approach. The size and complexity G2A, while not huge, helps explain why such a system has not been developed before – one might have imagined it as an afternoon of work, which it was not. We expect that with further development and refinement, by ourselves and others, G2A will benefit the community of researchers interested in cognitive modeling for HCI.

One open research area is related to bracketing (Kieras & Meyer, 2000). In bracketing a fastest possible model and a slowest reasonable model (based on optional or inefficiently executed task components as well as different human operator speeds, which we have not explored here) are derived from a base strategy; results can then help a designer decide, for example, whether the performance demands of a system are likely to support the wide range of human capabilities. G2A could contribute to bracketing by generating different models given a base strategy represented as a GOMSL model. If the fastest and slowest models are within the search space of G2A's translations – an important consideration, but a reasonable expectation in some domains – then these might be identified automatically. Work along these lines could lead to useful extensions to G2A, first in the form of translations of higher-level GOMSL methods rather than only primitive operators, and second in the form of modifications to internal ACT-R parameters as well as model structure.

A different benefit relates to model size and reuse. The largest models that G2A generates for the `Edit Document` task contain about 100 productions, and nothing prevents us from generating much larger models for more complex tasks. It is time-consuming and error-prone to generate models of this size by hand even if the modeled behavior is completely understood. Adding methods to a GOMSL model and examining their translation is straightforward in G2A and easier, in our experience, than writing ACT-R productions directly. G2A offers a possible solution for tasks too large to be handled with conventional ACT-R development tools.

Potential benefits of G2A will involve tradeoffs that can only be determined by further research. For example, it may be that increasing model modularity, which makes larger models more feasible, involves a reduction in accuracy. This is suggested by our model refinement efforts, but it is not clear that this is a necessary implication of G2A processing. It may happen that abstract tuning of models via high-level compiler-like directives does not allow sufficient precision, but again this outcome is untested. Still, we believe that G2A represents a promising approach. G2A advances the state of the art in cognitive model generation and points to several important areas for further research in HCI.

Acknowledgements

We thank Mike Byrne, Wayne Gray, Bonnie John, David Kieras, and other audience members at the International Conference for Cognitive Modeling, 2004, for comments that improved this article. The information in this article does not necessarily reflect the position or policies of the US government, and no official endorsement should be inferred.

References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Quin, Y. (2004). An integrated theory of the mind. *Psychological Review, 111*(4), 1036–1060.

Anderson, J., & Lebiere, C. (1998). *The atomic components of thought*. Mahwah, NJ: Lawrence Erlbaum.

Brooks, F. P. (1975). *The mythical man-month: Essays on software engineering*. Reading, MA: Addison-Wesley.

Byrne, M. D., & Gray, W. D. (2003). Returning human factors to an engineering discipline: expanding the science base through a new generation of quantitative methods – Preface to the special section. *Human Factors, 45*, 1–4.

Card, S. K., Moran, T. P., & Newell, A. (1983). *The psychology of human–computer interaction*. Hillsdale, NJ: Lawrence Erlbaum.

Fitts, P. M. (1954). The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology, 47*, 381–391.

Freed, A. R. (2003). The effects of interface design on telephone dialing performance. M.S. dissertation, Department of Computer Science and Engineering, The Pennsylvania State University.

Freed, M., Matessa, M., Remington, R., & Vera, A. (2003). How APEX automates CPM-GOMS. In *Proceedings of the International Conference on Cognitive Modeling*, Bamberg, Germany. Universitätsverlag Bamberg.

Gray, W. D., John, B. E., & Atwood, M. E. (1993). Project Ernestine: validating a GOMS analysis for predicting and explaining real-world task performance. *Human–Computer Interaction, 8*(3), 237–309.

John, B. E. (2003). Information processing and skilled behavior. In J. M. Carroll (Ed.), *HCI models, theories, and frameworks*. San Francisco, CA: Morgan Kaufman.

John, B. E., Prevas, K., Salvucci, D. D., & Koedinger, K. (2004). Predictive human performance modeling made easy. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI'04). Vienna, Austria: ACM.

John, B. E., Vera, A., Matessa, M., Freed, M., & Remington, R. (2002). Automating CPM-GOMS. In *Proceedings of CHI'02: Conference on Human Factors in Computing Systems* (pp. 147–154). New York: ACM.

Jones, R. M., & Wray, R. E. (2004). Toward an abstract machine architecture for intelligence. In R. Jones, R. E. Wray, & M. Scheutz (Eds.), *AAAI Worshop on Intelligent Agent Architectures: Combining the Strengths of Software Engineering and Cognitive Systems* (pp. 46–52). Menlo Park, CA: AAAI Press.

Kieras, D. E. (1999). A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN3. University of Michigan, Ann Arbor, MI. ftp://www.eecs.umich.edu/people/kieras/GOMS/GOMSL_Guide.pdf.

Kieras, D. E., & Meyer, D. E. (1997). An overview of the EPIC architecture for cognition and performance with application to human–computer interaction. *Human–Computer Interaction, 12*(4), 391–438.

Kieras, D. E., & Meyer, D. E. (2000). The role of cognitive task analysis in the application of predictive models of human performance. In J. M. Schraagen, S. F. Chipman, & V. L. Shalin (Eds.), *Cognitive task analysis*. Mahway, NJ: Lawrence Erlbaum.

Lewis, R. L., Vera, A. H., & Howes, A. H. (2004). A constraint-based approach to understanding the composition of skills. In *Proceedings of the Sixth, International Conference on Cognitive Modeling* (pp. 148–153). Mahwah, NJ: Lawrence Erlbaum.

MacKenzie, I. S. (1995). Movement time prediction in human–computer interfaces. In R. M. Baecker, J. Grudin, W. A. S. Buxton, & S. Greenberg (Eds.), *Readings in human–computer interaction: Toward the year 2000* (pp. 483–493). San Francisco, CA: Morgan Kaufmann.

MacKenzie, I. S. (2003). Motor behavior models for human–computer interaction. In J. M. Carroll (Ed.), *HCI models, theories, and frameworks*. San Francisco, CA: Morgan Kaufman.

MacKenzie, I. S., & Buxton, W. (1994). The prediction of pointing and dragging times in graphical user interfaces. *Interacting with Computers, 6*, 213–227.

MacKenzie, I. S., Sellen, A., & Buxton, W. (1991). A comparison of input devices in elemental pointing and dragging tasks. In *Proceedings of the Human Factors Society* (pp. 161–166), Human Factors and Ergonomics Society, Santa Monica, CA.

Newell, A. (1990). *Unified theories of cognition*. Cambridge, MA: Harvard University Press.

Pew, R. W., & Mavor, A. S. (1998). *Modeling human and organizational behavior: Application to military simulations*. Washington, DC: National Academy Press.

Ritter, F. E., Reifers, A., Klein, L. C., Quigley, K., & Schoelles, M. (2004). Using cognitive modeling to study behavior moderators: pre-task appraisal and anxiety. In *Proceedings of the Human Factors and Ergonomics Society* (pp. 2121–2125). Santa Monica, CA: Human Factors and Ergonomics Society.

Ritter, F. E., Shadbolt, N. R., Elliman, D., Young, R. M., Gobet, F., & Baxter, G. D. (2003). Techniques for modeling human performance in synthetic environments: A supplementary review. Wright Patterson Air Force Base, OH: Human Systems Information Analysis Center. Available from iac.dtic.mil/hsiac/S-docs/SOAR-Jun03.pdf.

Salvucci, D. D., & Lee, F. J. (2003). Simple cognitive modeling in a complex cognitive architecture. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI '03) (pp. 265–272). New York, NY: ACM.

Silfverberg, M., MacKenzie, I. S., & Korhonen, P. (2000). Predicting text entry speed on mobile phones. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI '00) (pp. 9–16). New York, NY: ACM.

St. Amant, R., Horton, T., & Ritter, F. E. (2004). Model-based evaluation of cell phone menu interaction. In *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI '04). ACM.

Yost, G. R. (1993). Acquiring knowledge in Soar. *IEEE Expert, 8*(3), 26–34.