CogTool+: Modeling human performance at large scale

HAIYUE YUAN, Centre for Vision, Speech, and Signal Processing (CVSSP), University of Surrey, United Kingdom
SHUJUN LI, Kent Interdisciplinary Research Centre in Cyber Security (KirCCS), School of Computing, University of Kent, United Kingdom
PATRICE RUSCONI, School of Psychology, University of Surrey, United Kingdom

Cognitive modeling tools have been widely used by researchers and practitioners to help design, evaluate and study computer user interfaces (UIs). Despite their usefulness, large-scale modeling tasks can still be very challenging due to the amount of manual work needed. To address this scalability challenge, we propose CogTool+, a new cognitive modeling software framework developed on top of the well-known software tool CogTool. CogTool+ addresses the scalability problem by supporting the following key features: 1) a higher level of parameterization and automation; 2) algorithmic components; 3) interfaces for using external data; 4) a clear separation of tasks, which allows programmers and psychologists to define reusable components (e.g., algorithmic modules and behavioral templates) that can be used by UI/UX researchers and designers without the need to understand the low-level implementation details of such components. CogTool+ also supports mixed cognitive models required for many large-scale modeling tasks and provides an offline analyzer of simulation results. In order to show how CogTool+ can reduce the human effort required for large-scale modeling, we illustrate how it works using a pedagogical example, and demonstrate its actual performance by applying it to large-scale modeling tasks of two real-world user-authentication systems.

CCS Concepts: • **Human-centered computing** → **Human computer interaction (HCI)**.

Additional Key Words and Phrases: Cognitive modeling, software, simulation, automation, parameterization, CogTool, human performance evaluation, cyber security, user authentication

ACM Reference Format:
Haiyue Yuan, Shujun Li, and Patrice Rusconi. 2021. CogTool+: Modeling human performance at large scale. *ACM Trans. Comput.-Hum. Interact.* 0, 0, Article 0 (2021), 38 pages. https://doi.org/xx.xxxx/xxxxxx.xxxxxx

1 INTRODUCTION

Cognitive models have been proved to be effective and useful to study and investigate human behaviors. Among all, those models that allow estimation of human performance of completing a particular computer-based task are attracting a lot of interest from both research and commercial communities. Cognitive models such as Keystroke-Level Model (KLM) [7] and other models following the GOMS (Goals, Operators, Methods, and Selection) rules [15] are widely used to evaluate human performance and refine UI designs more efficiently without prototyping and user testing [9].

Authors' addresses: Haiyue Yuan, Centre for Vision, Speech, and Signal Processing (CVSSP), University of Surrey, Guildford, Surrey, GU2 7XH, United Kingdom, haiyue.yuan@surrey.ac.uk; Shujun Li, Kent Interdisciplinary Research Centre in Cyber Security (KirCCS), School of Computing, University of Kent, Canterbury, Kent, CT2 7NF, United Kingdom, s.j.li@kent.ac.uk; Patrice Rusconi, School of Psychology, University of Surrey, Guildford, Surrey, GU2 7XH, United Kingdom, p.rusconi@surrey.ac.uk.

1073-0516/2021/0-ART0 $15.00
https://doi.org/xx.xxxx/xxxxxx.xxxxxx

A number of software tools (e.g., CogTool [14, 16], SANLab-CM [25], Cogulator [37]) have been developed to facilitate and simplify cognitive modeling.

CogTool [14] is one of the most popular, open-source cognitive modeling tools being widely used by researchers and practitioners. CogTool and its various extensions have been applied to different domains for both research and industry communities. CogTool, used to model a computer-based task, consists of the following steps: 1) define the UI including the size and position of all widgets and their functionalities; 2) describe how the user would interact with elements of the UI step by step; this process will be referred to as the user-interaction workflow for the remaining of the paper. Then, CogTool translates its high-level inputs into a low-level model following the ACT-R (Adaptive Control of Thought-Rational) architecture [2, 3] written in the common Lisp programming language [40]. It then uses this model to produce a prediction of human performance on the user interface.

It is convenient to model computer-based tasks using CogTool. However, it could be difficult and time-consuming to model complex and dynamic tasks or systems such as the challenge-based user-authentication systems presented in [10, 28, 30, 39], especially for modeling dynamic UIs or user interactions based on randomly generated challenges or user responses.

These are the challenges to scale and extend CogTool's capabilities:

(1) To conduct large-scale modeling tasks (semi-)automatically.
(2) To dynamically update/change default values of cognitive modeling operators and parameters such as those related to Fitts' law whose updating CogTool does not currently support
(3) To build mixed probabilistic models through simple steps.

We discuss these challenges below with greater details.

For the first challenge, let us consider an example of modeling the task of entering a simple 6-digit PIN (Personal Identification Number) to help investigate fine-grained issues such as differences between individual 6-digit PINs, 6-digit PIN groups (weak PIN vs. strong PIN), or inter-keystroke, timing-related cyber attacks [19]. This requires producing up to 10^6 models to cover all possible PINs, as entering each 6-digit PIN results in a different interaction workflow.

For the second challenge, although CogTool allows the user to change the default values of some cognitive modeling operators, it does not support their dynamic updates. Previous research [23, 24, 32, 44] also identified some limitations of having fixed values of cognitive modeling operators, which could potentially affect the accuracy of the predicted user performance time. The latest version of Cogulator [1] allows the user to add new operators, or change the execution time of existing operators without changing the application source code. However, it still lacks support for an automated process, and it requires lots of manual work for large-scale modeling.

Finally, for the third challenge, existing cognitive modeling tools allow the user to simulate different methods to complete a task, however, they do not explicitly support modeling mixed probabilistic models, and they normally require the user to interact with third-party software tools to conduct further analyses.

In this paper, we propose an approach aimed to address these limitations and to improve cognitive modeling tools such as CogTool. We propose a new cognitive modeling software framework and a research prototype software tool, both called CogTool+, which extend the widely used tool CogTool to solve the above-mentioned scalability problems of existing cognitive modeling tools. CogTool+ provides UI/UX researchers and designers with a number of useful key features to model complex, and especially dynamically changing, UI elements and the human performance of the corresponding complex tasks for which they are used.

[1]http://cogulator.io/

CogTool+ is designed for UI/UX researchers, designers and other practitioners as its main end users. As a unique feature, it supports a clear separation of tasks, allowing programmers and psychologists to define reusable components that can be easily used by end users without the need to understand the low-level implementation details of such components. This approach allows a different level of scalability: programmers, psychologists, and end users of CogTool+ can work together in an asynchronous but effective manner to support each other on large-scale human performance modeling tasks. Psychologists can define reusable parameterized behavioral templates based on their theoretical and empirical studies on human cognition, perception, and motion. Programmers can define general-purpose algorithmic components as reusable software modules, e.g., different types of randomization functions that can be used by UI/UX designers and practitioners without any programming experience to model dynamic UIs and other algorithmic parts of a computer system.

The rest of the paper is organized as follows. The next section presents related work. Then, we describe the proposed software framework CogTool+ with implementation details in Section 3, which is followed by a pedagogical example in Section 4 to illustrate the use of CogTool+ for modeling a simple user-authentication system. The evaluation of CogTool+ is discussed in Section 5, using two large-scale modeling tasks of two real-world user-authentication systems. Limitations of our work and future directions are discussed in Section 6 before the final section concludes this paper.

2 RELATED WORK

Human cognitive modeling has been extensively studied and used in the HCI domain. One of the well-established cognitive modeling theories used for designing UIs and predicting human behavior is Goals, Operators, Methods, and Selection rules (GOMS) [9, 15]. A number of variants of GOMS models such as KLM, CMN-GOMS [8], and CPM-GOMS [15] have been widely used for refining the task procedure, predicting task completion time, and discovering UI design issues [24]. Despite their success, there are some limitations and challenges. Previous work [16] reported that HCI interface designers found it relatively difficult to learn and use GOMS-type models in practice. It also remains a challenge to model complex tasks such as user performance on multi-modal UIs in a car navigation system [6, 29]. There are several approaches to respond to these limitations and challenges. The use of software tools to (semi-)automatically facilitate modeling has been the one that attracts more attention.

A number of open source software tools such as CogTool [16], SANLab-CM [25], and Cogulator [37] have been developed, and the integration of low-level cognitive architectures such as ACT-R [1–3] and Soar [18, 33] with these tools makes them capable of modeling more complex and broader types of human cognitive processes. SANLab-CM and CogTool are the most widely-used tools in the HCI community. SANLab-CM is specialized in modeling CPM-GOMS which combines the task decomposition of a GOMS analysis with a model of human resource usage at the level of cognitive, perceptual, and motor operations. SANLab-CM supports low-level, parallel modeling of cognitive processes as well as the prediction of execution time for subtle, overlapping patterns of activities by extremely expert users. Similarly, CogTool has the functionality to simulate the cognitive, perceptual, and motor behavior of humans, and generate predictions of performance/execution time by skilled users to complete computer tasks [16] based on KLM, which is implemented using the ACT-R cognitive framework [1–3]. The dedicated graphical user interface (GUI) of CogTool makes it easier for researchers and designers to annotate design sketches for prototyping and evaluation. Furthermore, other researchers have built other software tools on the basis of CogTool. For instance, Feuerstack and Wortelen [11] used the front end of CogTool to develop the Human Efficiency Evaluator (HEE) to predict the distribution of attention and the average reaction time.

Among all the existing software tools, CogTool has a large number of users, and it has proven to be a useful tool in various research areas. Luo and John demonstrated that the predicted time matches the execution time from actual humans in a study investigating hand-held devices [20]. Teo and John used CogTool to model and evaluate a previously published web-based experiment, and they found that it generated better predictions than any other published tools [36]. More recently, Gartenberg et al. [13] modeled the use of a mobile-health application with two designs of UI. The comparison between two UI models was found to be consistent with the findings from a real human user study.

CogTool is not only the focus of academic research, but also industry. Bellamy et al. [4] compared the usability of a new parallel programming toolkit built on Eclipse with a traditional command line programming editor. The comparison revealed that mouse-based interaction is faster than the programmer preferred keyboard interaction using command line. In their later work [5], researchers from IBM and Carnegie Mellon University worked together to evaluate the integration of CogTool into software development teams to improve the communication and usability analysis within a product team and between a product team and its customers.

Apart from being used in traditional HCI research, CogTool was proven to be useful in cyber security research. Kim et al. [17] used CogTool to evaluate the usability of a shoulder surfing resistant mobile user-authentication system, and Sasse et al. [31] combined CogTool with a user study to estimate the usability of a user-authentication system. More recently, Yuan et al. [44] used CogTool with eye-tracking data to successfully model a user-authentication system. They reproduced some human-related security issues, and discovered some UI design flaws, which were identified in a previous study [26].

In addition, extended versions of CogTool have been developed to support automation and other advanced features. Swearngin's CogTool-Helper [34] supports the automatic creation of frames with no human intervention. However, the automated creation feature works only with existing OpenOffice or Java Swing applications. Considering that one of the main advantages of using cognitive modeling software tools such as CogTool is to model prototypes (even with paper/drawing-based prototypes), CogTool-Helper's approach has its limitations, which were also acknowledged by the developers of CogTool-Helper with the aim of addressing them in their future works. The most similar work to our proposed approach is human performance regression testing (HPRT) built based on CogTool-Helper [35]. HPRT can generate all possible interaction paths, and evaluate human performance predictions for the same task. However, it is relatively difficult to use as it requires specific knowledge of CogTool-Helper, CogTool, and a GUI Testing frAmeworRk (GUITAR) [21]. It could cause problems of fragmentation, which is another issue we would like to address in our proposed approach.

Despite its popularity, CogTool has some limitations. Inherited from the GOMS-type models, CogTool does not support the prediction of the time required by a learning process (i.e., the time taken by an individual to go from the novice through the intermediate and the expert stages [24]), which could be valuable to the design and assessment of UIs. Shankar et al. [32] compared CogTool simulation time with actual user time from lab studies for an enterprise application in an Agile environment. The results suggested that there is a positive correlation between the two. However, they identified that the default 'thinking time' (i.e., 1.2 seconds) in CogTool underestimates the actual 'thinking time' for some specific tasks. This is actually a problem known by the developers of CogTool, so CogTool is designed to allow values of variables such as 'thinking time' to be modified manually by the end user, which is however quite inconvenient to do especially for large-scale modeling tasks. In addition, it would be too time-consuming when there is the need to model all possible interaction workflows using CogTool, which could undermine the CogTool's usability and its reputation of fast prototyping. Furthermore, Yuan et al. [44] identified the need to use external data such as eye-tracking data to guide the design of interaction workflows. Although some default parameters of CogTool such as 'Think' and 'Look-at' can be edited manually, in a comparative study to look at the difference between cognitive modeling and user performance analysis for touch screen mobile interface design, Ocak and Cagiltay [23] suggested that the default 'Think' time should be modified depending on the context of use. They also recommended that the default 'Look-at' time should be adjustable automatically according to the length of the text in a reading scenario.

3 COGTOOL+: A NEW COGNITIVE MODELING SOFTWARE FRAMEWORK

In this section, we describe the new cognitive modeling software framework CogTool+ and its implementation, extended from one of the most well-known open source modeling tools, CogTool [14]. CogTool+ [2] is effectively a framework extending CogTool to support large-scale human performance modeling tasks in a more flexible and reconfigurable way. CogTool+ does not change the low-level cognitive modeling core of CogTool, so it is still based on the KLM model. The overall system architecture of CogTool+ is shown in Figure 1, with the following important key features helping enhance the scalability of CogTool:

- An *enhanced XML schema* to design and define modeling tasks to support a *higher level of parameterization and automation* especially for UIs with dynamically and algorithmically changing elements.
- *Algorithmic components*: Different from existing modeling tools, CogTool+ supports algorithmic components that can dynamically change the UI and human cognitive processes. This is achieved by allowing the software to interface with externally defined executable function, written in JavaScript code in our current implementation.
- Allowing *external data* to be incorporated easily as part of a modeling task. Differently from existing approaches, we designed a flexible way to integrate external data using algorithmic components to better model human cognitive processes.
- Unlike CogTool, but similar to some other modeling tools, CogTool+ also supports designing *mixed models* to reflect the probabilistic nature of many human cognitive processes.
- An *offline analyzer* for supporting data analysis and visualization.
- A *clear separation of tasks* so that computer scientists, programmers and psychologists can provide reusable components to help end users of CogTool+ more easily.

As illustrated in Figure 1, the black icon of the human silhouette and a white board indicates where human users can be involved in the working flow. Users can use the *Model Generator* to design models. Next, the *Model Interpreter* and the *Model Simulator* can process the user-generated model to produce simulation results automatically. Users can supply these results to the *Offline Analyzer* to visualize and review the simulation. In addition, users can provide *external data* to each component of CogTool+ when necessary.

To use CogTool+, the user does not need to have expertise in programming, but she/he just needs to be able to use written software modules by following instructions (e.g., how to use a random function from a graphical user interface). Psychology-informed elements such as 'Think' and 'Homing' supported by CogTool are still supported by CogTool+. In addition, external data such as those from behavioral studies in experimental psychology (e.g., visual-search behavioral and eye-tracking data, Fitts's Law distribution data) and data from previous related literature can be used to interface with CogTool+ in order to drive and guide the modeling process. In addition, computer scientists and programmers can package external data and develop reusable algorithmic modules that can form part of behavioral templates and data sets to add values to CogTool+.

[2] Code is available at https://github.com/hyyuan/cogtool_plus

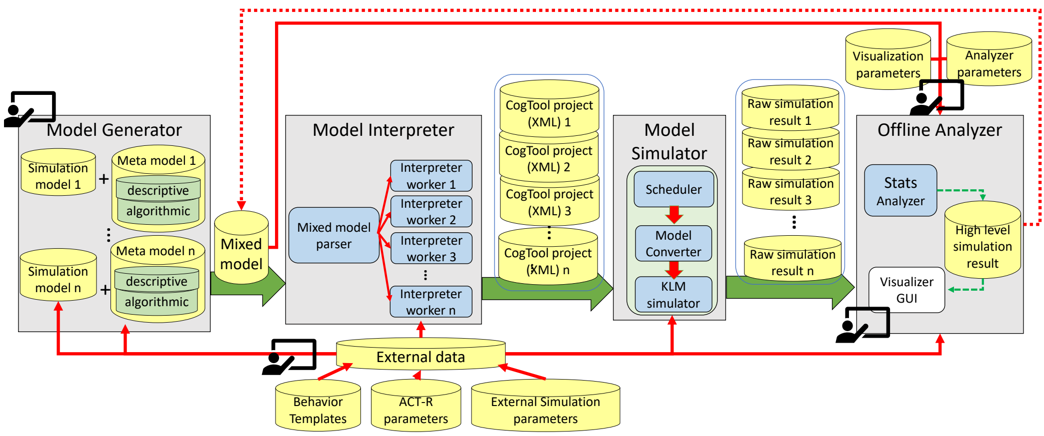


Fig. 1. The system architecture of CogTool+ with key components and processes

The rest of this section presents more details of the system architecture and provides examples to facilitate a better understanding of the different features of CogTool+. All the examples used in this section are parts of a more complicated modeling tasks on 6-digit PIN entries, which will be detailed in Section 5.1.

3.1 Model Generator

The model generator is responsible for the description of the system UI and user-interaction tasks in the form of simulation models, meta models, and mixed models, all using a human- and machine-readable language.

3.1.1 Simulation models. One simulation model sets parameters to facilitate the design of one meta model, and also contains information to configure the simulation process. Composing a simulation model consists of three steps:

(1) To define the total number of simulations that need to be carried out for a particular task (i.e., the value defined using `<trial>` as illustrated in Figure 2).
(2) To configure the simulation setting. This is defined using the `<pref-setting>` element as illustrated in Figure 2. There are many options for configuring simulations settings, which we discuss below.
(3) To define any external variable from external data sources that will be used in a later stage of the modeling process. As illustrated in Figure 2, 100 random 6-digit PINs saved in the 'PINs.csv' file are defined as an 'ArrayList' variable with the ID of 'externalPin'.
In addition, we can use external data to drive the generation of `<fitts_cof>` and `<fitts_min>` such as loading predefined values stored in external files.

For instance, we can configure `<fitts_cof>` and `<fitts_min>`. These two parameters correspond to the two coefficients in the Fitts Law [12] equation. As shown in Figure 2, having a `<type>dynamic</type>` setting, `<fitts_cof>` produces a Gaussian distribution with mean of 50 and standard deviation of 1.0, and `<fitts_min>` produces a Gaussian distribution with mean of 75 and standard deviation of 1.5. The size of the generated distribution is determined by the number of trials defined at the beginning of the simulation model (i.e., `<trial>100</trial>`). On the other hand, a static `<type>` can be used to assign fixed values to these two parameters (i.e., 48 and 136,

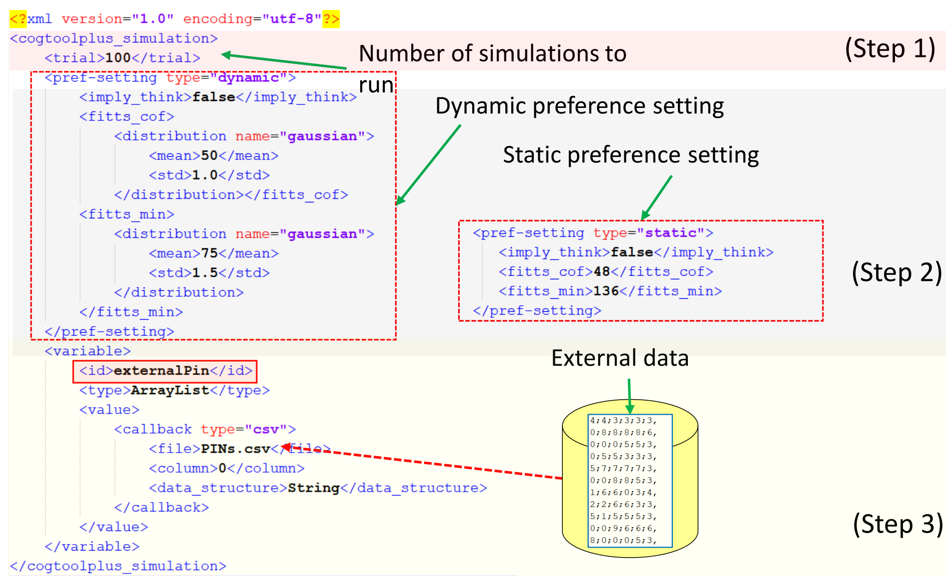


Fig. 2. An example of simulation models written in XML.

respectively, as shown in the Figure 2). More details about the implementation to achieve these can be found in Section 3.5.

In addition, other parameters can be configured in step 2. For instance, CogTool has a default 1.2 seconds of thinking time automatically added to the first demonstration step or a first 'Look-At' step.

There are two ways for the designer to modify the value of thinking time using CogTool. One is to manually change the value when defining the 'Thinking' variable the first time. Another one is to update the value manually in the 'Script Step List' from the CogTool interface, where 'Script Step List' is used to let the designers define the interaction workflow. If there are multiple 'Thinking' variables, it will require the designer to manually update them all one by one. Although it would be possible to update it/them programmatically and dynamically using CogTool, it would involve programmers to work with CogTool's source code to provide additional features. This is where CogTool+ makes the difference. CogTool+ does it in a programmatic way by using algorithmic elements. Designers/users can use the proposed XML language to compose higher-level descriptions of interaction workflow as well as defining and ingesting parameters such as 'Think' and 'Look-at' dynamically. Parameter definition should be informed by previous research. An example comes from the psychological literature on visual search showing that individuals' search times for a target can occur within 1 second [38, 41].

As shown in Figure 2, the element `<imply_think>` is used to give users/designers the control over disabling/enabling the default 'Thinking' step. In addition,the `<call_back>` function can be added here to allow CogTool+ to dynamically assign values to 'Think' step to increase the level of automation.

It is worth emphasizing that any changes to the parameters defined at step 2 should be based on empirical evidence, for example they can be informed by psychological behavioral studies depending on different systems/use cases/scenarios.

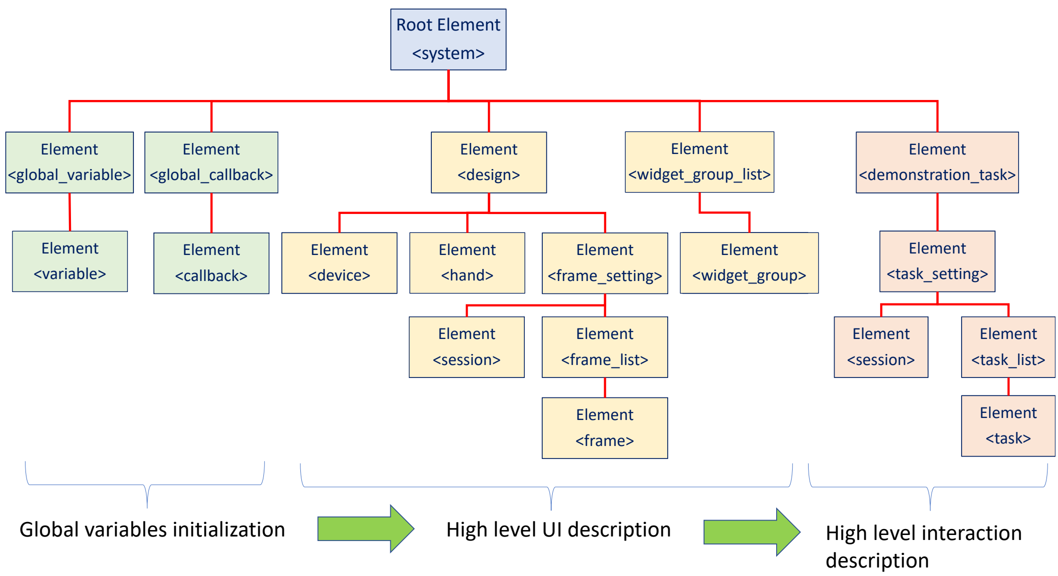


Fig. 3. The XML tree structure of a descriptive meta model.

3.1.2 *Meta models.* A meta model is used to define high-level UIs and interaction workflows. It consists of two sub-models: a *descriptive model* and an *algorithmic model*. Below, we will present detailed explanations of our implementations with examples.

Descriptive models. A descriptive model is responsible for defining the high-level UI elements and the high-level user interactions, and it describes the interface to communicate with its associated algorithmic model. We designed an XML-based human-machine readable language to construct a descriptive model. As illustrated in Figure 3, a descriptive model consists of three building blocks: *global variable initialization*, *high-level UI description*, *high-level interaction description*. The arrows between them indicate the sequential order of building a descriptive model. The process always starts with *global variables initialization*, and ends with *high-level interaction description*. Each building block has a number of elements with their children elements to support specific tasks. Elements in green define global variables, elements in yellow and elements in red describe UI-related components and user-interaction-related components, respectively.

(1) *Global variables initialization*: In a descriptive model, global variables need to be initialized, so that they can be referred to at a later stage. A `<global_variable>` usage example is presented later to demonstrate its usage.
(2) *High-level UI description*: For this building block, the user needs to describe the UI in a relatively abstract way. The global variables defined earlier can be used here to derive a more detailed description of UI elements when it is parsed to a model interpreter 3.2.
    - `<design>`: This element and its child elements deal with the high-level description of UIs. `<device>` indicates the main devices used for the interaction such as mouse or touch screen. `<hand>` identifies which hand will be used for the modeling and simulation. `<frame_setting>` defines the general setting of how to describe UIs at a high level. `<frame_setting>` has a list of `<frame>` defined in `<frame_list>`, where each frame represents the graphical representations of a specific UI. `<frame_setting>` can be set to

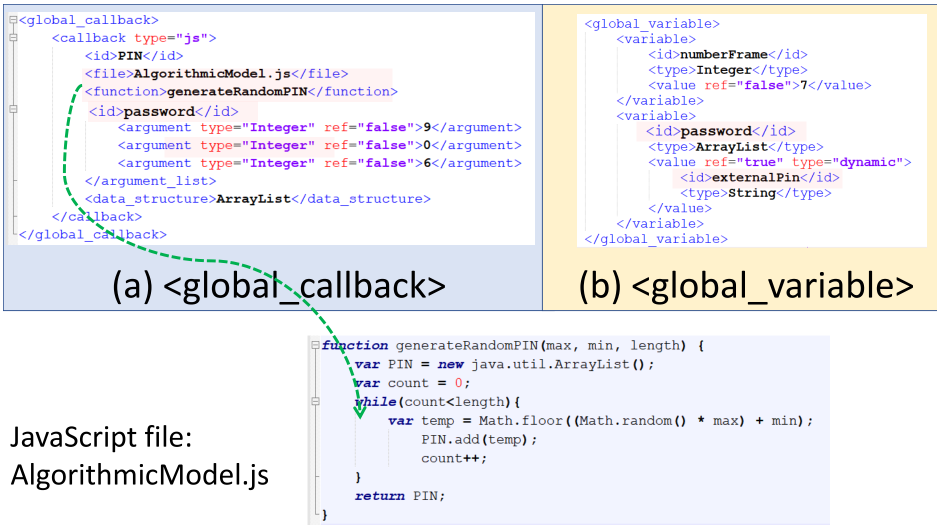


Fig. 4. Example of using <global_callback> and <global_variable> to create random 6-digit PINs

'dynamic' or 'static' using its attribute <type>. If it is set to 'dynamic' , the model interpreter can interpret the high-level model of UI defined in the frame, and dynamically and automatically convert it to one or more different low-level descriptions of UI depending on the user setting. This cannot be achieved using CogTool easily, which requires the user to define all frames manually. A <frame_setting> usage example is provided later to show the modeling details using CogTool+ to achieve this.

- <widget_group_list>: It categorizes similar widgets into groups for further use.

(3) *High-level interaction description*: Coarse user interactions need to be defined in this building block. Similar to the *high-level UI description*, global variables and functions in the algorithmic model can be utilized to derive low/atomic level user-interaction steps using a model interpreter 3.2. A <demonstration_task> contains a <task_setting>, which consists of <session> element and <task_list> element. An interaction workflow is defined in <task_list> including of a number of <task>. Each <task> describes an atomic interaction action such as 'look at', 'mouse click', or 'tap'. Same as the <frame_setting>, <task_setting> can be 'dynamic' if the user needs to model dynamic user interactions. It should be noticed that in the original CogTool project, such atomic actions could only be implemented in a single widget. This can be achieved using the 'static' <type> attribute in CogTool+ as well. Unlike CogTool, the user can assign an atomic action to a group of widgets that are defined in <widget_group_list> using CogTool+, which will need to work together with a dynamic <frame_setting>. In addition, for each <task>, the user can define some <callback> (i.e., the same as the one used in <global_callback>) interacting with the algorithmic model to get dynamic inputs. A <task_setting> usage example is presented later to illustrate the process of defining high-level user interactions.

<global_variable> *usage example.* Here we present an example of using two approaches to create 100 6-digit PINs as illustrated in Figure 4.

The first approach is to utilize the `<global_callback>` function to work with the algorithmic model. A `<global_callback>` can have multiple child `<callback>` elements, where each one describes how to communicate with the accompanying algorithmic model. It has an attribute `<type>`, which can be set to either 'js' and 'csv'. 'js' suggests that `<callback>` will call and compile a JavaScript function defined in the algorithmic model and return the value, whereas 'csv' indicates that `<callback>` will read a Comma-Separated Values (CSV) file and return the value. All values returned from this part are considered as global variables.
As illustrated in Figure 4 (a), using `<global_callback>`, a global variable with the ID of 'password' is created by calling and compiling a JavaScript function `generatedRandomPIN()` that is defined in the `AlgorithmicModel.js` file. The integers '9' and '0' representing the range of PIN digits, and the integer '6' representing the length of the PINs are described using `<argument>` elements to assign input arguments to the JavaScript function to generate one random 6-digit PIN, where each digit is an integer between 0 to 9. Input with the trial number (i.e., `<trial>100</trial>`) defined in the simulation model (see Figure 2, CogTool+ can automatically generate 100 random 6-digit PINs for further use.

The second approach to generate 100 random 6-digit PINs is to use `<global_variable>`. Similar to the definition in any other computer programming languages, global variables defined in this part will be available for use during the entire modeling process. As shown in Figure 4 (b), two global variables are created. One has the ID of 'numberFrame' and value of 'Integer' 7. Another global variable has the ID of 'password'. By setting the `ref` attribute of `<value>` to be 'true', the value of this variable is the 'externalPIN' variable created earlier using the simulation model (see Figure 2), which contains 100 random 6-digit PINs as mentioned in Section 3.3.

`<frame_setting>` *usage example.* Here, we present a simple example as illustrated in Figure 5 to demonstrate how to use 'dynamic' `<frame_setting>` with the global variable created in the `<global_variable>` usage example to describe the UI for a 6-digit PIN entry task.
First, the objective is to convert the graphical representation of the UI (i.e., Figure 5 (b)) to the high-level description of UI (i.e., Figure 5 (c)) using XML. Figure 5 (a) shows snippets of the XML code. For instance, the highlighted `<widget>` elements define features such as type, size, and position for the buttons 'slash' and 'minus'. In addition, widgets with similar properties can also be categorized together using `widget_group_list` and `widget_group` elements. As shown in Figure 5 (a), the 0-9 number buttons are grouped as a widget group with the ID of 'enter pin' as highlighted. Then, we can recall the global variable 'numberFrame' defined earlier in the `<global_variable>` usage example. The attribute `type` of `<frame_setting>` is set to be 'dynamic'. Together, this allows CogTool+ to automatically generate low-level descriptions for seven (i.e., 'numberFrame' has the value of 'Integer' 7) frames (see Figure 5 (c)). Hence, it is possible to conduct fine-grained analyses such as the inter-keystroke time difference, where each frame corresponds to one step of the user interaction that could be either pressing a digit key or the <Enter> key.

`<task_setting>` *usage example.* As shown in Figure 6, the `task_setting` is set to be dynamic. The global variables 'numberFrame' and 'password' defined in the `<global_variable>` usage example and the widget group ‘enter pin’ defined in the <frame_setting> usage example can be referred to in order to facilitate creating a series of button tapping events (i.e., <type>tap</type>).

Algorithmic models. In CogTool+, an algorithmic model is written in JavaScript. Such models make CogTool+’s parameterization and automation of the modeling process possible. Algorithmic models are “plug-and-play” components that give users/designers the freedom to add external data to a descriptive model, as shown in Figure 1. For instance, to model more complex conditional

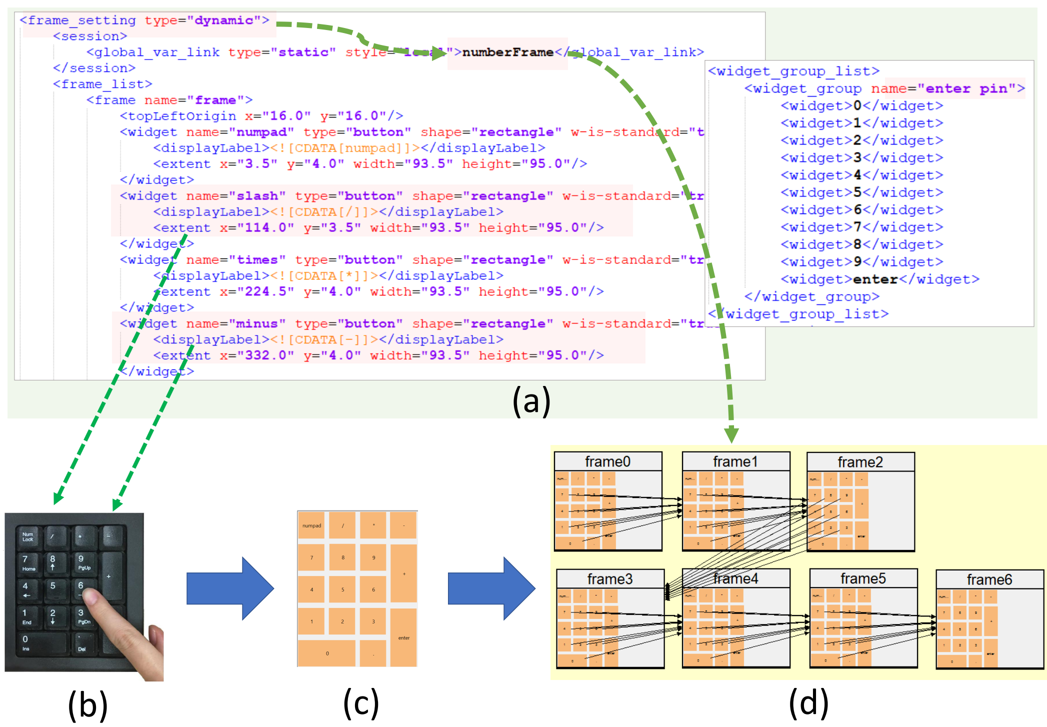


Fig. 5. Example of using ‘dynamic’ <frame_setting> to describe the UI for the PIN entry task

```
<demonstration_task start_frame = "frame0" name = "demo">
  <task_setting type="dynamic">
    <session>
      <global_var_link type="static" style="local">numberFrame</global_var_link>
    </session>
    <task_list>
      <task name = "t1" type = "group">
        <widget_group>enter pin</widget_group>
        <touch_screen>
          <targetIndex ref="true">
            <global_var_link type="dynamic" style="local">password</global_var_link>
          </targetIndex>
          <type>tap</type>
          <delay>0.0</delay>
        </touch_screen>
      </task>
    </task_list>
  </task_setting>
</demonstration_task>
```

Simulation model

High level UI description/Descriptive model

Fig. 6. XML code example to describe high-level user interactions

interactive systems, the user can program a JavaScript function, which will be compiled using the model interpreter to generate a dynamic interaction workflow in a recursive and iterative way, rather than having to design it manually step by step.

Furthermore, if the user of CogTool+ is not familiar with programming in JavaScript or any other programming languages, an alternative way is to utilize a data format such as CSV, XML or JSON to reconfigure pre-defined algorithmic models that CogTool+ supports. For instance, in our current implementation, the CSV format is used to store predefined data in a CSV file, and a parser follows a simple syntax to read the data in the CSV file to define the meta model demonstrated in the example shown in 3.1.2. This approach is just an indicative example and can be easily generalized to use other data formats or to allow the parser to use such data files in other different ways. The model interpreter can process it to create dynamic designs.

CogTool+ is designed to be backward-compatible with CogTool. As illustrated in Figure 1, the generated data from the model interpreter is a series of CogTool compatible cognitive models. CogTool+ inherits CogTool's pipeline of converting these cognitive models to low-level Lisp scripts, simulate, and produce atomic-level predictions. In other words, the powerful predictive ability of CogTool remains in CogTool+.

In addition, algorithmic models allow more elements/modules to be injected and integrated with CogTool+ to support large-scale human performance modeling tasks. These added elements including algorithmic module libraries and behavioral templates database are made transparent to users who do not need to know the internal functioning of such elements.

We have demonstrated how an algorithmic model written in JavaScript can work together with the descriptive model to define global variables in Section 1. Later in this paper, we will present more examples to demonstrate how the descriptive, algorithmic, and simulation models work together.

3.1.3 Mixed models. A mixed model is a mixed-probabilistic model consisting of a number of meta models with their own probabilities. Here we present a use case of a mixed model to explain its concept and illustrate our implementation. The modeling task is to predict the overall performance of completing a 6-digit PIN entry task using the PIN pad as shown in Figure 5 (a). Three different input devices (touch screen, keyboard, and mouse) can be used to complete this task. It is assumed that 10% of the sampling population is left-handed and 90% is right handed for both touch screen and mouse users. Also, the percentages of users using three input devices are assumed to be 40%, 30%, and 30%, respectively. To complete this task using CogTool+, it only needs to design individual meta model for each subset of users, and then build a mixed-probabilistic model consisting of all individual meta models with their probabilities as illustrated in Figure 7.

A light blue block in the figure represents a meta model, a dark blue block represents a sub-mixed model, and a green block represents a mixed model. A sub-mixed model can consist of several meta models, or a number of sub-mixed models, or a mixture of meta models and sub-mixed model. The mixed model at the top level has the same property as the sub-mixed model, but it is the root node of the modeling tree. The implementation of a mixed model uses XML. By using such mixed models, we could better understand the overall average behavior as well as the performance of any subsets of users. However, it should be noted that the main aim of supporting mixed models is to provide options for further analysis. Users can still use CogTool+ without defining mixed models, and users should be aware that more work will be incurred for designing mixed models and conducting further data analysis.

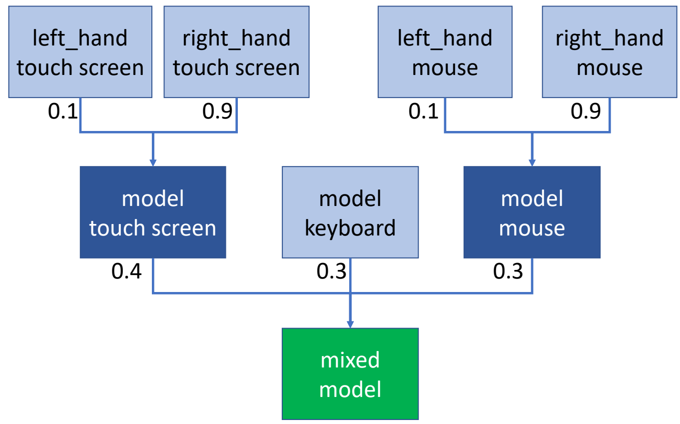


Fig. 7. The tree-like structure of an example of complex mixed models.

3.2 Model interpreter

The model interpreter takes a mixed model or a meta model (which can be seen as a mixed model with just one meta model) as the input. When a mixed model is the input, the model interpreter uses a mixed model parser, which is a customized XML parser, to understand the composition and structure of the mixed model. This is followed by the allocation of the interpreter workers for the analysis of each individual meta model with its accompanying simulation model. Finally, these interpreter workers generate a number of CogTool-compatible projects written in XML.

Each interpreter worker consists of an XML parser and a translator as illustrated in Figure 8, and each XML parser contains a core processor and a dynamic parser. The implementation of the core parser is similar to a Document Object Model (DOM) XML parser, which loads the complete contents of the simulation model and descriptive model, and creates a complete hierarchical tree in memory.

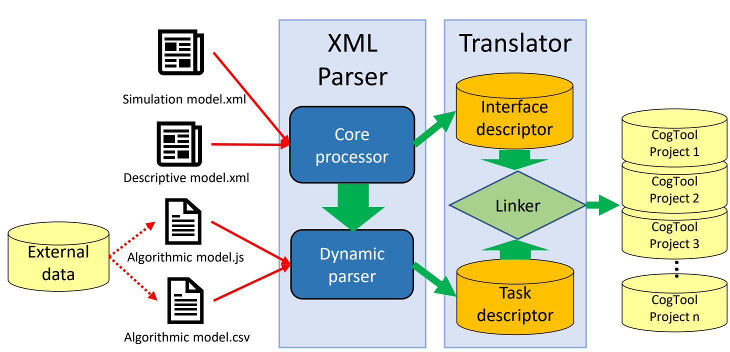


Fig. 8. The internal structure of the interpreter worker

By scanning this, the core processor classifies and redirects the high-level UI description and high-level user interaction description to the interface descriptor and the dynamic parser, respectively. The interface descriptor processes and translates high-level descriptions to low-level descriptions of UIs such as layout of the UIs, size of widgets, position of widgets etc. Then the dynamic parser reads the algorithmic models, and use different classes to process them based on the model type

```
CogToolPlusCSVParser parser = new CogToolPlusCSVParser();
switch (dataStructure) {
    case "Double":
        callback.setResult(parser.DoubleArrayReadCSV(file).get(row));
        break;
    case "Integer":
        callback.setResult(parser.IntegerArrayReadCSV(file).get(row));
        break;
    case "String":
        callback.setResult(parser.StringArrayReadCSV(file).get(row));
        break;
}
```

(a)

```
ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("JavaScript");
engine.eval(new FileReader(file));
Invocable inv = (Invocable)engine;
output = dynamicInvokeFunction(inv, function , inputArguments);
callback.setResult(output);
```

(b)

Fig. 9. Selected source code of the dynamic parser that processes algorithmic models written in (a) CSV format and (b) JavaScript format

(i.e., JavaScript or CSV). As illustrated in Figure 8, external data can also feed into an algorithmic model.

Figure 9 illustrates how the dynamic parser works at the source code level. Figure 9 (a) shows a few lines of code that reads a CSV file and parse the value based on the defined data type to the `callback` object using `CogToolPlusCSVParser` class. Figure 9 (b) demonstrates how to use an existing Java Class `ScriptEngineManager` to dynamically compile a function written in a JavaScript file given a number of arguments (e.g., see <argument_list> in Figure 10(a)) using `dynamicInvokeFunction`, and then return the value to `callback` object. Finally, the dynamic parser sends these returned values saved in `callback` objects with high-level user interaction description to the task descriptor. Then the task descriptor interprets and converts them to low-level user interaction description (i.e., atomic-level interaction steps). Next, the linker is used to integrate the low-level description of UIs and user interactions to produce a number of CogTool projects written in XML. Each converted CogTool project is stored locally, so that its validity and modeling details can be independently evaluated and reviewed.

3.3 Model simulator

The main task of a model simulator is to run computer simulations and collect results of user performance predictions. As shown in Figure 10, the scheduler arranges the order of processing [3] and it sends the schedule to the model converter and the KLM simulator. The model converter takes a number of CogTool projects/tasks and convert each one into a cognitive model using a back-end ACT-R framework written in common Lisp [40] programming language. Then the KLM simulator takes the converted ACT-R models and it runs the simulation to produce the simulation trace in

[3]the current implementation only supports sequential processing, but we will implement parallel processing in a future version

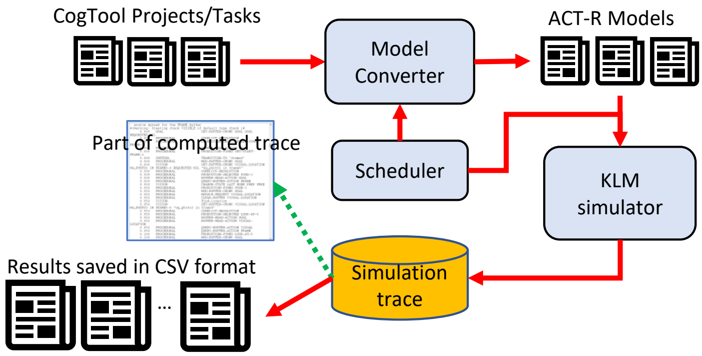


Fig. 10. The flowchart for demonstrating the working pipeline of the model simulator.

terms of completion time for each atomic task, which contains detailed information about the user performance prediction (e.g., overall time, time per operator such as cognition, vision, motor etc.). Finally, these simulation results are saved locally in the CSV format.

3.4 Offline analyzer

According to the specification given in the mixed model, user-defined visualization parameters, and analyze parameters, the offline analyzer post-processes raw simulation results to produce high-level simulation results for the user to review. It should be noted that all meta models are interpreted and simulated to produce user performance predictions without considering their probabilistic information defined in the mixed model. In other words, they are independent of the mixed model to some extent. One of the advantages of this approach is that the user can have a certain freedom to modify the design of the mixed model to post-process raw simulation results without the risk of re-doing the whole simulation, which offers an easy way to have iterative refinement and review. This is consistent with the nature of modeling human cognitive processes that involves iterations of design and simulation. We will present more details of the analysis of simulation results in Section 5.2.2.

We implemented a stats analyzer and a visualization GUI as the main software modules of the offline analyzer.

Stats analyzer. The stats analyzer collects raw simulation results, and post-processes these data by incorporating the analyzer parameters. For instance, the user could adjust the analyzer parameters to instruct the stats analyzer to produce predicted time information for a particular atomic action involving a specific element of the UI. The generated high-level simulation results are stored locally in the CSV format, and they will be further used to facilitate the data visualization process.

Visualization GUI. The implementation of the visualization GUI combines the use of JFreeChart [22] and Processing [27], providing an interactive platform to view and manipulate simulation results. As demonstrated in Figure 1, visualization parameters are needed to indicate the type of visualization (e.g., bar chart and/or histogram) and data sources (e.g., which part/element of the modeled system needs to be visualized). There are two main features of the visualization: one is to show the tree structures of a given mixed model; another one is to allow users to view a bar chart and/or histogram of any node in the tree structures based on the user-defined visualization parameters. It should be noted that the visualization process is independent of the simulation and prediction processes, meaning that the change of visualization parameters could not affect any prior processes

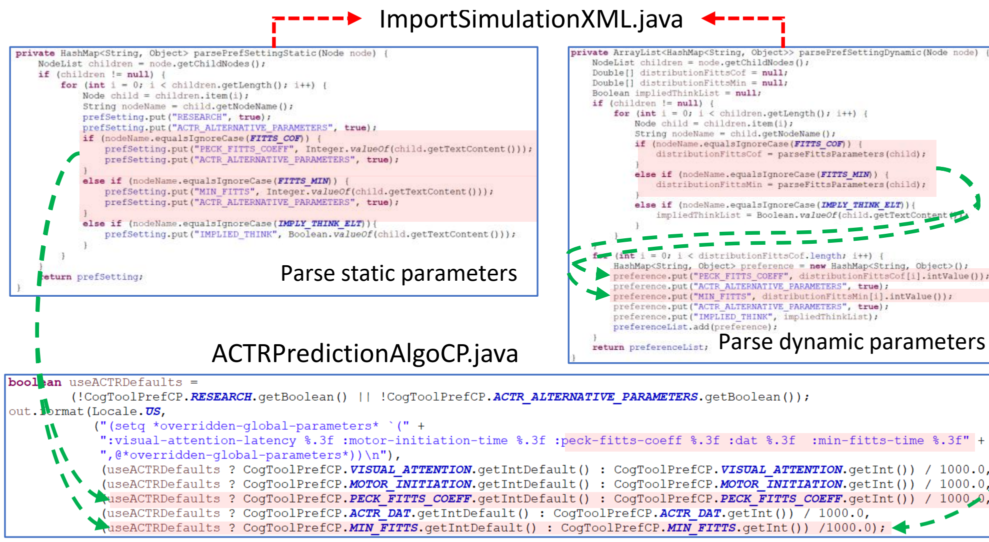


Fig. 11. Snippets of code that deals with the modification of Fitts's Law parameters

although it will produce a different visual content. We will present more details and examples in Section 5.2.2.

3.5 External data

One of the key features of CogTool+ is to allow the software to work with external data to guide and help modeling and simulation. As briefly mentioned in the previous sections, the design of human- and machine-readable language allows users to use `callback` in the descriptive model to link external data generated by either an algorithmic model (via JavaScript or CSV) or direct input.

Our implemented research prototype of CogTool+ currently supports three types of external data: behavioral templates database, ACT-R parameters, and external simulation parameters. Previous research [44] has shown that eye-tracking data can reveal human behavioral patterns that could affect the human cognitive modeling tasks. Such insights extracted from eye-tracking log data could be programmed as reusable behavioral templates to run within CogTool+ to facilitate cognitive modeling tasks. The current behavioral templates are described in JavaScript based on a manual analysis of empirical studies and results from previous relevant research. However, as part of our future work we will develop methodologies and tools to automatically extract and construct behavioral templates from experimental data such as eye-tracking and EEG data.

Some of the ACT-R parameters have fixed values in CogTool. Although some parameters can be modified by enabling CogTool's 'CogTool Research Commands' option, there are still a number of limitations as reviewed in Section 2. The design of CogTool+ allows users to have external data source to initiate/amend such parameters to better and more flexibly define and model human cognitive tasks. For instance, the user could conduct empirical experiments to get more realistic Fitts's Law parameters, and then use them in the modeling process. As mentioned in Section 3.3, this can be achieved using the simulation model to define static and/or dynamic parameters. Figure 11 shows our implementation at the code level to allow the modification of Fitts's Law parameters.

`ImportSimulationXML.java` parses the simulation model, converts all variables, and saves them to the `prefSetting` object. The 'prefSetting' object saves all configuration parameters for the modeling and simulation process. As highlighted in `ImportSimulationXML.java` (see Figure 11), the function `parsePrefSettingStatic()` and the function `parsePrefSettingDynamic()` are used to parse static preference setting and dynamic preference setting respectively. The former allow updating of the Fitts's Law parameters with fixed values, and the latter assigns dynamic values such as distribution to Fitts's Law parameters as mentioned in Section 3.1.1. As highlighted in `ACTRPredictionAlgoCP.java` (see Figure 11), a new variable `MIN_FITTS` is added to `CogToolPrefCP` class to link the corresponding element in the ACT-R architecture implemented in Lisp and written as `min-fitts-time %.3f`. As shown in Figure 11, if the value of `CogToolPrefCP.PECK_FITTS_COEEF` or the value of `CogToolPrefCP.MIN_FITTS` is modified, `ACTRPredictionAlgoCP.java` can modify them in Lisp at the back end.

In addition, external simulation parameters are allowed to work with the *Offline Analyzer* to configure and manipulate post-processed high-level simulation results. We will present more details of integrating external data with the modeling and simulation processes in Section 5.

4 A PEDAGOGICAL EXAMPLE: MODELLING A SIMPLE GRAPHICAL USER-AUTHENTICATION SYSTEM

In this section, we present a pedagogical example to illustrate the process and the typical workload involved when using CogTool+ to model a system. In this example, we will create a mixed model, a simulation model and two meta models to model 150 different users using a simple graphical user-authentication system. Half of the 150 users are left-handed, and the other half are right-handed.

This system is a simplified version of an observer-resistant password system (ORPS) named 'Undercover' [30]. As the main objective here is to demonstrate the model creation process using CogTool+, we do not present simulation results in this section. We did model the full Undercover system, and all modeling details and simulation results can be found in Section 5.2.

4.1 Understanding the system

Undercover is developed based on the concept of partially observable challenges. To use Undercover [30], the user needs to complete the following tasks:

- To set five secret pictures called 'pass-pictures' as the password from a set of images.
- To respond to seven challenge screens, whereby each challenge screen consists of a hidden challenge and a public challenge:

(1) Given a hidden challenge [4], the user needs to obtain a hidden response which is the position index of the pass-picture in the public challenge (1-4 if present and 5 if absent) to respond to a challenge screen.
(2) To look for a hidden response in the correct hidden challenge button layout to get a new position index.
(3) To press the button corresponding to this new position index in the response button panel as shown in Figure 12 (b3).

For instance, one picture identified as the 'pass-picture' in Figure 12 (a) is at position 2. Then the track ball sends a 'Left' signal to the user's palm. The user needs to look at the left button layout in Figure 12 (b2), and then work out the position of the index of the 'pass-picture' (i.e., number 2), which is in the fifth position. The final step is to press number 5 in Figure 12 (b3). More details and other security settings can be found in [26, 30].

[4] The hidden challenge is transmitted to the user's palm via a haptic device (a track ball) as shown in Figure 12 (b1). Five different rotation/vibration modes of the track ball represent five different values: 'Up', 'Down', 'Left', 'Right', and 'Center' (vibrating). Four pictures and a 'no pass-picture' icon form a public challenge as shown in Figure 12 (a). As demonstrated in Figure 12 (b2), each hidden challenge value corresponds to a specific layout of five response buttons labeled 1-5.

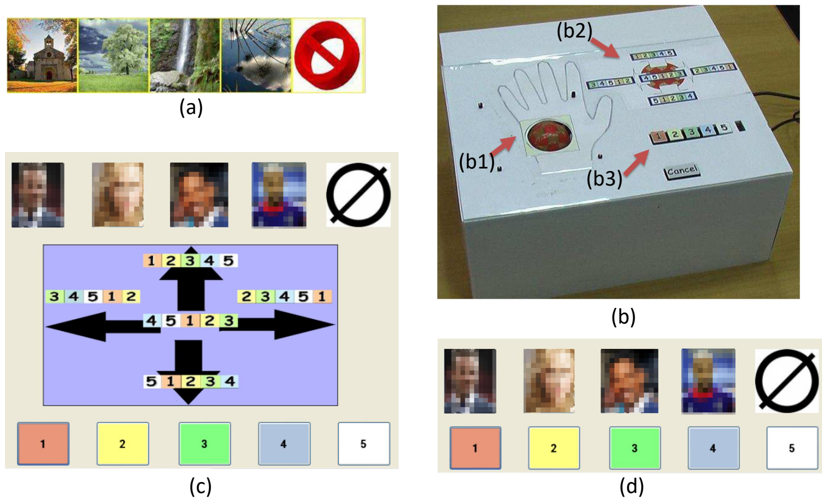


Fig. 12. The UI of Undercover: (a) the public challenge panel shown on the computer display [30]; (b) a box composed of the following UI components [30]: (b1) a track ball to transmit the hidden challenge, (b2) the hidden challenge button layout panel, (b3) the response button panel (c) implementation of Undercover from Perković [26] (d) simplified version of the Undercover system for the pedagogical example

For this pedagogical example, we decided to use a simplified version of the Undercover system as depicted in Figure 12 (d) to demonstrate the modeling workflow using CogTool+. The user interactions to model are simplified as follows: for each challenge, the user needs to identify whether the 'pass-picture' is presented or not, and subsequently complete the challenge accordingly; if one 'pass-picture' is present, the user needs to press one button from position '1' to '4' based on the position of the 'pass-picture'. If a 'pass-picture' is absent, button '5' needs to be pressed.

Using CogTool to model one person using this system would start by creating a CogTool project with a CogTool task. Each CogTool task would start by converting the GUI of the system to CogTool frames, followed by demonstrating the user interaction, where the user needs to click on each CogTool frame via the CogTool Design interface to produce demonstration scripts. Then the CogTool can compute and generate the simulation results automatically. Bear in mind that preparation work such as the selection of 'pass-pictures' and the arrangement of the seven challenge screens needs to be carried out in advance to the hands-on modeling process as mentioned above.

Different from using CogTool, the first step of using CogTool+ is to have a more in-depth understanding of how the system works at a higher level. The user needs to look at how to better include the preparation work as part of the modeling process as well as how to model and simulate at scale (i.e., 150 users). As depicted in Figure 13, the simulation model can instruct CogTool+ to model 150 users. Then the mixed model can incorporate the mixed probability information into the modeling and simulation process. To model each individual user, the meta model deals with the following four sub-tasks, where sub-task 3 and sub-task 4 need to be carried out for all seven challenge screen generated by sub-task 2.

- Sub-task 1: Five 'pass-pictures' should be selected from 28 pictures.
- Sub-task 2: There are seven challenge screens in total. For five of them, each challenge screen contains one unique 'pass-picture', while other two challenge screens have no 'pass-picture'. In addition, the decoy pictures for each challenge screen should be different.

- Sub-task 3: As the selection of 'pass-pictures' and then arrangement of seven challenge screens are known, the position of the 'pass-picture' for each challenge can be derived.
- Sub-task 4: Given the presence/absence of the 'pass-picture', one button needs to be pressed from the response panel.

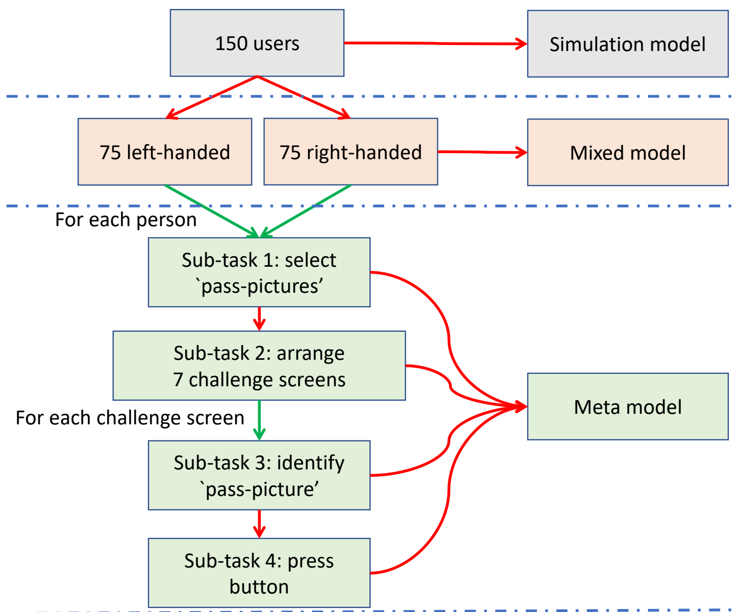


Fig. 13. Flowchart of CogTool+ models design process.

4.2 Creating a simulation model

The requirement is to model 150 users using this system. Hence, we need to produce 150 models and compile 150 simulations. As illustrated in Figure 14 (a), the `<trial></trial>` is set to be 150. Based on the observations of the eye-tracking study we conducted [44] and other previous psychological studies that show how visual search times can occur even within 1 second [38, 41], we argue that the default 1.2 seconds of thinking time might be overestimated depending on the user task. We believe that the thinking time should be dynamic and follow a distribution of values. Instead of using the default 'Thinking' time, we can thus add customized timing information to the meta model to better model the system[5]. Hence, the `<imply_think></imply_think` is set to be false so that the 1.2 seconds 'Thinking' step will not be automatically added.

As there is no need to dynamically change the simulation settings, the attribute `type` of `<pref-setting>` is set to be false.

4.3 Creating a mixed model

As illustrated in Figure 14 (b), the 'mixed_model' has two meta models with equal weight of 0.5. One is named as 'Left-Hand-Model', and another one is named as 'Right-Hand-Model'. To define the preferred hand is straightforward using the descriptive model (see Figure 3) by setting the

[5]More details can be found in Section 5.2.1, where JavaScript function `getScanPath()` and `getThinkTime()` are used to add dynamic timing information to the modeling process

```xml
<cogtoolplus_simulation>
    <trial>150</trial>
    <pref-setting type="static">
        <imply_think>false</imply_think>
    </pref-setting>
</cogtoolplus_simulation>
```

(a) The simulation model written in XML.

```xml
<cogtoolplus_mixed>
    <name>pedagogical_example_demo</name>
    <level>
        <property>1</property>
        <id>mixed_model</id>
        <model_list>
            <simulated_level_model>
                <id>Left-Hand-Model</id>
                <weight>0.5</weight>
            </simulated_level_model>
            <simulated_level_model>
                <id>Right-Hand-Model</id>
                <weight>0.5</weight>
            </simulated_level_model>
        </model_list>
    </level>
</cogtoolplus_mixed>
```

(b) The mixed model written in XML.

Fig. 14. Example of the simulation model and the mixed model

value of the <hand> element to ‘left’ or ‘right’. The offline analyzer further down to the system architecture (see Figure 1) can utilize the mixed probability information to produce simulation results accordingly.

4.4 Creating a meta model

Apart from the difference of defining the preferred hand, the rest of the ‘Left-Hand-Model’ meta model is identical to the rest of the ‘Right-Hand-Model’ meta model. Figure 15 demonstrates the interaction between the descriptive model and the algorithmic model of a meta model. As described in Section 3.1.2, a descriptive model has three parts: global variable initialization, high-level UI description and high-level interaction description.

The *global variable initialization* completes sub-tasks 1 and 2. The algorithmic model provides JavaScript functions `generatePassPicture()` and `arrangeChallenge()` to support modeling the dynamic elements. Figure 16 shows the snippets of the XML code.

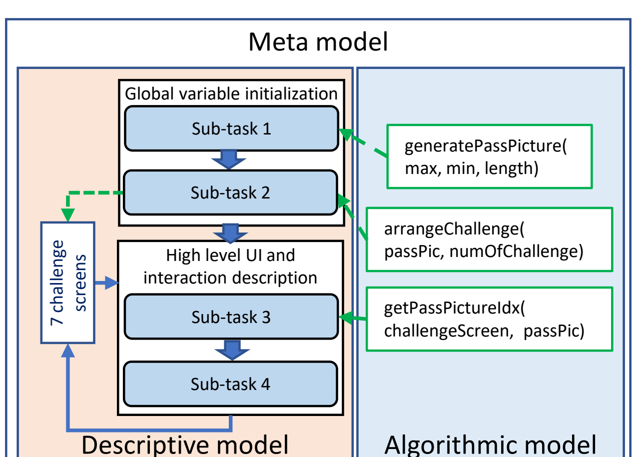


Fig. 15. The meta model: the descriptive model and the algorithmic model

<global_variable> creates a global variable with ID of ‘numChallenges’ and value of integer ‘7’. Then <callback> is used to call the JavaScript function `generatePassPicture()` from the algorithmic model and define three input arguments, where 28 represent the maximum integer value, 1 represents the minimum integer value, and 5 represents five random non-repeated integers. The model interpreter can call the `ScriptEngineManager` as described in Figure 9 (b) to evaluate this particular JavaScript function in run time to generate an `ArrayList` data saved as another global variable with ID of ‘passpicture’. Another <callback> is also defined to call the function `arrangeChallenge()`. This function requires two ‘static’ input arguments, meaning that we can use pre-defined global variables as input arguments. As illustrated in Figure 16, ‘numberOfChallenges’ and ‘passpicture’ are the two input arguments for this function. The output of this function is a global variable with ID of ‘challenges’, which is saved as an `ArrayList` for later use.

```
<global_variable>
    <variable>
        <id>numOfChallenges</id>
        <type>Integer</type>
        <value ref="false">7</value>
    </variable>
</global_variable>
```

```
<callback type="js">
    <id>passpicture</id>
    <file>undercover.js</file>
    <function>generatePassPicture</function>
    <argument_list>
        <argument type="Integer" ref="false">28</argument>
        <argument type="Integer" ref="false">1</argument>
        <argument type="Integer" ref="false">5</argument>
    </argument_list>
    <data_structure>ArrayList</data_structure>
</callback>
```

```
    <callback type="js">
        <id>challenges</id>
        <file>undercover.js</file>
        <function>arrangeChallenge</function>
        <argument_list>
            <argument type="ArrayList" ref="true">
                <callback_link type="static" style="global">passpicture</callback_link>
            </argument>
            <argument type="Integer" ref="true">
                <global_var_link type="static" style="local">numOfChallenges</global_var_link>
            </argument>
        </argument_list>
        <data_structure>ArrayList</data_structure>
    </callback>
</global_callback>
```

Fig. 16. XML code for global variable initialization of the descriptive model

The *high-level UI description* and *high-level interaction description* are developed to complete sub-tasks 3 and 4. The output of completing objective 2 is the arranged seven challenge screens. For each challenge screen, the layout of the UI is converted into XML code (i.e., similar to the example showed in Figure 5 (a)).

<task names="t1"> element as illustrated in Figure 17 (a) calls the JavaScript function `getPassPictureIdx()` as shown in Figure 17 (b) from the algorithmic model. This function takes one challenge screen from the array-list variable ‘challenges’ and one ‘pass-picture’ from the array-list variable ‘passPictures’ to derive the position of the ‘pass-picture’, and save it as a variable with the ID of ‘passPicIdx’. This variable is later refereed in the <task name="t2"> element as shown in Figure 17 (a) to indicate which button needs to be pressed.

<task name=‘t1’> and <task name=‘t2’> are used together to define the *high-level interaction* (i.e., button pressing events). The <widget_group> ‘photo group’ and ‘button group’ represent the group of widgets to display images at public challenge panel and the group of buttons at the response panel of the system GUI, respectively.

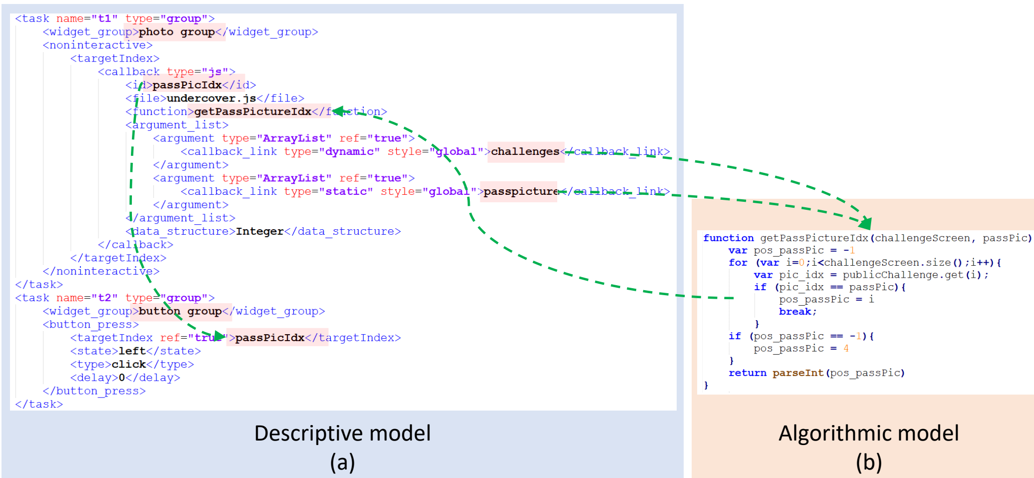


Fig. 17. Illustration of using a JavaScript function to facilitate describing the high-level interaction description

5 EVALUATION OF COGTOOL+

In this section, we present an evaluation of our implemented prototype of CogTool+ by applying it to model two real-world user-authentication modeling tasks – modeling 6-digit PIN entries and the graphical password authentication system Undercover [30] already mentioned before.

5.1 Modeling 6-digit PIN entries

PINs remain one of the most widely used user-authentication methods in everyday life, e.g., authentication on mobile devices and access control to online banking. Several types of inter-keystroke timing attacks make use of the leaked keystroke timing information to infer a user's PIN, which can be a serious threat to users relying on such PINs. For instance, Liu et al. [19] proposed a user-independent inter-keystroke timing attack on PINs that performed significantly better than random guessing attacks. The attack methodology relies on an inter-keystroke timing dictionary built from Fitts's Law, which relies on conducting real human user study to derive parameters of this model. In this subsection, we demonstrate that CogTool+ is cost-effective and accurate for modeling 6-digit PIN entries at a relative large scale.

5.1.1 Modeling PIN entries. 50 different 6-digit PINs were used in the real human user study conducted by Liu et al. [19]. Each PIN was entered using the number pad as illustrated in Figure 5 (b). Our aim here is to compare the inter-keystroke timing sequences of simulated data generated using CogTool+ with the real human user data.

PIN	$k_1 \rightarrow k_2$	$k_2 \rightarrow k_3$	$k_3 \rightarrow k_4$	$k_4 \rightarrow k_5$	$k_5 \rightarrow k_6$	$k_6 \rightarrow$ <Enter>
777777	202.2	204.0	207.9	204.1	212.8	320.2
530271	229.6	224.9	214.5	245.8	246.2	278.1
603294	241.2	227.4	203.4	239.8	233.1	292.2

Table 1. Examples of inter-keystroke timing sequences (in ms) for PIN entry tasks

As illustrated in Table 1, each row is the timing sequence of entering one PIN. For a 6-digit PIN, six timing intervals are recorded. For instance, $k_i \rightarrow k_j$ represents the time interval (in ms) between pressing the j-th digit key and pressing the i-th digit key, and $k_6 \rightarrow$ <Enter> is the time between pressing the <Enter> key and the last digit key. The process of modeling 50 6-digit PIN entries is similar to the examples showed in Section 3. There are three major steps:

(1) A simulation model similar to the example depicted in Figure 2 with static preference setting added. The `<trial></trial>` is set to be 50, and a `<callback>` function is used to link external data (i.e., 'PINs.csv' file that contains 50 PINs. More information about these PINs can be found in [19]). This data set is also made available to the descriptive model as a variable with the ID of 'externalPin'.
(2) The descriptive model as shown in Figure 5 (a) is used to describe the graphical representation of the UI (i.e., Figure 5 (b)) to the high-level description of UI as shown in Figure 5 (c) using XML.
(3) As demonstrated in Figure 6. The simulation model automatically parses one PIN to the descriptive model, where this PIN is stored as a `<global_variable` with id of 'password' as highlighted. Given this PIN, the descriptive model automatically generates a series of pressing button user interactions. The 'numberFrame' highlighted is defined as another `<global_variable>` in the descriptive model with its attribute 'type' of `<frame_setting>` set to be 'dynamic'. This can allow the model interpreter to automatically generate a low-level description of seven frames (see Figure 5 (d)), where each frame corresponds to either pressing a digit or pressing the <Enter> key. The time differences between seven frames forms the inter-keystroke timing sequences.

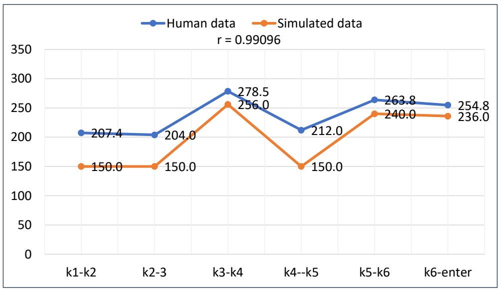
Finally, the above three-step process is automatically executed until all 50 PIN entry tasks are modeled (i.e., `<trial>50</tiral>`). As there is no need to have a mixed probability model for this task, the mixed model only contains one meta model with weight of 1.

5.1.2 Results. In the real human user study in [19], each participant was asked to enter a random 6-digit PIN five times in a training session to familiarise with the given task. These participants could be considered as skilled users, which made their performance data comparable with the simulated data produced using CogTool+. Then, each participant was instructed to enter each PIN 15 times.

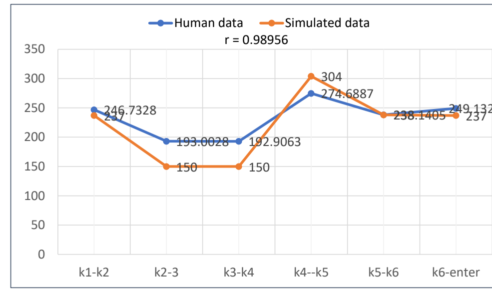
In this evaluation experiment, we used the mean value of inter-keystroke timing sequences from the user study to make a comparison with the simulated data using CogTool+. Figure 18 illustrates the comparison between the human data and the simulated data for a number of selected PINs. As shown in Figure 18 (a), (b), (c), and (d), the correlation coefficients for PIN 000533, PIN 100086, PIN 990872, and PIN 443333 are 0.99096, 0.989956, 0.94458, and 0.97311, respectively. In addition, the mean and standard deviation of correlation coefficient for all 50 PINs are 0.807 and 0.233, suggesting a strong association between the human timing data and the simulated timing data for all 50 given 6-digit PINs.

5.1.3 Comparison of efforts needed to model 6-digit PIN entry tasks: CogTool+ vs. CogTool. Here we present more details to elaborate on the efforts needed for this modeling task using CogTool+, compared with the efforts needed to model the same task using CogTool. Figure 19 shows the comparison, where the light red color cells and red arrows represent the manual work needed, and the light green cells and green arrows represent the automated process.

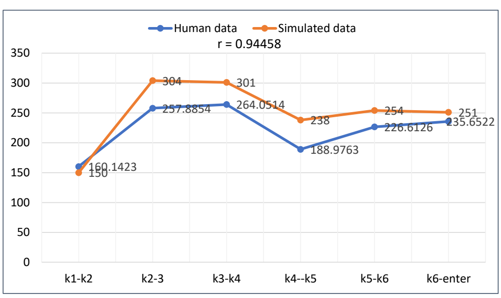
For the preparation of this modeling task, 50 PINs used in this study were provided externally [19]. We stored them in the CSV format. We manually developed three models for CogTool+: a meta model, a simulation model, and a mixed model. Using CogTool, the user would need to create one CogTool project with 50 CogTool tasks to model 50 PIN entry tasks manually. Each CogTool task consists of one UI design and one demonstration script. As 50 CogTool tasks share the same UI



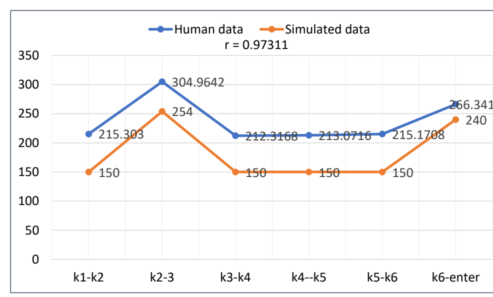
(a) Inter-keystroke timing data for the PIN '000533'.



(b) Inter-keystroke timing data for the PIN '100086'.



(c) Inter-keystroke timing data for the PIN '990872'.



(d) Inter-keystroke timing data for the PIN '443333'.

Fig. 18. Comparison of inter-keystroke timing data between human user and simulation, where y-axis is the performance time in milliseconds, and x-axis is the inter-keystroke time interval, r represents the correlation coefficient between human data and simulated data

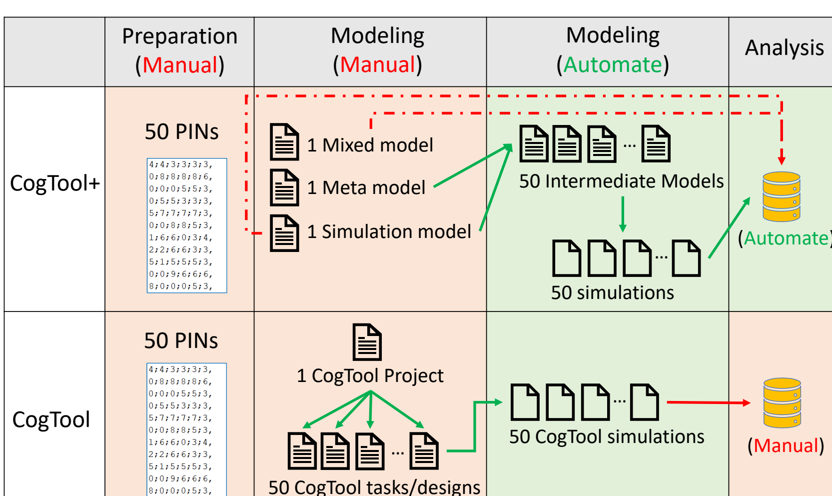


Fig. 19. Comparison of efforts needed to model 6-digit PIN entry tasks using CogTool+ vs. CogTool

669 design, the user would just need to copy and paste the UI design. Although only one CogTool frame

is enough to model the UI for the PIN entry task, the reason to have a number of CogTool frames for each UI design is to accurately measure the inter-key stroke timing difference to compare with the real human user study. The user needs to make seven clicks on each CogTool frame for all CogTool frames to generate one demonstration script. In total, that would be 350 clicks to produce all demonstration scripts. Then the CogTool can utilize the back-end ACT-R architecture to compile and run the simulation automatically.

Both CogTool+ and CogTool can automatically generate 50 simulations. The model interpreter of CogTool+ produces 50 intermediate models, which are equivalent to 50 CogTool tasks. As we can define simulation parameters in the simulation model and parameters for probabilistic modeling in the mixed model, CogTool+ can use these parameters to handle the data collection and analysis automatically. To do the same task using CogTool would require the user to collect all simulation results first, and then conduct the analysis manually using other external software tools such as Microsoft Excel etc.

Compared with CogTool, the place where CogTool+ can make a significant difference is the use of the meta model to reduce the workload needed.

For this study, there is no need to design an algorithmic model as a part of the meta model, thereby the meta model only contains a descriptive model. As illustrated in Figure 3, each descriptive model has the same structure that includes three parts: global variable initialization, high-level UI description, and high-level interaction description.

- **Global variable initialization**: as demonstrated in Figure 4, only a simple syntax is needed to define a global variable, which interfaces with the simulation model to read a PIN.
- **High-level UI description**: the development of this part starts with the similar approach that CogTool has to convert the PIN pad UI to one frame written in XML format. Using CogTool+, only one frame is need to be defined. With the 'dynamic' frame setting, the model interpreter can use the global variable to automatically derive a number of frames with associated transitions between frames in run-time. With CogTool, although it is not too time consuming to do the same task using 'copy and paste', it still requires a significant amount of time to repeat the action 50 times.
- **High-level interaction description:** the development of this part only requires a user to define coarse user interactions. As mentioned in Section 3.2, the model interpreter can automatically generate a number of button pressing events and derive the transition from an action event to next frame if needed. As mentioned earlier in this section, doing the same task for all 50 PINs using CogTool would require the user to manually complete 350 clicks. In addition, the user needs to constantly pay attention to model the correct PIN, which can increase the mental workload that would potentially slow down the modeling process.

5.2 Modeling Undercover

The details of modeling a simplified version of the 'Undercover' system have been presented in Section 4. In this part of the paper, we present more details on modeling the full 'Undercover' system. In particular, we demonstrate the usefulness of CogTool+ in modeling more complex and dynamic parts of the 'Undercover' system. We also present the simulation results in comparison with the results of the real human performance data reported in [26].

The brief description of the Undercover system has been introduced in Section. 4. There are several reasons why we chose Undercover to evaluate CogTool+. Undercover is a relative complex system that involves different cognitive tasks, and it has a combination of static UIs and dynamic user interactions. It is very difficult to model such a system using CogTool. We aim to prove that the advantage of achieving parameterization and automation in CogTool+ can allow cyber security researchers to model complicated systems such as the Undercover system. We also aim to look at both the estimated prediction using CogTool+ and the real human performance data from a lab-based user study [26] to evaluate CogTool+.

5.2.1 Modeling Undercover using CogTool+. To make an adequate comparison with the findings reported by Perković [26], we used CogTool+ to model their implementation of Undercover (see Figure 12 (c)). The main finding from their study is the non-uniform human behaviors which indicate potential security problems in the use of Undercover. We aimed to find out if we can automatically detect such insecure behaviors using CogTool+.

Using the same approach as the one presented in Section 4, we need to have a comprehensive understanding of the work flow of using the Undercover system.

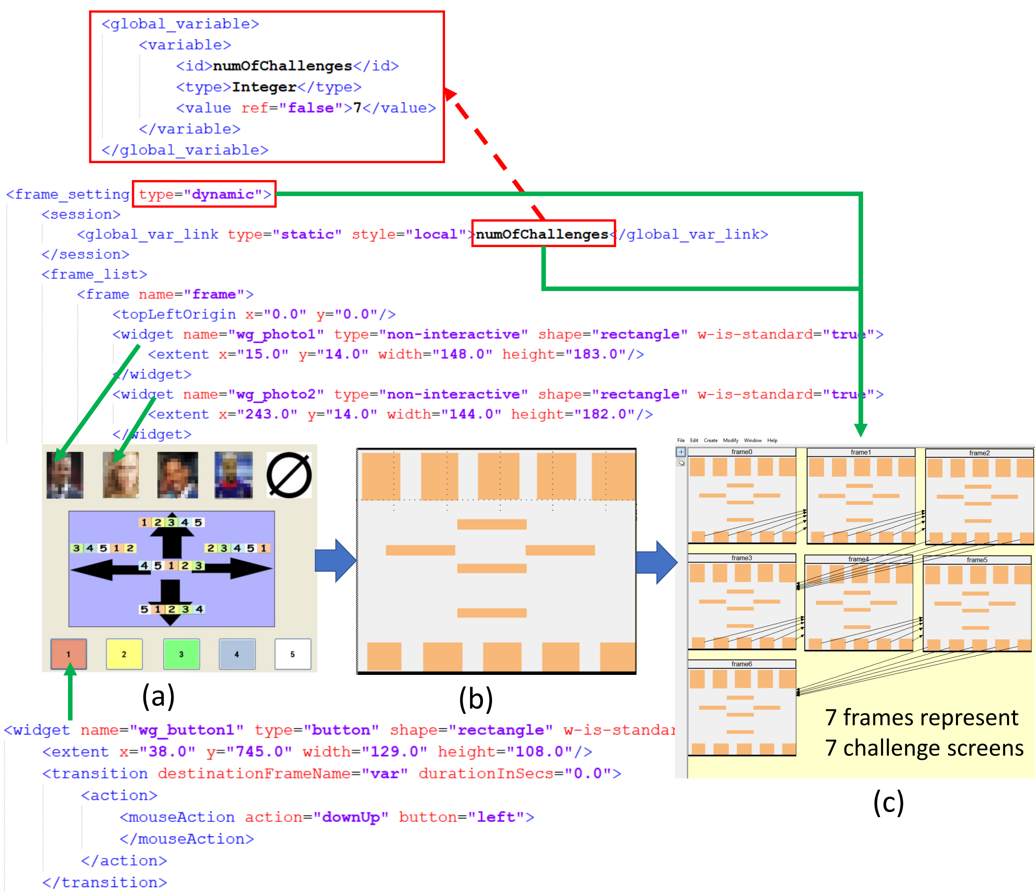


Fig. 20. Modeling the creation of seven challenge screens: (a) the Undercover UI; (b) Visualization of the Undercover UI model for one challenge screen; (c) Visualization of the Undercover UI models for seven challenge screens

Each user needs to select five 'pass-pictures', and complete seven challenge screens. Each challenge screen has the same graphical representations as shown in Figure 12 (c), and we considered this as one static element to be modeled using a descriptive model. Figures 15 and 16 in Section 4.4 show the modeling process of selecting five 'pass-pictures' and for arranging 7 challenge screens, respectively. Here, Figure 20 illustrates more details and the visual representation in addition to the pedagogical example presented earlier. Figure 20 (a) represents the Undercover UI. Then we converted it into the high-level description of UI as illustrated in Figure 20 (b) using XML.

Then we defined a global variable in the descriptive model (i.e., <global_variable>, as highlighted in the red rectangle, which indicates the number of challenge screens), and set the attribute 'type' of <frame_setting> to be 'dynamic'. The model interpreter can interpret this, and automatically produce a low-level description of the seven challenge screens (see Figure 20 (c)).

Similar to the demonstration in Figure 15, there is a number of sub-tasks requiring dynamic inputs/outputs:

- Sub-task 1 (see 'Sub-task 1' in Section 4.1)
- Sub-task 2 (see 'Sub-task 2' in Section 4.1)
- Sub-task 3 (see 'Sub-task 3' in Section 4.1)
- Sub-task 4: Random hidden challenge for each challenge screen: a random hidden challenge needs to be generated (i.e., one value from 'Up', 'Down', 'Left', 'Right', 'Center').
- Sub-task 5: Public response for each challenge screen: The hidden challenge is known from Sub-task 4, then we can derive the specific layout corresponding to the generated hidden challenge. Also, the position of 'pass-picture' is known from Sub-task 3, then the correct button to press can be derived.

Furthermore, each challenge screen contains the same challenge tasks with different content repeated seven times, thus suggesting another dimension of the dynamic nature of the modeling task. We developed an algorithmic model consisting of a few JavaScript functions to handle these dynamic elements.

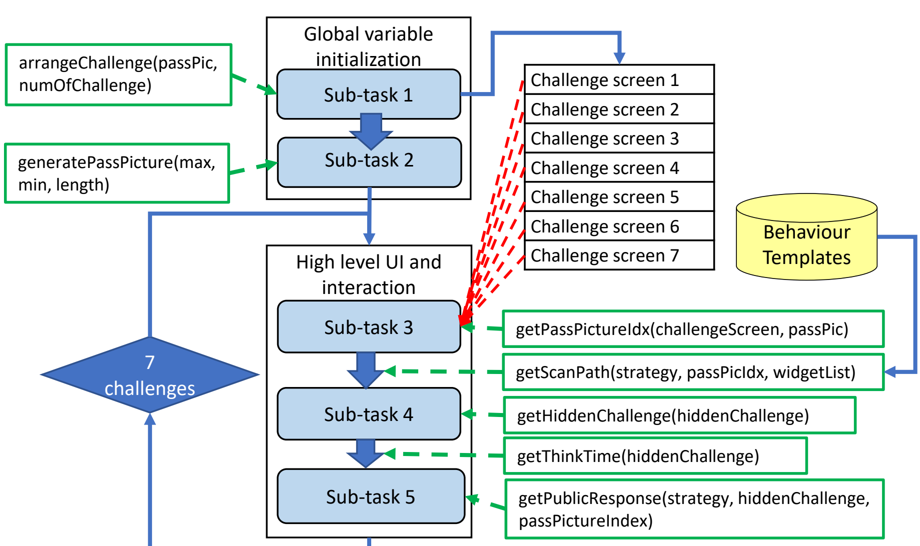


Fig. 21. The flowchart of modeling the Undercover user authentication process.

As demonstrated in Figure 21, contents in the green rectangles are the JavaScript functions defined in the algorithmic function. Apart from the functions (i.e., `generatePassPicture()`, `arrangeChallenge()`, and `getPassPictureIdx()`) already mentioned in Section 4.4, function `getScanPath()` is created to model the visual-search process of finding the 'pass-picture' among an array of pictures. A previous study [44] revealed that there are several visual scan paths for such task. In that study, most of the participants adopted a search strategy of center-left-right (i.e., start the search process from the middle, and move left and right), and a minority of participants simply searched from left to right. Different visual search strategies will result in different visual search times, `getScanPath()` can be considered as an example of updating the 'Thinking' time dynamically. As illustrated in Figure 21, this function acts as the interface to add such behavioral template databases to the algorithmic model to better model the cognitive task.

In addition, the function `getHiddenChallenge()` generates a random hidden challenge index. There are five values of hidden challenge, and we used 1 to 5 to represent each value. An index to represent the hidden challenge is randomly generated for *Sub-task 4*. Lastly, *Sub-task 5* utilizes function `getPublicResponse()` to take the 'pass-picture' position index and the hidden challenge index to derive the public challenge response (i.e., which button needs to be pressed at the end of each challenge screen).

The effort to derive the public response needs to be taken into consideration in the modeling process as each hidden challenge index corresponds to a different hidden challenge button layout panel as shown in Figure 12 (b), which could result in different reaction times. The button layout for hidden challenge 'Up' has the same order of button (i.e., 1, 2, 3, 4, 5) as the response button panel. We could assume that there is no or minimum effort needed to identify the public challenge response in this case. However, button layouts corresponding to other hidden challenges have completely different order of buttons (i.e., '3, 4, 5, 1, 2' for hidden challenge 'Left', '4, 5, 1, 2, 3' for hidden challenge 'Center', '2, 3, 4, 5, 1' for hidden challenge 'Right', and '5, 1, 2, 3, 4' for hidden challenge 'Down'). We could assume that some effort is needed to derive the public response for these cases.

Except for hidden challenge 'Up', we treated other cases as a single visual target search problem. The relationship between the reaction time and the windows size (i.e., the number of images) is believed to be linear [42, 43]. The reaction time can be predicted using $t = 0.583 + 0.0529 \cdot w$ [43], where w is the number of images. We incorporated this information in a JavaScript function `getThinkTime()` to dynamically derive the extra time incurred between *Sub-task 4* and *Sub-task 5* given a hidden challenge. Similar to the function `getScanPath()`, `getThinkTime()` shows another example of using an algorithmic model to dynamically update the 'Thinking' time.

In addition, participants have the tendency to visually confirm the position of the 'pass-picture' before pressing the button. To add this finding to the model, we added another atomic action 'look-at' towards the position of the 'pass-picture' before pressing the correct button for *Sub-task 5*.

Compared with the design of a meta model for the Undercover system, the design of a simulation model and a mixed model is simpler and similar to the examples demonstrated in Section 4. We designed a number of individual meta models named CLR-Only (center-left-right without confirmation process), LR-Only (left-right without confirmation process), CLR-Confirm (center-left-right with confirmation process), and LR-Confirm (left-right with confirmation process) to represent the different behavior patterns. Then we gave different weights to the different meta models. For each meta model, an accompanying simulation model was designed to produce 150 predictions. In total, this mixed model generated 150×4=600 predictions, whereby each prediction took approximately 1 second to be processed. As all meta models for this study contained the same algorithmic model, and shared the same simulation setting, only one simulation model and one algorithmic model were needed.

The behavior patterns and weight used in the modeling process were obtained from our previous research [44]). These behavior patterns can be written as behavioral templates database for other users to re-use. By doing this, we wanted to demonstrate the fast prototyping and get some insights into how CogTool+ works, which can be simplified as: 1) building a simplified GUI even on a piece of paper; 2) conducting some quick experiments to extract behavior data; 3) using such external data to drive the modeling process. This simplified process could be quicker and more accurate than applying general rules/models.

5.2.2 Results and Visualization. Figure 22 shows a graphical representation of the visualization GUI. Each rectangle is a node in the mixed-model tree. Four nodes labeled with 'CLR-Only', 'CLR-Confirm', 'LR-Confirm', and 'LR-Only' are representations of the meta models defined earlier. Node 'CLR' represents a mixed-probabilistic model (a.k.a, CLR model) consisting of a 'CLR-Only' meta model and a 'CLR-Confirm' meta model, and node 'LR' represents a mixed-probabilistic model (a.k.a, LR model) consisting of a 'LR-Only' meta model and a 'LR-Confirm' meta model. Node 'Visual Search' is the overall mixed-probabilistic model for this modeling task. To view the relationship between the different nodes, a user needs to click on one node. If there are any other nodes related to the selected node, all of them will be highlighted with a yellow arrow connecting associated nodes as shown in Figure 22.

In addition, user-defined visualization parameters determine the arrangements of the rounded corner rectangles in the graph. Each rounded corner rectangle is a representation of a type of figure that the user wants to see. For this modeling task, we were more interested in the predicted average response time for each hidden challenge value. As revealed by the previous lab-based user study [26], real human users responded to hidden challenge 'Up' the fastest. Our model produced similar results (see Figure 23 (a) and (b) for our results and results from the user study). To be noted that Figure 23 (a) is the screenshot of the actual figure produced by CogTool+ visualization module, and Fig 23 (b) is the actual figure from the paper [26].

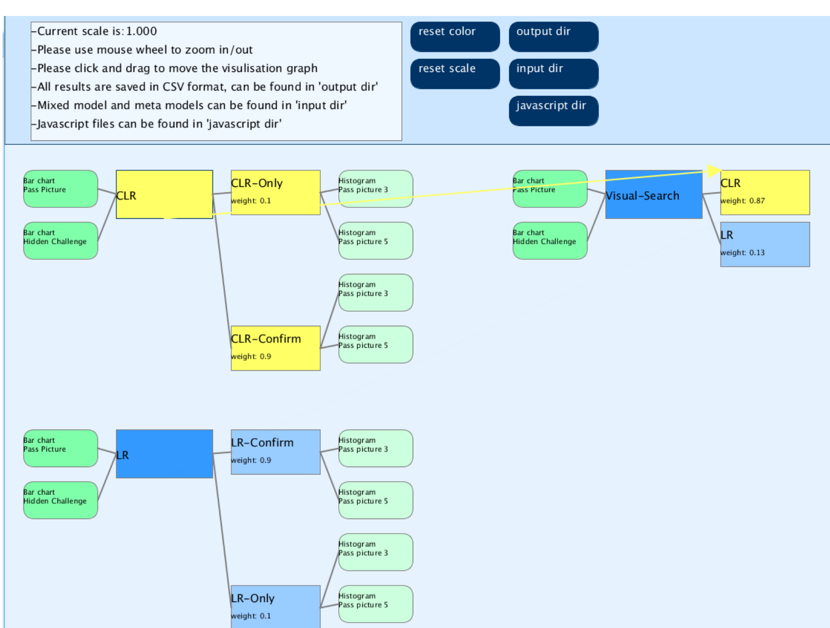
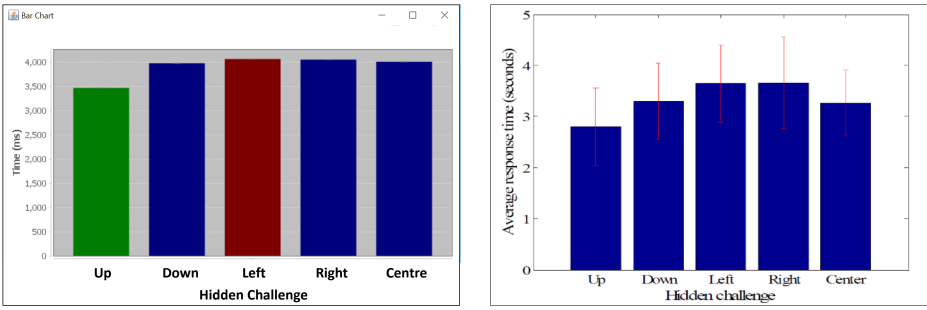


Fig. 22. The visualization of the modeling task on Undercover.



(a) (b)

Fig. 23. (a) Bar chart produced by CogTool+ showing the predicted average response time per hidden challenge c_h using CogTool+; (b) Average response time per hidden challenge c_h using real human data (the error bars correspond to standard deviation) [26]

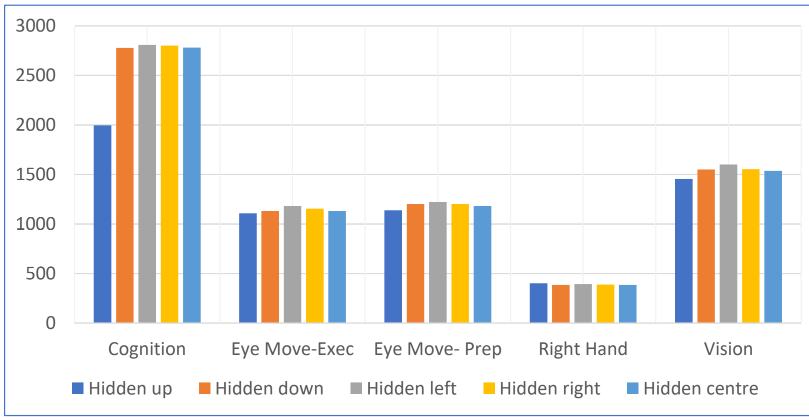
Since CogTool+ predicts performance of skilled users, and data from [26] were obtained from relatively unskilled individuals, we did not expect that our results could match the results reported in [26] exactly. In addition, there are differences between our experiment and the study in [26]. For instance, participants in [26] were separated into two groups, one was told to use the mouse to interact with Undercover, and another group was informed to use keyboard to interact with Undercover. Some degree of discrepancy in the results was therefore anticipated. The main finding from [26] was that security issues can be discovered by investigating human behaviors/performance patterns, in particular the non-uniform time distribution of response time. In our modeling attempt, we were initially more interested in investigating whether CogTool+ could discover such behavior patterns rather than establishing a direct comparison to the results by [26]. We did identify similar, non-uniform patterns in the results produced by CogTool+ (i.e., for both hidden challenge and pass image reaction times, we identified the slowest timing). These results suggest that the non-uniform patterns could be predicted even without taking into account the participants' skill level, which could explain the outstanding discrepancy between the predicted vs. real user data. One unanswered question in the original study [26] is to find the cause of these nonuniform behaviors, and there was no conclusive answer. Thanks to the CogTool's support to extract operation information of the ACT-R model, CogTool+ inherits such features and could help us further investigate this by looking at detailed timing data for each operator.

As shown in Figure 24 (a) and (b) [6], the 'Cognition' operator [7] required more time for each task compared with other operators for both CLR and LR models, meaning that the 'Cognition' operator could be the major contributor to the shortest reaction time for the 'Hidden Up' challenge regardless of the visual search strategy. In other words, 'Hidden Up' required 'Cognition' less than other challenges did.

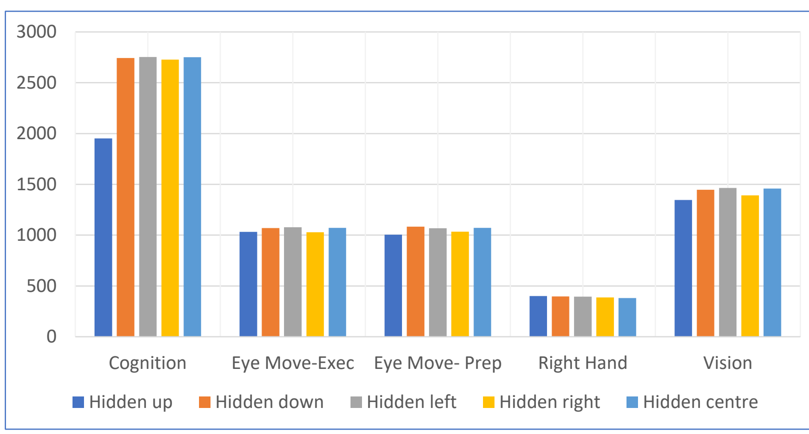
5.2.3 Comparison of the efforts needed to model Undercover: CogTool+ vs. CogTool. Here we explain in more detail the efforts needed to model Undercover system, compared with the efforts needed using CogTool to complete the same task. As illustrated in Figure 25, light red cells and red arrows

[6] As there are parallel operations and overlapped timing, the sum of these operations' time does not equal to the overall response time reported in other figures

[7] The Cognition operator includes the thoughts the model has (i.e., 'Think' steps) and other types of cognitive operators that initiate motor movements and visual attention shifts. (From CogTool user guide, avaiable at https://github.com/cogtool/documentation/tree/master/end-user/user-guide)



(a) Detailed timing data per hidden challenge c_h for CLR model.



(b) Detailed timing data per hidden challenge c_h for LR model.

Fig. 24. Operation timing data of the ACT-R model for different CogTool+ models.

represent the need of manual work, light green cells and green arrows represent the automated process.

Before building the model using either CogTool+ or CogTool, there is the need to understand the Undercover system thoroughly as we mentioned in Section 5.2.1, especially for its dynamic elements.

As shown in Figure 25, using CogTool+, four individual meta models (CLR only, CLR confirm, LR only, and LR confirm), one simulation model, and one mixed model are needed to complete 600 modeling tasks. Each meta model consists of a descriptive model and an algorithmic model. As all meta models use the same algorithmic model, there are four (descriptive models) + one (algorithmic model) + one (simulation model) + one (mixed model) = seven individual models need to be developed in XML format manually. It is worth noting that we design the algorithmic model to generate the dynamic data in run-time automatically.

For CogTool, the user would need to manually develop one CogTool project with 150 CogTool tasks for CLR-only, 150 CogTool tasks for CLR-confirm, 150 CogTool tasks for LR-only, and 150

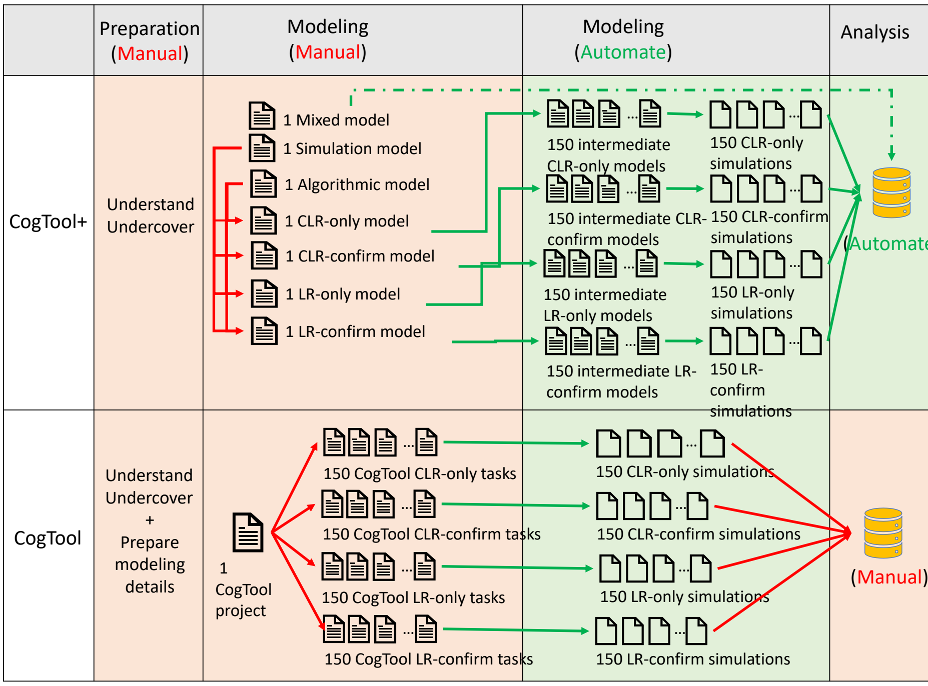


Fig. 25. Comparison of efforts needed to model Undercover using CogTool+ vs. CogTool

CogTool tasks for LR-confirm (i.e., 600 CogTool tasks in total). It should be noted that each single CogTool task needs to consider the dynamic data, and the standard version of CogTool does not support the automatic generation and integration of such data in run-time. This would require the user to prepare dynamic data for 600 CogTool tasks manually in advance. It would require the user to use external software tools to generate such data fairly to avoid any unnecessary bias. In addition, it can be very time-consuming to convert and integrate such dynamic data using CogTool at large scale.

It should be noted that the model interpreter of CogTool+ can input the seven individual models to automatically output 600 intermediate models, which are equivalent to 600 CogTool projects/-tasks. As depicted in Figure 25, both CogTool+ and CogTool can automatically finish 150×4=600 simulations.

The parameters defined in the mixed model and simulation parameters can be used to deal with the data collection and data analysis automatically using CogTool+. By contrast, CogTool would require the user to do the same task manually.

To support modeling the Undercover system using CogTool+, we spent most of our efforts to design the algorithmic model and meta models following the approach showed in Section 4 and Section 5.2.1.

Algorithmic model. The algorithmic model written in JavaScript has seven functions (i.e., see green highlights in Figure 21). It requires a beginner level of programming knowledge and thorough understanding of the Undercover system to handcraft these functions. We spent more time understanding the Undercover system and converting the authentication task into a number of sub-tasks, compared with the time needed to produce the JavaScript functions. The programming part only requires knowledge to use existing random functions and some basics such as logic, conditional, and arithmetic operations.

We would like to emphasize that it would require a similar amount of effort to dissect the Undercover system and convert it to computational models regardless of the modeling software tools used. In other words, to model the algorithmic part of the Undercover system using CogTool would require the same or even more effort.

Descriptive model. Refer to the Figure 3, each descriptive model has the same structure that includes the global variable initialization, high-level UI description, and high-level interaction description. In this experiment, all descriptive models including CLR-only, CLR-confirm, LR-only, and LR-confirm share the same code-base for global variable initialization and high-level UI description. There is only a minor difference of high-level interaction description among these four descriptive models.

- **Global variable initialization**: As illustrated in Figure 16 and explained in Section 5.2.1, simple syntax is used to define both <global_variable> and <global_callback>.
- **High-level UI description:** Similar to the effort needed for modeling PIN entry tasks, the high-level UI description starts with converting one UI layout to one frame written in XML format. The dynamic frame setting allows the model interpreter to utilize the global variables and call the JavaScript functions in run-time to generate seven frames with corresponding transitions between frames automatically. This can be done using CogTool, but it requires lots of manual work to complete the task frame by frame for creating the required 600 CogTool projects.
- **High-level interaction description:** For all descriptive models, we need to define coarse user interactions. The minor difference between different descriptive models depends on the visual-search strategy to be modeled. Different parameters can be used with function `getScanPath()` to assign different visual search strategy dynamically. 'CLR confirm' and 'LR confirm' models require one additional interaction step to model the confirmation behavior compared with the 'CLR only' and 'LR only' models. Figure 26 shows one example of converting high-level interaction description of the 'CLR only' model to its low-level interaction description. The low-level interaction description automatically generated using CogTool+ is equivalent to scripts manually generated using CogTool.
  The coarse user interaction includes four steps: 1) find a picture, which consists of deriving the pass picture position, and selecting the visual search strategy; 2) receive the random hidden challenge; 3) derive the hidden response; 4) derive the public response and action. As illustrated in Figure 26, CogTool+ can automatically generate detailed low-level interaction descriptions for seven frames, where the red arrows also represent the correct transitions between frames.
  To do the same for a single frame using CogTool will require a user to manually go through interactions step by step by clicking on the CogTool frame via the CogTool Design interface. In the same time, the user needs to pay attention to accurately integrate the dynamic data into the interaction steps script.

In summary, there are several advantages of using CogTool+ to model Undercover:

- The first one is the modeling part. Undercover has its algorithmic elements including the selection of pass pictures from an image pool, image arrangement for the public challenge interface, and generation of random hidden challenges, that are difficult to capture and model using existing software tools.

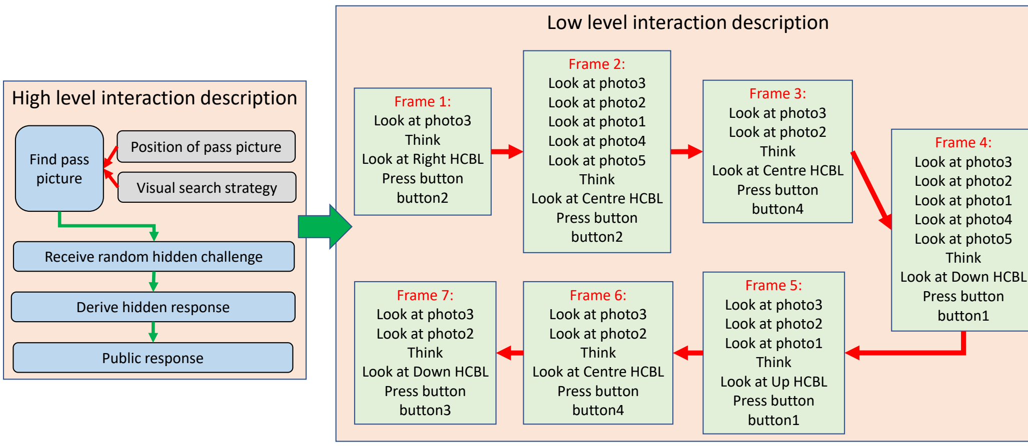


Fig. 26. Example of using high-level interaction description of the 'CLR only' model to generate low-level interaction description (equivalent to CogTool interaction script). HCBL stands for hidden challenge button layout (i.e., Figure 12 (b2))

- The second one is to allow external data-driven modeling, whereby scholars can use empirically determined patterns extracted from eye-tracking data to interface with the modeling process. In addition, such external data can be generalized as behavioral patterns/templates to be used in other modeling tasks.
- The third one is to conduct relatively large modeling tasks (600 simulations) with significant less effort than existing tools. It should be noted that each simulation has its own parameters including the pass picture, the public challenges, and the hidden challenges, that are automatically generated using the proposed algorithmic model.
- The detection of insecurity behaviors is reflected by looking at the overall human performance prediction to observe any anomaly such as non-uniform behavior data. Currently the offline analyser only supports basic functionality, and the auto-detection will depend on more advanced analyses such as statistical analyses to offer users more concrete information on the detection of insecure behaviors. We plan to address this aspect in our future work.

5.3 Additional remarks

We have used CogTool+ to model two tasks, and we showed that our approach can produce simulated data that are similar to the findings of real human-user studies. In terms of the effort needed to model these tasks using CogTool+, our approach is considerably more streamlined compared to the real human-user research, which is often a time-consuming and financially expensive process that involves ethics applications, participant recruitment, experiment design and setup, and data collection. Furthermore, our approach could be considered as an addition or a supplementary contribution to the CogTool research community to offer alternative ways for large-scale human performance modeling.

In this paper, we have demonstrated that we can use CogTool+ to model the 'Undercover' system and 6-digit PIN entry tasks. The reason for selecting these two examples is not that they are easy to model using CogTool+. They were selected because: 1) we would like to demonstrate how to use CogTool+ to model dynamic elements. Although our given examples show some limited number of dynamic elements, CogTool+ can be easily extended to support more dynamic elements by adding new algorithmic models. 2) One of the major challenges for cyber security researchers is to model highly dynamic UIs of cyber security system using existing cognitive modeling software such as CogTool. This actually spurred the development of CogTool+.

We developed and implemented CogTool+ by adopting and extending CogTool with additional models and interfaces. It inherits CogTool's full capability to model many different UIs as proved by its wide use in the HCI community. We believe CogTool+ can only enhance the modeling capabilities of CogTool rather than limiting it, and we are confident that CogTool+ can be used to model different UIs in many other application areas. In our future work, we will investigate how to use CogTool+ to model more complicated UIs and conduct large-scale simulations. In addition, a similar approach to extending CogTool can be applied to other existing modeling tools to extend their capabilities but still maintain their valuable features and benefits. Two examples are the support of parallel modeling and capability to produce results in distribution format to represent the individual differences from SANLab-CM, and the support of modeling multi-tasking and working memory from Cogulator. Last but not least, we plan to work on these extensions to create a larger system that will allow different tools and models to be incorporated and work together in a single software framework.

6 LIMITATIONS AND FUTURE WORK

As discussed in the previous section, the use of algorithmic and descriptive models facilitates the parameterization and automation of the modeling process. JavaScript is the main way to develop an algorithmic model, which may require the user to have a certain level of programming knowledge. This would potentially affect the usability and bring extra burden to the user when using this system, and therefore we regard this as one of its possible limitations. To overcome this, we improved the design to allow the user to use external files in CSV format to achieve the same objective. However, this cannot fully afford the flexibility and dynamic nature of using JavaScript. To address this potential issue, we are planing to develop a set of JavaScript utility modules that would be frequently used in a modeling process to assist the end user. Furthermore, as mentioned in the previous section, JavaScript behavioral template databases have been added to the algorithmic model as external data to assist the modeling process. In addition, we can build behavioral template databases implemented in JavaScript as part of our future work.

The original CogTool supports modeling through the classical window, icons, menus, pointer (WIMP) user interface. The ultimate goal is to make CogTool+ fully compatible with CogTool. We prioritized its development to ensure that the software could model basic interaction tasks such as 'pressing button', using mouse, or touch screen. There is a number of graphical elements such as 'context menu', 'web link' and 'pull down list' that CogTool can model, but the current version of CogTool+ is not supporting. However, this system framework has been developed to be flexible and re-configurable. We are planning to add more software modules to fully support modeling WIMP (Windows, Icons, Menus, Pointer) user interface in our future work.

In addition, the current implementation of CogTool+ only features an easy-to-use GUI for data analysis and visualization. In future work, we would like to incorporate and extend CogTool GUI for modeling, design and develop UI/UX designer facing UI for XML editing.

In the present paper, we provided evidence that CogTool+ can be used to model cognitive tasks at large scale. Although we have conducted more evaluations of the system internally within our research centre, the proposed system CogTool+ has not yet been tested externally. We will make this software openly accessible and provide a platform so that other scholars and users can provide their feedback. We would like to see more researchers and practitioners using CogTool+ to test additional systems for a wider range of topics. We consider this as the first step to move forward, and possibly contribute to the progress of CogTool.

It is worth mentioning again that our current implementation CogTool+ inherits CogTool's limitations on what UI elements it can support, and the limitation of using KLM as the underlying cognitive model. However, CogTool+ has been developed and implemented in a way that has the flexibility to add software modules/components and external data sets. Based on this design principle, we are investigating and extending our research to develop a more general framework with new software tools that can go beyond CogTool+ by adding/integrating other cognitive models, UI modeling components and software modules.

7 CONCLUSION

In this paper, we propose a new cognitive modeling software framework called CogTool+ that extends the widely used open-source software tool CogTool to enhance its support on modeling large-scale human performance tasks. The implemented prototype CogTool+ presents possible solutions to address these concerns with capabilities to support parameterization and automated model generation. Human- and machine-readable language designed in XML format is used to facilitate the design of the mixed model and the meta model, which allow users to model dynamic interaction tasks as well as processing and generating large number of cognitive models automatically in a programmatic manner.

We evaluated CogTool+ by modelling 6-digit PIN entry tasks, and reproduce fine-grained inter-keystroke data similar to real human data obtained from a lab-based user study [19]. In addition, we took a relative complex user-authentication system, Undercover [30], for evaluation. The results revealed that we can use CogTool+ to conduct large-scale experiments and reproduce some non-uniform human behavior patterns which have been identified in a lab-based user study [26].

ACKNOWLEDGMENTS

This work was supported by the UK part of a joint Singapore-UK research project "COMMANDO-HUMANS: COMputational modeling and Automatic Non-intrusive Detection Of HUMan behAviour based iNSecurity", funded by the Engineering and Physical Sciences Research Council (EPSRC) under grant number EP/N020111/1.

REFERENCES

[1] ACT-R Research Group, Department of Psychology, Carnegie Mellon University. [n.d.]. ACT-R. Website, http://act-r.psy.cmu.edu/.

[2] John R. Anderson. 2007. *How Can the Human Mind Occur in the Physical Universe?* Oxford University Press.

[3] John R. Anderson, Daniel Bothell, Michael D. Byrne, Scott Douglass, Christian Lebiere, and Yulin Qin. 2004. An Integrated Theory of the Mind. *Psychological Review* 111, 4 (2004), 1036–1060. https://doi.org/10.1037/0033-295X.111.4.1036

[4] Rachel Bellamy, Bonnie John, John Richards, and John Thomas. 2010. Using CogTool to Model Programming Tasks. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools.* ACM, Article 1. https://doi.org/10.1145/1937117.1937118

[5] Rachel Bellamy, Bonnie E. John, and Sandra L. Kogan. 2011. Deploying CogTool: integrating quantitative usability assessment into real-world software development. In *Proceedings of 2011 33rd International Conference on Software Engineering*. IEEE, 691–700. https://doi.org/10.1145/1985793.1985890

[6] Michael D. Byrne. 2001. ACT-R/PM and menu selection: Applying a cognitive architecture to HCI. *International Journal of Human-Computer Studies* 55, 1 (2001), 41–84. https://doi.org/10.1006/ijhc.2001.0469

[7] Stuart K. Card, Thomas P. Moran, , and Allen Newell. 1980. The Keystroke-level Model for User Performance Time with Interactive Systems. *Commun. ACM* 23, 7 (1980), 396–410. https://doi.org/10.1145/358886.358895

[8] Stuart K. Card, Allen Newell, and Thomas P. Moran. 1983. *The Psychology of Human-Computer Interaction.* L. Erlbaum Associates Inc.

[9] Gray Wayne D., John Bonnie E., and Atwood Michael E. 1993. Project Ernestine: Validating a GOMS Analysis for Predicting and Explaining Real-World Task Performance. *Human Computer Interaction* 8, 3 (1993), 237–309. https://doi.org/10.1207/s15327051hci0803_3

[10] Alexander De Luca, Katja Hertzschuch, and Heinrich Hussmann. 2010. ColorPIN: Securing PIN Entry Through Indirect Input. In *Proceedings of the 2010 SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1103–1106. https://doi.org/10.1145/1753326.1753490

[11] Sebastian Feuerstack and Bertram Wortelen. 2015. Revealing Differences in Designers' and Users' Perspectives: A Tool-Supported Process for Visual Attention Prediction for Designing HMIs for Maritime Monitoring Tasks. In *Human-Computer Interaction – INTERACT 2015: 15th IFIP TC 13 International Conference, Bamberg, Germany, September 14-18, 2015, Proceedings, Part IV (Lecture Notes in Computer Science, Vol. 9299)*. Springer, 105–122. https://doi.org/10.1007/978-3-319-22723-8_9

[12] Paul M. Fitts. 1954. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology* 47, 6 (1954), 381–391. https://doi.org/10.1037/h0055392

[13] Daniel Gartenberg, Ross Thornton, Mortazavi Masood, Dustin Pfannenstiel, Daniel Taylor, and Raja Parasuraman. 2013. Collecting health-related data on the smart phone: mental models, cost of collection, and perceived benefit of feedback. *Personal and Ubiquitous Computing* 17, 3 (2013), 561–570. https://doi.org/10.1007/s00779-012-0508-3

[14] Bonnie E. John. [n.d.]. CogTool | Cognitive Crash Dummies: Predictive human performance modeling for UI design. Website, https://cogtool.wordpress.com/.

[15] Bonnie E. John and David E. Kieras. 1996. The GOMS Family of User Interface Analysis Techniques: Comparison and Contrast. *ACM Transactions on Computer-Human Interaction* 3, 4 (1996), 320—-351. https://doi.org/10.1145/235833.236054

[16] Bonnie E. John, Konstantine Prevas, Dario D. Salvucci, and Ken Koedinger. 2004. Predictive Human Performance Modeling Made Easy. In *Proceedings of the 2004 SIGCHI Conference on Human Factors in Computing Systems*. ACM, 455—-462. https://doi.org/10.1145/985692.985750

[17] Siwan Kim, Hyunyi Yi, and Jyun Yi. 2014. FakePIN: Dummy Key Based Mobile User Authentication Scheme. In *Ubiquitous Information Technologies and Applications: CUTE 2013 (Lecture Notes in Electrical Engineering, Vol. 280)*. Springer, 157–164. https://doi.org/10.1007/978-3-642-41671-2_21

[18] John E. Laird. 2012. *The Soar Cognitive Architecture*. MIT Press.

[19] Ximing Liu, Yingjiu Li, Robert H. Deng, Bing Chang, and Shujun Li. 2019. When Human cognitive modeling meets PINs: User-independent inter-keystroke timing attacks. *Computers & Security* 80 (2019), 90–107. https://doi.org/10.1016/j.cose.2018.09.003

[20] Lu Luo and Bonnie E. John. 2005. Predicting Task Execution Time on Handheld Devices Using the Keystroke-level Model. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems*. ACM, 1605–1608. https://doi.org/10.1145/1056808.1056977

[21] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* 21, 1 (2014), 65–105. https://doi.org/10.1007/s10515-013-0128-9

[22] Object Refinery Limited. [n.d.]. JFreeChart. Web page, http://www.jfree.org/jfreechart/.

[23] Nihan Ocak and Kursat Cagiltay. 2017. Comparison of Cognitive Modeling and User Performance Analysis for Touch Screen Mobile Interface Design. *International Journal of Human-Computer Interaction* 33, 8 (2017), 633–641. https://doi.org/10.1080/10447318.2016.1274160

[24] Jaehyon Paik, Jong W. Kim, Frank E. Ritter, and David Reitter. 2015. Predicting User Performance and Learning in Human-Computer Interaction with the Herbal Compiler. *ACM Transactions on Computer-Human Interaction* 22, 5, Article 25 (2015). https://doi.org/10.1145/2776891

[25] Evan W. Patton and Wayne D. Gray. 2010. SANLab-CM: A tool for incorporating stochastic operations into activity network modeling. *Behavior Research Methods* 42, 3 (2010), 877–883. https://doi.org/10.3758/BRM.42.3.877 Software available at https://github.com/CogWorks/SANLab-CM.

[26] Toni Perković, Shujun Li, Asma Mumtaz, Syed Ali Khayam, Yousra Javed, and Mario Čagalj. 2011. Breaking Undercover: Exploiting Design Flaws and Nonuniform Human Behavior. In *Proceedings of the Seventh Symposium on Usable Privacy and Security*. ACM, Article 5. https://doi.org/10.1145/2078827.2078834

[27] Casey Reas and Ben Fry. 2006. Processing: programming for the media arts. *AI & Society: Knowledge, Culture and Communication* 20, 4 (2006), 526–538. https://doi.org/10.1007/s00146-006-0050-9

[28] Volker Roth, Kai Richter, and Rene Freidinger. 2004. A PIN-entry Method Resilient Against Shoulder Surfing. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*. ACM, 236–245. https://doi.org/10.1145/1030083.1030116

[29] Dario D Salvucci. 2001. Predicting the Effects of In-car Interfaces on Driver Behavior Using a Cognitive Architecture. In *Proceedings of the 2001 SIGCHI Conference on Human Factors in Computing Systems*. ACM, 120–127. https://doi.org/10.1145/365024.365064

[30] Hirokazu Sasamoto, Nicolas Christin, and Eiji Hayashi. 2008. Undercover: Authentication Usable in Front of Prying Eyes. In *Proceedings of the 2008 SIGCHI Conference on Human Factors in Computing Systems*. ACM, 183–192. https:

//doi.org/10.1145/1357054.1357085

[31] M. Angela Sasse, Michelle Steves, Kat Krol, and Dana Chisnell. 2014. The Great Authentication Fatigue – And How to Overcome It. In *Cross-Cultural Design: 6th International Conference, CCD 2014, Held as Part of HCI International 2014, Heraklion, Crete, Greece, June 22-27, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8528)*. Springer, 228–239. https://doi.org/10.1007/978-3-319-07308-8_23

[32] Anil Shankar, Honray Lin, Hans-Frederick Brown, and Colson Rice. 2015. Rapid Usability Assessment of an Enterprise Application in an Agile Environment with CogTool. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*. ACM, 719–726. https://doi.org/10.1145/2702613.2702960

[33] Soar Research Groups, University of Michigan. [n.d.]. Home - Soar Cognitive Architecture. Website, http://soar.eecs.umich.edu/.

[34] Amanda Swearngin, Myra Cohen, Bonnie John, and Rachel Bellamy. 2012. Easing the Generation of Predictive Human Performance Models from Legacy Systems. In *Proceedings of the 2012 SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2489–2498. https://doi.org/10.1145/2207676.2208415

[35] Amanda Swearngin, Myra B. Cohen, Bonnie E. John, and Rachel K.E. Bellamy. 2013. Human Performance Regression Testing. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 152–161. https://doi.org/10.1109/ICSE.2013.6606561

[36] Leonghwee Teo and Bonnie E. John. 2008. CogTool-Explorer: Towards a Tool for Predicting User Interaction. In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*. ACM, 2793–2798. https://doi.org/10.1145/1358628.1358763

[37] The MITRE Corporation. [n.d.]. Cogulator: A Cognitive Calculator. Website, http://cogulator.io/.

[38] Anne Treisman and Janet Souther. 1985. Search Asymmetry: A Diagnostic for Preattentive Processing of Separable Features. *Journal of Experimental Psychology: General* 114, 3 (September 1985), 285–310.

[39] Emanuel von Zezschwitz, Alexander De Luca, Bruno Brunkow, and Heinrich Hussmann. 2015. SwiPIN: Fast and Secure PIN-Entry on Smartphones. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. ACM, 1403–1406. https://doi.org/10.1145/2702123.2702212

[40] Wikimedia Foundation, Inc. [n.d.]. Lisp (programming language). Web page, https://en.wikipedia.org/wiki/Lisp_(programming_language).

[41] Jeremy M. Wolfe. 2001. Asymmetries in visual search: An introduction. *Perception & Psychophysics* 63, 3 (01 Apr 2001), 381–389. https://doi.org/10.3758/BF03194406

[42] Geoffrey F. Woodman and Marvin M. Chun. 2006. The role of working memory and long-term memory in visual search. *Visual Cognition* 14, 4-8 (2006), 808–830. https://doi.org/10.1080/13506280500197397

[43] Geoffrey F. Woodman and Steven J. Luck. 2004. Visual search is slowed when visuospatial working memory is occupied. *Psychonomic Bulletin & Review* 11, 2 (2004), 269–274. https://doi.org/10.3758/BF03196569

[44] Haiyue Yuan, Shujun Li, Patrice Rusconi, and Nouf Aljaffan. 2017. When Eye-tracking Meets Cognitive Modeling: Applications to Cyber Security Systems. In *Human Aspects of Information Security, Privacy and Trust: 5th International Conference, HAS 2017, Held as Part of HCI International 2017, Vancouver, BC, Canada, July 9-14, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10292)*. Springer, 251–264. https://doi.org/10.1007/978-3-319-58460-7_17