Urbas, L., Nekrasova, L. & Leuchter, S. (2005). State Chart Visualization of the Control Flow within an ACT-R/PM User Model. In A. Dix & A. Dittmar (Hrsg.) *4th International Workshop on TAsk MOdels and DIAgrams for user interface desgin TAMODIA 2005*, Danzig (S. 43-48). New York: ACM. http://www.safety-critical.de/doc/tamodia2005.pdf

State Chart Visualization of the Control Flow within an ACT-R/PM User Model

Leon Urbas
Technische Universität Berlin
Center of Human-Machine Systems
Jebensstr.1, Sekr J2-2, D-10623
Berlin
++49 (30) 314-72007
urbas@zmms.tu-berlin.de

Ljudmilla Nekrasova
Technische Universität Berlin
Center of Human-Machine Systems
lne@zmms.tu-berlin.de

Sandro Leuchter
Fraunhofer IITB
Fraunhoferstr. 1, D-76131 Karlsruhe
+49 (721) 60 91-424
sandro.leuchter@iitb.fraunhofer.de

ABSTRACT
We present a novel visualization for ACT-R/PM models of cognitive processes to support the model development. Because the underlying production system paradigm does not specify an explicit flow of control, it is rather difficult to grasp the structure of this kind of user models. Therefore, we developed an algorithm that analyzes the interdependencies of ACT-R/PM productions by resembling the main parts of the matching process of the production cycle. The algorithm produces a graph with nodes as specifications of the state of the declarative memory and edges as productions which are applicable in these states. States are generalized to reduce the complexity of the control flow. The graph is transformed into a state-chart like visual representation. Goal oriented behavior with sub-goaling is considered with sub-graphs. The algorithm is implemented as a plug-in for the integrated development environment eclipse.

Categories and Subject Descriptors
D.2.2, [**Design Tools and Techniques**]

General Terms
Algorithms, Documentation, Economics, Human Factors, Languages.

Keywords
Cognitive User Modeling. Visualization of Flow Control, State Chart

1. INTRODUCTION
ACT-R [2-3,5-6] is a cognitive architecture and a programming environment for user models. These models describe the users' cognitive structures and processes at a fairly atomic level. An ACT-R user model consists of a set of production rules and the specification of initial declarative memory elements organized in a semantic network. In each cycle of the production system, the condition part of all production rules is tested against the current state of the active declarative memory elements. Then one of the matching rules is selected by a conflict resolution algorithm which incorporates a network of sub-symbolic measures. Finally, the action part of the selected rule is executed to modify content and activity of declarative memory elements.

ACT-R not only implements a theory of human associative memory but provides the modeler with detailed mechanisms for perception and motor action. The human mind is abstracted as a modular system. Central executive is realized as a production system core that interacts with perception, motor action, memory and other subsystems via buffers. These buffers implement laws and restrictions of data retrieval and access between the central production system and the modules. The authors of the latest version of ACT-R [3] are convinced that a mapping of some elements of the architecture to certain cortical regions like dorsolateral and ventrolateral prefrontal cortex (DLPFC, VLPFC) or basal ganglia is possible (Figure 1).

The specification of the cognitive architecture imposes severe constraints on how to model user behavior. From an engineering point of view, these constraints are even supporting, because of the guidance they might give to the modeler. Therefore we think that it is more effective and efficient to use cognitive architectures than general purpose/AI languages like Prolog or cognitive toolboxes like COGENT [7].

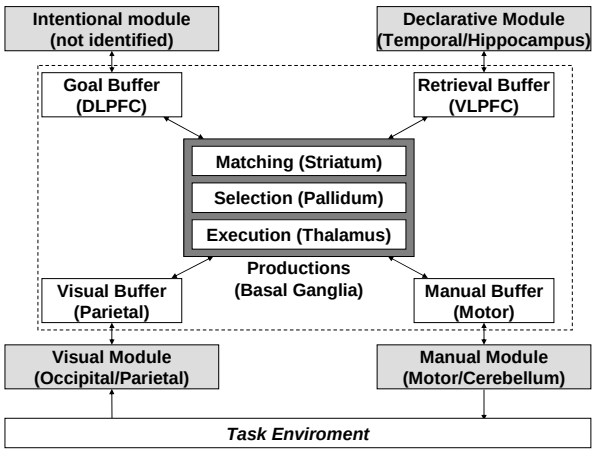

Figure 1. Modular software architecture of the cognitive architecture ACT-R (adapted from [3]).

Despite of its very fine-granular description of users' cognitive processes, ACT-R/PM has been applied in different HCI task domains, e.g. cell phone menu design [1], car driving [12], air traffic control [11], flight management [13], and process control in chemical plants [14].

However, ACT-R/PM is not yet an industrial engineering tool to efficiently describe goal-oriented behavior of users in human-machine-interaction. One reason is, that the underlying production system paradigm does not specify an explicit flow of control (i.e. function application in the functional or method sending in the object-oriented programming paradigm). A conflict resolution algorithm decides at run-time which one of all the possible production rules should actually fire. The purpose of this scheme is to handle program modifications during runtime, to increase robustness to program changes by providing modularity and independence of program elements. The drawback of this programming paradigm shows up in program development (modeling, testing, debugging, and sharing). The lack of explicit representation of control flow makes it rather hard to get familiar with someone else's (including myself after a few days) program. In our opinion, the main key to understanding a user model is the possible sequence of productions.

2. CONTROL FLOW IN PRODUCTION SYSTEMS

A program is a sequence of instructions. Normally the sequence of program instructions (e.g. in a file) is not the same as the observable sequence of instruction processing. The processing sequence is the instantiation of a program instruction sequence. It results from the control flow within the program. In procedural languages the sequence of instructions is explicitly defined by the modeler at programming time. Instructions are normally executed in the order they appear. Control flow statements change this sequential flow of control and allow to execute blocks of statements conditionally (if-else, switch-case) or repeatedly (while, do-while, for). Further language elements specify branching (break, continue, label, goto, subroutine/function-calls, return) and exception handling (try-catch).

Despite of all these high-level statements, programs can be reduced to three basic elements instruction statement, goto-statement, and conditional.

All other cases can be represented as instruction sequences and conditional constructs. Branching to subroutines can be represented as goto-statements, iteration control is the combination of conditional, instruction sequence, and goto-statement.

Thus a program can be represented as a graph with instructions as nodes. Directed edges connect an instruction with its next possible successors. In this notation, a program instantiation is a sequence of nodes that describes a way through the graph from the start instruction to a stop instruction via the edges.

In production systems the sequence of instructions, that is the sequence of production instantiation applications, is not stated explicitly (Figure 2b) but a result of working memory state and conflict resolution algorithm. Only in well defined static problem solving domains one could just record the sequence of production instantiation applications during run-time and analyze this log then after. However, at least in the human-machine interaction domain, the program instantiation depends on externally perceived information. Furthermore, most cognitive architectures utilize stochastic noise at some point of model execution. It is obvious, that visualizing a sequence of instruction processing as shown in Figure 2c can not provide the necessary information to match the description of the production system model (Figure 2b) with the observation. In fact, to serve this purpose one has to derive and display the program graph. Here nodes describe distinguishable states of the system (as defined by the condition part of the productions or the state variables of a procedural language), the directed edges depict possible transformation (by applying the action part of an applicable production) to another distinguishable state of memory.

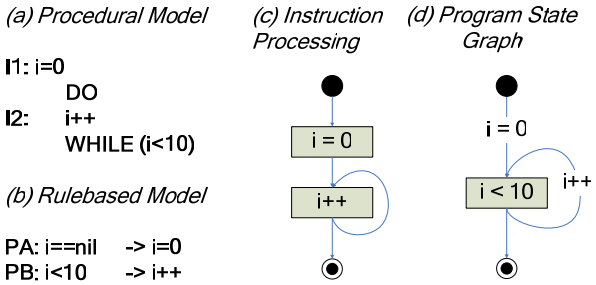

Figure 2. Different Models of a simple counting example program as (a) procedural model, (b) rulebased model, (c) Instruction Processing Graph, (d) Program State Graph

3. EXAMPLE

This section illustrates the basics of the visualization with a simple model of mental addition. The goal of the model, which was taken from the ACT-R 5.0 Tutorial Unit 1 at http://act-r.psy.cmu.edu/tutorials/, is to add two small numbers. These Numbers are given in arg1 and arg2, the result must be smaller than 11. Addition is implemented as a counting process. That is, the algorithm starts the computation with arg1 and increments this by 1 for arg2 times. The number of times the increment was executed is maintained in the variable count.

The structure of the addition-as-counting model is quite simple. The model assumes that the user has a stable representation of the order of numbers. Thus, the initial working memory consists of 10 facts that represent the successors of numbers 0 to 9. The goal itself has four named slots, i.e. name-value pairs. The slots arg1 and arg2 contain the addends, sum represents the result of the addition, and count maintains a counter. The procedure is implemented with four productions, all of them manipulate the same goal, sub-goaling is not necessary.

Initialize-Addition: This production can be applied if sum is nil, that is no value has yet been assigned to this variable. The action part of this production sets the sum slot of the goal to arg1, and count is set to 0. Furthermore, the retrieval buffer is asked for a declarative memory element that represents the successor of arg1.

Terminate-Addition. It can be applied if sum is not nil and arg2 and count have the same value i.e. count was arg2 times incremented by 1. In that case count is set to nil. The previously described initialize-addition production can not be applied after that because sum is not nil. No other production can be applied either thus the computation stops.

Increment-Sum matches if the goal's slot sum is not nil, count is not nil, and the retrieval buffer holds a fact about the successor of sum. Once the production is executed, sum is set to the retrieved successor and the retrieval buffer is asked for the successor's successor.

Increment-Count: This one can be applied if the goal's slot sum is not nil, count is not nil and the retrieval buffer holds a fact about the successor of count. The application of this production sets count to its successor and the retrieval buffer is asked for the successor of sum.

Setting the goal to add the number five and two (second-goal ISA add arg1 5 arg2 2)[1] results in a sequence of instruction processing as tabulated in Table 1. From this sequence one can see, that model alternates between incrementing the count from 0 to 2 and incrementing the sum from 5 to 7. Initialize–Addition starts things going and requests a retrieval of an increment to the sum, i.e. the successor. Increment-Sum processes that retrieval and requests a retrieval of an increment to the count. That production fires alternately with Increment-Count, which processes the retrieval of the counter increment and requests a retrieval of an increment to the sum. Terminate-Addition recognizes when the counter equals the second argument of the addition and modifies the goal in a way that no condition of any production can match. This is simply done by setting count to nil. In consequence, the program stops.

Table 1. Sequence of Instruction Processing for the addition-as-counting Example while adding five and two

Time [sec]	Name of Production Fired
0.050	Initialize-Addition
0.150	Increment-Sum
0.250	Increment-Count
0.350	Increment-Sum
0.450	Increment-Count
0.500	Terminate-Addition
0.500	* Nothing to run: No productions, no events.

We use this introductory example as starter in our modeling courses. Despite of its simplicity, it is our experience, that most of our students have severe difficulties to predict the observed instruction processing sequence from the user model code. In our opinion the main barrier is the unfamiliarity with the production system approach – most of our students do not have any artificial intelligence background.

But, if we support them with a state graph, showing the states of the goal buffer as edges and the productions as transitions (Figure 3), we can observe that the process of understanding is significantly furthered. Up to now, no empirical sound study was conducted, so it remains unclear if this is a transferable observation that in fact can be attributed to the graphical support (and not to the enthusiasm of the teacher about this didactical tool).

[1] The slots sum and count, which are referenced by the productions, are initialized to nil by default

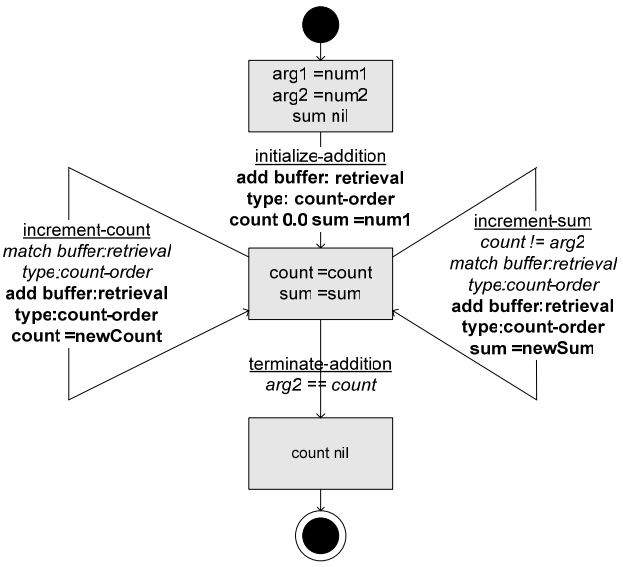

Figure 3. Program State Graph of the addition-as-counting model of the ACT-R/PM Tutorial Unit 1. Nodes denote states of the goal buffer, each edge stands for one production rule and is labeled by the productions name, additional conditions on other buffers and the modification on buffers

4. GENERALIZED STATES

The conflict resolution algorithm of ACT-R/PM checks all of the buffer conditions – goal, memory, perception and motor system, etc. Considering all of the combinations of conditions that are described by the condition part of the set production would most probably result in an explosion of the state space. Therefore some reduction strategy is necessary. A natural choice is to reduce the analysis on the goal buffer only. This is because the goal buffer is of particular importance in ACT-R. The goal is normally used to represent different stages of problem solving and plan execution and to preserve problem specific data over the course of production execution. A heavily used design pattern in ACT-R user models is to enforce a particular sequence of productions by manipulating the goal. If one production application should be followed by another, the first production modifies the goal's state in its action part in a way that the second production is triggered in the next production cycle.

In our implementation of the program state graph, nodes represent goal states, which trigger certain production sets. Goal states are relevant assignments of the current goal's slots. Transitions between states will be conducted by productions. At this point of time, every production in the model is represented by exactly one transition in the state chart.

To make the state chart more concise, we try to reduce the set of possible assignments to goal conditions to generalized goal states. This is accomplished by a unification process. This process seeks to generalize the specification of the slots of a goal state. Slots can be set to a certain value, to any value but a certain one, be bound to no value or be bound to any value.

5. SUB GOALING

Sub-goaling and creation and handling of parallel goals are major features of ACT-R/PM models. In the state chart visualization every goal is treated as a super-state. The goal states and transitions between them are represented as sub-states and transitions between them. Goal creation and release is represented as transitions between super-states (see Figure 9).

6. IMPLEMENTATION DETAILS

6.1 Goal State Generalization

The details of the algorithm are described in [10]. In this paper, we concentrate on the central element of goal states generalization. In particular in recursive models one might observe, that the number of distinguishable states (without start and stop states) might be less than the number of patterns found in the condition and action parts of the production rule. This is illustrated in Figure 4. The two production rules P1: S1 → S2 and P2: S3 → S4 both operate on the same three-slot goal type. S1 and S3 are the condition patterns, S2 and S4 are action patterns that will be applied on the goal element, whenever the related production fires. Table 2 lists the condition and action patterns in detail.

Table 2. Condition and Action patterns of goal state generalization example

Production	Goal Pattern	Slot1	Slot2	Slot3
P1	Condition: S1	A	=var1	=var2
	Action: S2		=var3	
P2	Condition: S3		=var4	
	Action: S4	nil		

Within the context of the model, the goal states S2 and S3 can be generalized by S1. That means that executing S2 does not change the principal nature of the goal state and S3 matches whenever S1 matches. Given an initialization of the goal state that fills all of the slots, one can easily derive the program state graph given in figure 4.

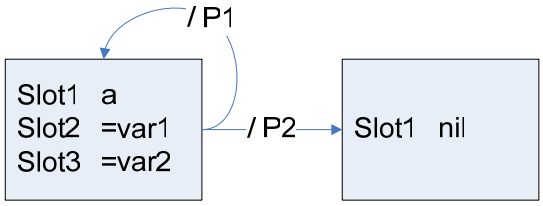

Figure 4. Program state graph for the goal state generalization example

The identification and generalization of goal states is accomplished by means of the matching operator M. This operator compares the condition part pattern of a production with the current goal. The comparison process is not symmetric as the current goal is guarantied to be more specific than any matching condition part pattern. This means that for all productions that match with the goal buffer the following is true: (1) the slots of the condition part as well as the action part of the production that refers to the goal pattern constitute a subset of the slots of the goal buffer, and (2) the values of the production slots do match the values of the goal buffer slots or can be unified.

This leads to the following formal definition of the matching operator M [10]. Given the following data structures and relations

goal state S = (goal-type, slots)
slots = {sl | sl = (name, symbol)}
symbol = (type, value)
type ∈ { nil, var, const}
g : S → goal-type
s: S → slots
Symbol : sl → symbol
Value: symbol → value
Name: sl → name
Type: symbol → type

Let S1 and S2 be two goal states, then

S1 M S2 == true

iff

L1: g(S1) == g(S2) &&
L2: ∀ sl_1 ∈ s(S1) ∃ sl_2 ∈ s(S2) :
L3: Name(sl_1) == Name(sl_2) &&
L4: (Type(Symbol(sl_1))==Type(Symbol(sl_2)) == nil ||
L5: Type(Symbol(sl_1)) == Type(Symbol(sl_2)) == var ||
L6: (Type(Symbol(sl_1)) == Type(Symbol(sl_2)) == const &&
L7: Value(Symbol(sl_1)) == Value(Symbol(sl_2))) ||
L8: (Type(Symbol(sl_1)) == var &&
L9 : Type(Symbol(sl_2)) == const))

Let's put this formal description into words: Line 1 (L1) tells us, that S1 and S2 may only match if they relate to the same goal-type. (L2) If this is granted, then for every slot of S1 there must be a matching slot in S2. Two slots match if both (L3) name and (L4-L9) content of the slot match. The content of two slots match if one of the following conditions is true: (L4) both are empty (nil), (L5) both are to be bound (var), (L6-7) both refer to constants with the same value or finally if (L8-9) slot 1 can be bound and slot 2 refers to a constant.

If we now apply the match operator M on the goal states S1, S2, S3 and S4 as given in figure 4 we will easily see, that S2 M S1 == true (L2, L5). However, S1 M S2 == false because (L2) fails. Both conclusions are true for S1 and S3. Finally, S4 M S1 == false; nil and var do not match. This leads to the result, that S2 and S3 can be generalized to S1 while S4 is a distinct state. This results in the graph as shown in figure 4.

6.2 Visualization

The program state graph is visualized using the graphical notation of State Charts [8]. Figure 5 shows the basic elements: Boxes, labeled with S and T represent two distinguishable states. The arrow from S to T represents a state transition.

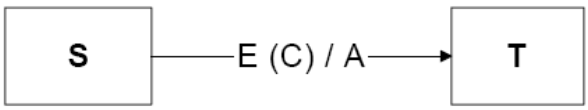

Figure 5. State Chart Notation for reactive systems. S,T are states, E is the Event that triggers the transition, C a condition that has to be fulfilled and A describes the Action taken during transition

The label at the transition follows a certain format and contains elements, which specify the transition. E denotes the event that triggers the transition, C describes a condition that has to be true for the transition to occur, and A is the action taken during transition. This set of basic elements is completed by special symbols for the initial state (filled circle) and the stop states (filled circle with white border).

To apply this well-known schema to program state graph visualization we modified it gently, as illustrated in Figure 3. The nodes (boxes with round edges) are labeled with the description of the slots of the generalized goal state they stand for. Each edge represents a single production. Instead of event E we label the edge with the name of the production. The tests on additional buffers that are not part of the generalized goal state analysis, but have to be fulfilled to allow the production to become part of the conflict set, are noted as condition C. However, we chose to not use brackets but to code this by color and an italic font type. The same is true for the action part of the production. This naturally maps to the action-element A of the state chart notation, once again due to the expected amount of text we decided to utilize color coding and a bold font type instead of the separating slash.

6.3 Implementation as Eclipse Plugin

We decided to implement the proposed algorithm and visualization tool within the Eclipse environment. Eclipse is a framework for custom integrated development environments. It was explicitly designed to be extended. This is done by means of plug-ins. A plug-in provides a new editor for a defined format. The editor extends the internal list of known capabilities for the defined file type and is activated by the user. In our case this is an ACT-R model.

From a programmers point of view the editor implements a public interface that defines the externally exposed functionality of all editors. This common interface allows, to integrate the new plug-in into the framework just as any other editor. The user won't see any difference, and indeed there is no difference, because eclipse's own editors are in fact plug-ins, too.

The plug-in for ACT-R/PM Model Visualization builds upon the Graphical Editing Framework (GEF). GEF is a plug-in that builds on different layers (Figure 6) and provides support for the drawing of primitive figures like boxes, connecting lines, text widgets, etc. Furthermore, GEF already implements some interaction primitives like the dragging of elements, undo/redo interfaces, etc.

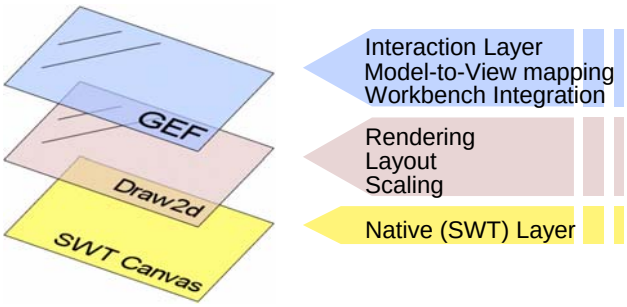

Figure 6. The GEF Layered Architecture builds upon other plug-ins of the Eclipse Framework and provides easy access to the functionally of those layers [4]

GEF implements a model-view-controller design pattern. To implement an editor, one has to provide three distinct packages. The model package provides data representation classes for each graphical element that should be displayed. The edit package contains the implementation for handling the graphical elements and the bend- and endpoints of each connection between elements. Finally, the classes of the figure package implement methods to display the graphical representations of all model elements in the editor window. A well written application-oriented documentation of the interfaces and functionality of the different layers is given by [9].

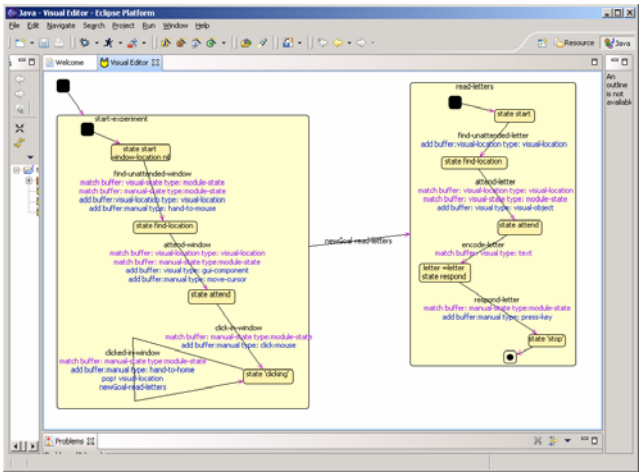

Figure 9. A screen-shot of the editor with a model that includes a sub-goal

7. CONCLUSION

The visualization for ACT-R user models has the potential to become a supporting software engineering and modeling tool. In particular it might be handy in the stage of implementing a formal model that describes a rule-based sequence of actions into the production paradigm of the cognitive architecture (Figure 9).

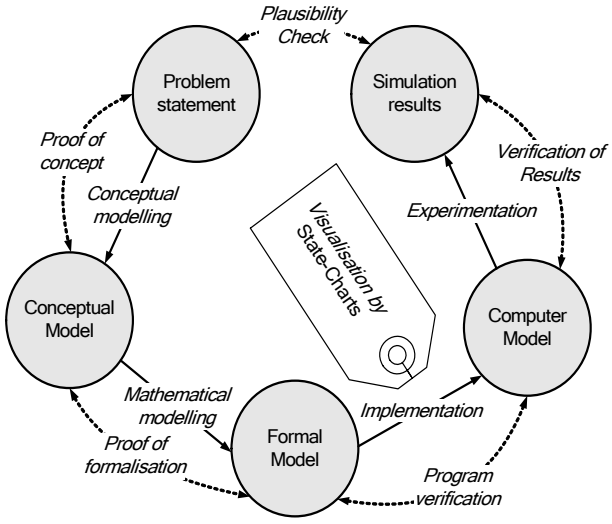

Figure 9. Within the user modeling process the visualization tool helps during implementation and program verification

At this moment however, the tool is hardly more than a proof of concept. It is used at our department to visualize the state structure of user models and thus allows us to gain insight on the control flow in complex cognitive models. On the top of the wish list is an improved layout manager. Due to the simplicity of the current implementations layout manager all models need to be rearranged spatially. The second important limitation of the current implementation is the scalability. It is not yet possible to cluster visualization elements or to zoom out. Further improvements will address these limitations: A more sophisticated layout manager and support for exploring large models is under construction.

The second direction of further development aims towards support functions for model analysis. Since ACT-R/PM models are internally represented as graphs it is possible to use simple measures about the connectedness of sub-graphs to partition the visualization in more or less independent clusters. Thus this tool is a first step towards a visual programming and modeling environment. A loose integration with the ACT-R/PM interpreter could make it an integrated editing and simulation environment.

Applied cognitive modeling will become an important engineering tool for analyzing and designing human-machine systems. It will clearly benefit from any software engineering support and more development tools like the visualization of ACT-R/PM models that was presented in this paper. The integration into a development environment would then make modeling even more efficient. Due to its highly modular plug-in concept the Eclipse framework would be a first choice candidate as an integration platform.

8. ACKNOWLEDGMENTS

This work was funded by VolkswagenStiftung within the Research Program “Junior Research Groups at German Universities”. Our special thanks go to Prof. F. Wysotzki, TU Berlin, for his lively interest and his support.

9. REFERENCES

[1] Amant, R. St., Horton, Th. E., and Ritter, F. E. (2004). Model-based evaluation of cell phone menu interaction. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI 2004)*. Retrieved Feb 28, 2005, from http://www.csc.ncsu.edu/faculty/stamant/papers/RSA-TEH-FER-chi04-actr.pdf.

[2] Anderson, J. R., and Lebiere, C. (1998)(Eds.). *Atomic Components of Thought*. Hillsdale, N.J.: Erlbaum.

[3] Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, Ch., and Lebiere, Y. Q. (2004). An integrated theory of the mind. *Psychological Review 11(4).* 1036-1060

[4] Bokowski, B. (2005) *Maßgeschneiderte grafische Editoren mit GEF[tailor-made graphical editors with GEF]*. Presentation at JUGS, SIG Eclipse (Stuttgart, 3.2005). Retrieved Jun 5, 2005 from http://www.eclipseteam.de/wiki/pub/Public/EclipseGef/GEF-JUGS.ppt

[5] Byrne, M. D. (2001) ACT-R/PM and menu selection: applying a cognitive architecture to HCI. *Int. J. Human-Computer Studies (2001) 55*, 41-84

[6] Byrne, M. D., and Anderson, J. R. (1998). Perception and Action. In J. R. Anderson, & Ch. Lebiere (Hrsg.) *Atomic Components of Thought*. Mahwah, NJ: Erlbaum. 167-200

[7] Cooper, R. P. (2002). *Modelling High-Level Cognitive Processes*. Mahwah, NJ: Lawrence Erlbaum Associates.

[8] Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3), 231-274.

[9] Moore, B., Dean, D., Gerber, A., Wagenknecht, G., & Vanderheyden, Ph. (2004). *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework.* IBM (Redbooks). Retrieved Jun, 5 2005 from http://www.redbooks.ibm.com/redbooks/pdfs/sg246302.pdf

[10] Nekrasova, L. (2005) *Programmvisualisierung von Produktionssystemen am Beispiel der kognitiven Modellierungssprache ACT-R [Program Visualization of production systems, exemplified with the cognitive architecture ACT-R]*. Master Thesis. TU Berlin, Center of Human-Machine Systems.

[11] Niessen, C., Leuchter, S., and Eyferth, K. (1998). A psychological model of air traffic control and its implementation. In *Proceedings of the Second European Conference on Cognitive Modelling (ECCM-98)*. Nottingham: Nottingham University Press. 104-111 Retrieved Feb 28, 2005 from http://www.zmms.tu-berlin.de/~sandro/doc/eccm98.pdf.

[12] Salvucci, D. D. (2001). Predicting the effects of in-car interface use on driver performance: an integrated model approach. *International Journal of Human-Computer Studies, 55(1),* 85-107.

[13] Schoppek, W., and Boehm-Davis, D. A. (2004). Opportunities and challenges of modeling user behavior in complex real world tasks. *MMI interaktiv, 7,* 47-60. Retrieved Feb 28, 2005, from http://useworld.net/ausgaben/06-2004/05-Schoppek_Boehm-Davis.pdf.

[14] Wallach, D. (1996). *Komplexe Regelungsprozesse: Eine kognitionswissenschaftliche Analyse [complex control processes: a cognitive science analysis]*. Wiesbaden: DUV.