Software Engineering for Cognitive Modeling: Visualization of Production Systems

Sandro Leuchter, Ljudmila Nekrasova & Leon Urbas

MoDyS Research Group, Center of Human-Machine Systems, Technische Universität Berlin, Germany
Sekr. J2-2, Jebensstr. 1, D-10623 Berlin, Germany
sandro.leuchter@tu-berlin.de, lne@zmms.tu-berlin.de, leon.urbas@zmms.tu-berlin.de

Abstract

Simulations of mental processes during the operation of technical systems can be a valuable source of information for system design, as part of support systems, for developing trainings, and to classify human errors. A current approach is to ground such human performance models on cognitive architectures like EPIC or ACT-R/PM. The founding of these architectures on theories of cognition leads to comparatively fine-grained models. As an example ACT-R/PM offers detailed mechanisms in the areas of memory and perception/action. Thus cognitive models become very complex to develop and understand. An analysis of the production system paradigm that underlies most of the cognitive architectures revealed that the development is very complex because the control flow is not explicitly represented. Since there is currently no widely used diagram type for production system based cognitive models we propose a new visualization for these models that incorporates information on the control flow.

The control flow is detected with a new algorithm for analyzing the interdependencies of ACT-R/PM productions. It mainly resembles the matching process of the production cycle. It does not include a detailed prediction of the outcome of the conflict resolution. This would result in a too fine grained chain of productions, that would not help understanding the purpose and structure of a model. The result of our algorithm is a directed graph. Nodes denote unique states of the declarative memory and productions which are applicable in these states are drawn as edges. States are generalized to reduce the complexity of the control flow. The graph is transformed into a statechart like visual representation. Goal oriented behavior with sub-goaling is considered with sub-graphs. The algorithm was implemented as a plug-in for the integrated development (IDE) environment eclipse.

This paper discusses the use of cognitive modeling for analyzing interactive technical systems. It also addresses the cognitive architecture ACT-R/PM as a modeling environment. We outline how abstraction and visualization as a software engineering methodology augments this model-based engineering approach. The article describes the visualization algorithm, its implementation as an eclipse plug-in and presents an example visualization.

1 Introduction

Simulations of mental processes during the operation of interactive technical systems can be applied to support system design, as part of support systems, or for developing trainings. A current approach is to ground such human performance models on cognitive architectures like EPIC (Kieras & Meyer, 1997) or ACT-R/PM (Anderson & Lebiere, 1998). From an engineering point of view it is more effective and efficient to utilize cognitive architectures than general purpose AI languages like Prolog or cognitive toolboxes like COGENT (Cooper, 2002) because the person developing the model is constrained and guided by the assumptions and specifications of the architecture. The level of description of cognitive architectures leads to comparatively fine-grained models. As an example ACT-R/PM offers detailed mechanisms in the areas of memory and perception.

1.1 Human-Machine Systems

Timpe, Giesa and Seifert (2004) describe the concept of human-machine systems as an “abstraction of the goal-oriented cooperation between persons and technical systems to accomplish a self-determined or assigned task […]. [This] implies that at least one person and at least one machine work together. The general structure is a control system in which an operator and/or a team make decisions to steer the technical system according to organizational basis, goals, task, and perceived feedback about the environment and the process conditions". Examples for such work environments are aviation, process control in chemical or (nuclear) power plants, automotive systems, and ship guidance. To reach objectives like high dependability and high performance of the whole human-machine system, high usability of the interfaces and adequate workload, general well-being and health of operators systems engineers not only have to obey "physical" laws for the proper design of the technical part of the system but also need to consider human data processing based on social, psychological and biological regularities to shape the goal-oriented information interchange between humans and machines.

1.2 Model Based Engineering

It is a current trend in applied ergonomics to deploy human performance models based on cognitive architectures to assess human factor measures and simulate the overall ability to operate a human-machine system (Ritter et al., 2003). ACT-R/PM has already been applied to modeling interaction in different HCI task domains within some research projects: e.g. cell phone menu design (Amant, Horton & Ritter, 2004), car driving (Salvucci, 2001), air traffic control (Niessen, Leuchter & Eyferth, 1998), flight management (Schoppek & Boehm-Davis, 2004), and process control in chemical plants (Wallach, 1996). But ACT-R/PM is not yet an effective industrial engineering tool. The production system paradigm adopted in this cognitive architecture is one reason for that.

2 ACT-R/PM as a Production System For Cognitive Modeling

ACT-R/PM is a production system for modeling cognitive processes and memory structures (Anderson & Lebiere, 1998). ACT-R/PM provides both a cognitive architecture and a programming environment. Programs are models of cognitive structures and processes. They contain production rule sets and the specification of initial declarative memory elements organized in a semantic network. ACT-R/PM contains a fine grained perception and motor action subsystem. It also incorporates several learning mechanisms on the symbolic (creation of new productions rules and declarative memory elements) and sub symbolic (tuning of conflict resolution parameters) level.

2.1 Production Systems

Production systems consist of two memory stores: The production memory consists of a set of productions. The working memory holds declarative data e.g. object representations. The content of the declarative memory can be accessed by the productions. A production follows the schema "IF Condition THEN Action". The condition part, often called left hand side, is tested against the elements of the declarative data store, the action part (right hand side) of the production modifies the declarative data store.

Production system processing can be interpreted as a syntactic transformation process. Data patterns in the working memory are transformed through subsequent production application. The production system scheme has been deployed in cognitive science modeling for several reasons:

- the processing paradigm fits well into the physical symbol theory (Newell 1980)
- because of the redundancy of applicable productions in terms of knowledge engineering it is robust against program changes
- the complexity of cognitive processes is encapsulated in the conflict resolution mechanism (see below).

Programs for production systems do not specify an explicit flow of control (i.e. function application in the functional programming paradigm or method sending in the object-oriented programming paradigm). The decision about the sequence of production-rule applications is delayed to the production cycle at runtime:

1. In the *evaluation cycle* the conditions of all production rules are tested against the current state of the declarative memory. All matching productions are instantiated.
2. One or more production instantiations are chosen for execution by the *conflict resolution* algorithm according to their currently assumed value. In ACT-R/PM exactly one of the matching rules is selected. The ACT-R/PM conflict resolution algorithm incorporates a network of sub-symbolic measures.
3. The chosen production instantiation(s) is (are) *executed*. Actions apply to modifications of the working memory and to output. If there is more than one production instantiation chosen (not in ACT-R/PM) then they are executed in parallel.

Thus productions do not call other productions directly. Instead they modify the working memory accordingly to trigger the application of other productions. The complexity of decision making for choosing the next production out of several applicable is transferred to the conflict resolution algorithm. The conflict resolution algorithm uses a more or less complex predefined measure to assess the value of production instantiations. Many production systems use only the one best instantiation.

The purpose of this scheme is to handle program modifications during runtime, to increase robustness to program changes, and providing modularity and independence of program elements. The drawback of this programming paradigm shows up in program development (modeling, testing, debugging). The lack of explicit representation of control flow makes it very hard to get an overview of someone else's program, to communicate about a program, and thus to develop in a team. Debugging is also obstructed.

2.2 Control Flow

A program is a sequence of instructions. Normally the sequence of program instructions is not the same as the observable sequence of instruction processing (e.g. call of subroutines and non-imperative programming style). The processing sequence is the instantiation of a program instruction sequence. It results from the control flow within the program. The control flow consists of three different elements:

- instruction statement
- goto-statement
- conditional

All other cases can be represented as instruction sequences and conditional constructs:

- *Call of subroutines* can be represented as goto-statement.
- *Iteration* can be represented as conditional, instruction sequence, and goto-statement.

Thus a program can be represented as a graph with instructions as nodes. The next possible instructions are connected with directed edges. A program instantiation is a sequence of nodes that describes a way through the graph from the start instruction to an halt instruction via the edges.

For the purpose of this paper complexity of a program is defined as the connectedness in the graph. This reflects the ease of finding a program instantiation. Goto-statements and instruction statements do not add complexity to a program. The complexity of the control flow depends sole on conditional constructs. Thus conditionals are analyzed in more detail.

There are different interpretations on conditional expressions possible. They depend on the programming paradigm:

- IF <cond1> THEN call <sub1> ELSE IF <cond2> THEN call <sub2> … ELSE call <sub3> FI
  This is the structured programming approach directly derived from **imperative programming**. There is a direct connection between the conditional and the following instructions. Thus it is very easy to get an overview of the control flow (e.g. flowchart with ISO 5807/DIN 66001).
- do B1 → SL1 [] … [] Bn → SLn od
  This is the *guarded expressions* view introduced by Dijkstra (1975). The conditions need not to be disjunctive. It is not predetermined which expression will be evaluated if several conditions are met. This approach is often deployed in **functional programming** languages for the definition of segments of partial functions.
- conclusion(X) :- cond(X)
  The logical interpretation of conditions is realized as rules in **logic programming languages** like prolog (Clocksin & Mellish, 1987). The resolution calculus used to interpret logical programs spans up a tree of inferences. Since in this calculus the conditions of rules are interpreted as conclusions of other rules it gets to direct "calls" of other rules. The semantics for these calls is embedded into the inference engine.
- <cond> → <action>
  In the **production system approach** (Post, 1943) the sequence of production instance application is not ad hoc visible. The conflict resolution algorithm decides which production instantiation follows which one.

Additionally production instantiations at run-time and production declaration at programming-time are not the same.

The declarative nature of this list's programming paradigms is increasing. The more decisions about conditionals are delayed to run-time and the more decisions are transferred to the run-time engine the higher the complexity of programs: It is less easy to get an overview how the program graph looks like. The comparison of interpretations of conditionals in different programming paradigms shows that declarative paradigms make it harder to predict and understand the control flow at programming time.

2.3 ACT-R/PM as a Cognitive Architecture

ACT-R/PM is a production system. Declarative knowledge is represented as chunks having an activation level which can spread over semantic relations between chunks. Productions have a strength value that represents how successful they were used in the past. Processing is guided by goal structures and the working memory configuration. The conditions of ACT-R productions match the working memory state i.e. chunks and goals. Actions can modify or delete existing or create new chunks as well as create sub-goals. The conflict resolution algorithm is complex and incorporates the activation and strength parameters. The purpose of ACT-R/PM is to provide an integrated theory that is able to account for as many cognitive phenomena as possible. The processing scheme directed by the conflict resolution algorithm resembles some cognitive phenomena including learning, and constraints and limitations of memory.

ACT-R/PM has a modular structure. The modules are organized as buffers. Productions can send commands to buffers and use their content for processing. The goal and the retrieval buffers are the interfaces to the working memory. Additional visual, visual-location, aural, and motor buffers provide perception and permit action taking in simulated environments. They incorporate ergonomic knowledge about perception and action execution.

3 Software Visualization with statecharts

The use of ACT-R/PM in applied modeling for human-machine system design and analysis results in big and complex cognitive models with several hundred productions. It is not easy to determine possible sequences of production applications and interference in such complex ACT-R/PM models. Thus there is a need for support in understanding and maintaining ACT-R/PM models. Model visualization is a possible way to support understanding of models.

Price, Baecker & Small (1993) introduced a taxonomy for software visualizations. They take scope, content, form, method, interaction, and effectiveness into account. For the purpose of this paper only content is of interest. The taxonomy refines content as algorithm, program, fidelity and completeness, and data gathering (see Figure 1).

Algorithm vs. program: A visualization can either present a concrete program implementation or a high level representation of the underlying algorithm. In either case the procedural and the data aspect can be visualized. The procedural aspect is the control flow i.e. the sequence of instructions. The data aspect is the structure of the data and the data flow i.e. which data element is used, manipulated and transferred in the program steps.

Fidelity and completeness: The presentation can but does not necessarily need to show the complete behavior. If it is run concurrently to parallel processes it might interfere with the visualized program.

Data gathering time: Program visualization needs information on what to present. Different systems can use different sources of information: at compile-time, at run-time, or both. If run-time data is taken into account there might also be a mapping between program time and visualization time or spatial layout. Visualization during run-time can be online or produced after program run-time.

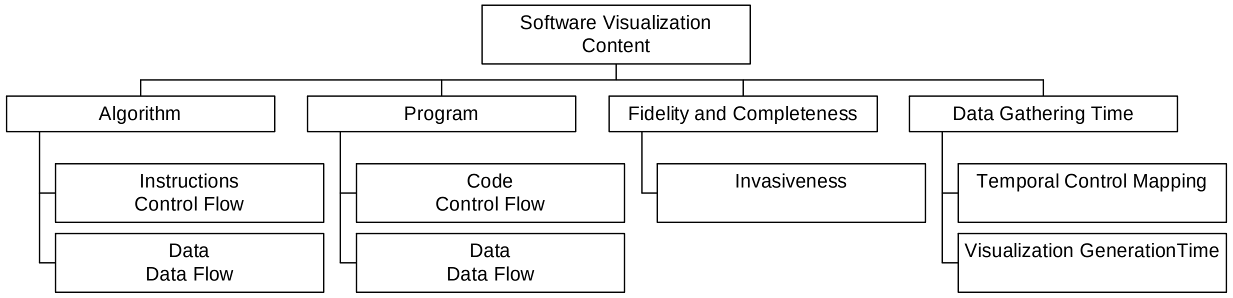


Figure 1: Taxonomy for content aspect in software visualization (Price et al., 1993)

A well known example for program visualization is the statechart notation (Harel, 1987). It can be used to specify the behavior of reactive systems. It extends the conventional state machine approach with the notion of sub- and super-states (depth), the AND-state, broadcast-communication and history elements. The statechart notation can be used to visualize algorithms. Since it has a full denotational semantics definition it is a programming language for itself and can be therefore used to visualize programs. The statechart notation fits in with the taxonomy shown in Figure 1 in the following way:

- *Algorithm/Program:* Statecharts do only visualize control flow. There is no direct feature to show data or data flow.
- *Fidelity and Completeness:* Statecharts can represent the complete behavior of a reactive system. They are not invasive.
- *Data Gathering Time:* Statecharts are normally used in advance as a modeling or programming tool. But statecharts can also be generated from log-files. There is no relation between spatial layout and program-time characteristics.

4 Visualization Algorithm

There are some attempts for the visualization of cognitive models based on production systems. They focus on data or run-time visualization:

- There is a notation for problem space trees and thus the data flow in the SOAR literature (data visualization at program design time).
- The goal hierarchy and thus the main data structure of ACT-R models is also often visualized (data visualization at program design time).
- CaDaDis is a system for displaying production application as a PERT-chart at run-time (Tor, Haynes, Ritter & Cohen, 2004) (program visualization at run-time).
- The ACT-R environment features a similar visualization for the use of buffers (program visualization at run-time).

But there are no formal descriptions or guidelines for visualizing control flow in production systems at programming-time. The remainder of this section describes a new visualization technique to present the control flow of a ACT-R/PM program. It is based on the statechart notation.

4.1 Control Flow

The new visualization uses the statechart notation and semantics to visualize control flow in ACT-R/PM production systems at programming time. The control flow of production systems is the sequence of production instantiation application. Obviously it is not concise enough to get an overview of the program functionality to visualize the whole sequence of production instantiation applications (program instantiation). Additionally the program instantiation depends on externally perceived information in modeling for human-machine system domains and stochastic noise in the case of cognitive modeling. Instead the program graph should be displayed.

The conflict resolution algorithm of ACT-R/PM is too complex to take it into account for the construction of the graph. Instead only the matching of goal conditions is used. The goal is normally used to represent different stages of problem solving and plan execution (i.e. control flow). If one production application should be followed by another one the first production modifies the goal's state in its action part so that the second production is triggered in the next production cycle.

The statechart visualization uses states to represent goal states that trigger certain production sets. Goal states are relevant assignments of the current goal's slots. Transitions between states represent productions. Every production is represented by exactly one transition in the statechart. Goal states are generated when productions are processed for the visualization.

Goal states are generalized to make the statechart more concise. A goal state is characterized by a relevant subset of its slots and their assignments. The slots of a goal state can be set to a certain value, be set to any value but a certain one, be bound to no value or be bound to any value. A unification process is used to generalize goal states.

Sub-goaling and creation and handling of parallel goals is an important feature of ACT-R/PM. In the statechart 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.

4.2 Visualization

In this paper we use the addition model from the ACT-R 5.0 Tutorial Unit 1 (http://act-r.psy.cmu.edu/tutorials/, Version June 2004) to demonstrate the new visualization. The goal of this model is to add two small numbers arg1 and arg2 (the result must be smaller than 11). The algorithm is to start the computation with arg1 and incrementing it arg2 times by 1.

The model consists of four productions. They all manipulate the same goal. The model does not use sub-goaling. The retrieval buffer is the only buffer used besides the goal buffer. The working memory consists of 10 facts (declarative memory elements) that represent the successors of figures 0 to 9. The goal has four slots: arg1 and arg2 hold the addends, sum represents the result of the addition, count holds a counter.

The first production initialize-addition can only be applied if sum is still nil (i.e. has not yet been assigned a special value). In that case the sum slot of the goal is set to arg1, count is set to 0. The retrieval buffer is asked for a declarative memory element that represents arg1 + 1.

The second production is 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 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.

The third production is increment-sum. It can be applied is the goal's slot sum is not nil and count is not nil and the retrieval buffer holds a fact about the successor number of sum. If the production is executed, sum is set to its retrieved successor. The retrieval buffer is asked for the next successor.

The last production is increment-count. It can be applied if the goal's slot sum is not nil and count is not nil and the retrieval buffer holds a fact about the successor number of count. If the production is applied then count is set the its successor and the retrieval buffer is asked for the successor of sum.

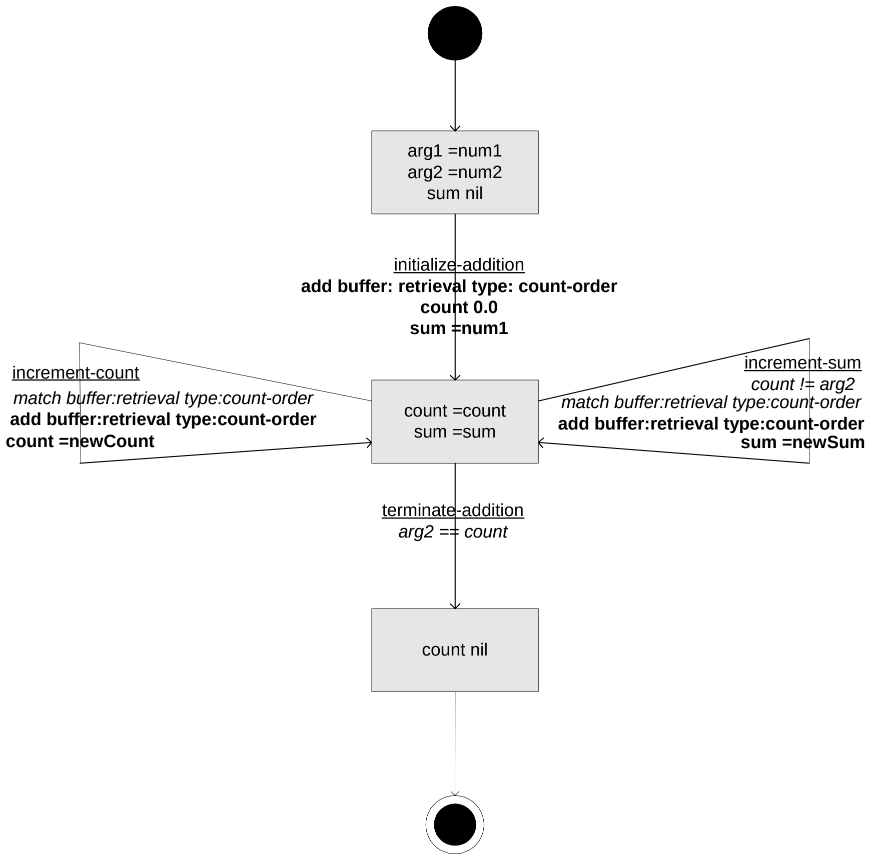

arg1 =num1
arg2 =num2
sum nil
initialize-addition
add buffer: retrieval type: count-order
count 0.0
sum =num1
increment-count
match buffer:retrieval type:count-order
add buffer:retrieval type:count-order
count =newCount
count =count
sum =sum
increment-sum
count != arg2
match buffer:retrieval type:count-order
add buffer:retrieval type:count-order
sum =newSum
terminate-addition
arg2 == count
count nil

Figure 2: Visualization of addition model from ACT-R Tutorial Unit 1

The visualization of the addition model does not present information about the addition-facts (see Figure 2). It does only show how the four productions are used under which goal states. The nodes in the visualization represent goal states i.e. relevant assignments of the current goal's slots. The transitions represent production applications. For example there are three different productions that can be applied in the second state: increment-count, increment-sum, and terminate-addition. Transitions hold additional information on the production application. The first is the name of the applied production. It is possible that there is additional information on conditions (italic text in Figure 2) and actions (bold text in Figure 2) of the productions. The visualization of the addition model reveals the complete control flow in a condensed form.

5 Implementation

The algorithm was implemented in an existing software architecture. The integrated development environment eclipse was chosen because it offers a framework for graphical software model representations and can be extended with new functionality.

5.1 Plug-In in the Eclipse Integrated Development Environment

Eclipse is an integrated development environment for programming with different languages. It supports for example Java, Perl, and C++. Eclipse is implemented in Java. It is based on a software architecture that allows it to be extended with new functionality. Eclipse uses a plug-in architecture to achieve this (see Figure 3).

Figure 4 shows how to extend eclipse with new capabilities using a plug-in: The plug-in provides a new editor for a defined file type (e.g. ACT-R model). The editor extends the internal list of known capabilities for the defined file type and thus can be activated by the user. The editor implements a public interface that defines the externally exposed functionality of all editors. This interface makes it possible for the workbench to call the new editor provided by the plug-in.

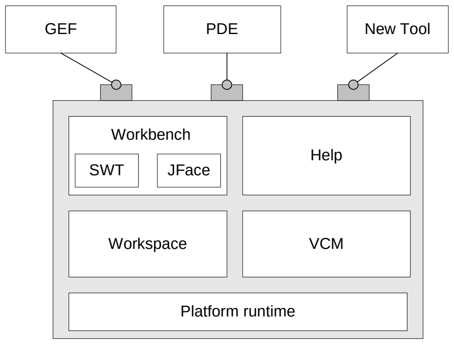


Figure 3: eclipse software architecture (Gallardo, Burnette & McGovern, 2003, p. 8)

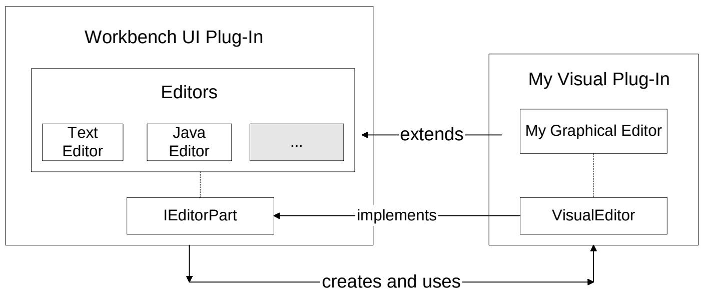
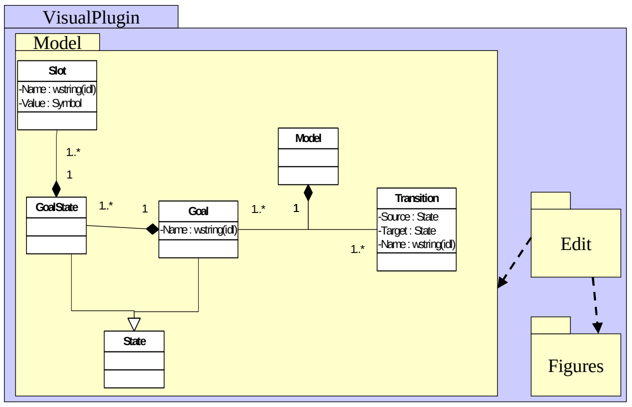


Figure 4: eclipse integration as a plug-in (Shavor et al., 2004, p. 201)

The GEF plug-in (graphical editing framework: Moore, Dean, Gerber, Wagenknecht & Vanderheyden, 2004) is a useful eclipse extension for software visualization purposes. It offers a complete framework for the implementation of additional visual editors with graph-like presentations. The framework is based on the model-view-controller design pattern (see Figure 5): Every graphical element to be displayed has a data representation. These representations are defined in the package Model. The Edit package contains implementations for handling the graphical elements and the bend- and endpoints of each connections between elements. The Edit classes display the graphical representations in the editor window with the Figures package. It contains graphical implementations for all Model elements.



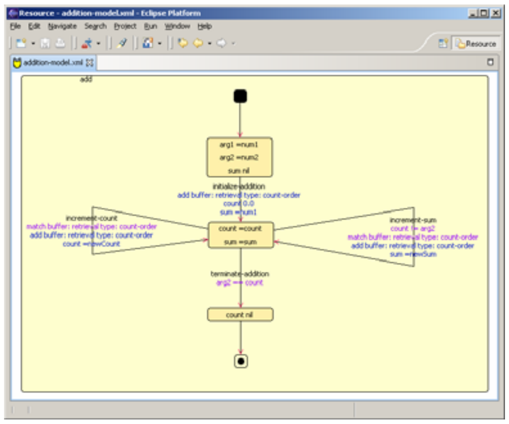

Resource - addition-model.xml - Eclipse Platform
File Edit Navigate Search Project Run Window Help
Resource
addition-model.xml
add

Figure 5: GEF conformant software design for the eclipse ACT-R visualization plug-in

Figure 6: Visualization running inside the eclipse IDE (addition model from ACT-R Tutorial Unit 1)

GEF editors work by building up a model of the objects to be displayed with the Model classes. The elements of the model are then visualized with their Figures counterparts. The interaction with the user is handled via the Edit package. All actions are reflected in the model as well as in the displayed figures. Figure 6 shows the integration of the visual editor within the eclipse environment. It is one of several possible editor views for the ACT-R model file type.

5.2 XML Representation of ACT-R/PM

The Eclipse plug-in is implemented in Java. The implementation of the visualization algorithm uses an XML format for representing ACT-R/PM models. It is easier to parse with Java using the JAXP standard package than the LISP format of the current ACT-R/PM (see Figure 7, left side). It is the same format like that of the Java implementation of ACT-R jACT-R (Harrison, 2002) (see Figure 7, right side). Thus jACT-R can be used as an GUI editor to produce well formed XML input files for the visualization algorithm. A transformation from jACT-R XML format to ACT-R-LISP format would be straight forward if the original ACT-R production system interpreter is the target system instead of jACT-R.

```
(p terminate-addition
  =goal>
    isa add
    arg1 =num1
    arg2 =num2
    count =num2
    sum =answer
==>
  =goal>
    count nil
  !output! (=num1 + =num2 is =answer))
(spp terminate-addition :p 1.0 …)
```

```
<production name="terminate-addition">
  <condition>
    <match buffer="goal" type="add">
      <slot name="arg1" equals="=num1"/>
      <slot name="arg2" equals="=num2"/>
      <slot name="count" equals="=num2"/>
      <slot name="sum" equals="=answer"/>
    </match>
  </condition>
  <action>
    <modify buffer="goal">
      <slot name="count" equals="nil"/>
    </modify>
    <output>=num1 + =num2 is =answer</output>
  </action>
  <parameters>
    <parameter      name="P" value="1.0"/>
    …
  </parameters>
</production>
```

Figure 7: ACT-R representation for production terminate-addition in LISP (original, left) and XML (jACT-R, right)

The XML format that is used has some optional extensions to the original jACT-R XML format. Information about the layout is additionally represented in the current implementation. Especially the screen positions for goal-states are stored.

6 Conclusion

The visualization for ACT-R/PM is a software engineering and modeling tool. It can be used to gain understanding of the control flow in complex cognitive models. The current implementation does only have a very simple automatic layout manager. It is necessary to rearrange a model. An 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: There will be a more sophisticated layout manager and support for exploring larger models. 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. The visualization of existing cognitive models is a first step towards a visual programming and modeling environment. An integration with the ACT-R/PM interpreter would make it an integrated editing and simulation environment.

Applied cognitive modeling will become an important engineering tool for analyzing and designing human-machine systems. This trend will clearly benefit from any software engineering support and more development tools like the visualization of ACT-R/PM models presented in this paper. An integration into a development environment would then make modeling even more efficient. Due to its powerful extension framework eclipse is a promising candidate for an integrated development platform.

References

Amant, R. St., Horton, Th. E., & Ritter, F. E. (2004). Model-based evaluation of cell phone menu interaction. *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.

Anderson, J. R. & Lebiere, C. (1998). *Atomic Components of Thought.* Hillsdale, N.J.: Erlbaum.

Clocksin, W. F. & Mellish, C. S. (1987). *Programming in Prolog.* Berlin: Springer.

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

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, *18*(8), 453-457.

DIN 66001. (1966). *Sinnbilder für Datenfluß- und Programmabläufe* [Graphical Symbols for Data and Program Flowcharts].

Gallardo, D., Burnette, E., & McGovern, R. (2003). *Eclipse in Action. A Guide for Java Developers.* Greenwich, CT: Manning.

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

Harrison, A. (2002). *jACT-R: Java ACT-R.* Carnegie Mellon University, Pittsburgh, PA: August 2-4, 2002: Paper presented at the 8th Annual ACT-R Workshop. Retrieved Feb 28, 2005 from http://act-r.psy.cmu.edu/workshops/workshop-2002/talks/AnthonyHarrison.pdf.

ISO 5807. (1985). *Information Processing - Document Symbols and Conventions for Data, Program and System Flowcharts, Program Network Charts and System Resource Charts.*

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

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).

Newell, A. (1980). Physical symbol systems. *Cognitive Science*, *4*, 135-183.

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

Post, E. L. (1943). Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, *65*, 197-268.

Price, B., Baecker, R., & Small, I. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, *4*(3), 211-266.

Ritter, F. E., Shadbolt, N. R., Elliman, D., Young, R., Gobet, F., & Baxter, G. D. (2003). *Techniques for modeling human performance in synthetic environments: A supplementary review* (State of the Art Reports). Wright-Patterson Air Force Base, OH: Human Systems Information Analysis Center (HSIAC), formerly known as the Crew System Ergonomics Information Analysis Center (CSERIAC). Retrieved Feb 28, 2005 from http://iac.dtic.mil/hsiac/S-docs/SOAR-Jun03-Front.pdf.

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.

Schoppek, W. & 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.

Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., & McCarthy, P. (2004). *Eclipse Anwendungen und Plug-Ins mit Java entwickeln* [The Java Developer's Guide to Eclipse]. München: Addison-Wesley.

Timpe, K.-P., Giesa, H., & Seifert, K. (2004). Engineering Psychology. In Charles Spielberger (Ed.), *Encyclopedia of Applied Psychology* (vol. 1) (pp. 777-786). New York: Elsevier Academic Press.

Tor, K., Haynes, S. R., Ritter, F. E., & Cohen, M. A. (2004). Categorical data displays generated from three cognitive architectures illustrate the ir behavior. In *Proceedings of the International Conference on Cognitive Modeling* (pp. 302-307). Mahwah, NJ: Lawrence Erlbaum. Retrieved Feb 28, 2005 from http://acs.ist.psu.edu/papers/torHRC04.pdf.

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