What is the Complexity of a Distributed System?

Anand Ranganathan, Roy H. Campbell
University of Illinois at Urbana-Champaign
{ranganat,rhc}@uiuc.edu

Abstract

Distributed systems are getting bigger and more complex. While the complexity of large-scale distributed systems has been acknowledged to be an important challenge, there has not been much work in defining or measuring system complexity. In order to defend against overwhelming system complexity, we need to be able to understand and measure complexity and then, attack the issues that cause complexity. In this paper, we define different aspects of system complexity and propose metrics for measuring these aspects. We also show how these aspects affect different kinds of people – viz. developers, administrators and end-users. Based on the aspects and metrics of complexity that we identify, we propose general guidelines that can help reduce the complexity of the system. Finally, we briefly describe how we have used some of these guidelines to reduce complexity in our middleware for autonomic ubiquitous computing environments.

1. Introduction

The size and complexity of distributed computing systems have been increasing inexorably in the recent past. Large-scale distributed systems such as internet systems, ubiquitous computing environments, grid systems, storage systems, enterprise systems and sensor networks often contain immense numbers of heterogeneous and mobile nodes. These systems are highly dynamic and fault-prone as well. As a result, developers find it difficult to program new applications and services for these systems; administrators find it difficult to manage and configure these complex, device-rich systems; and end-users find it difficult to use these systems to perform tasks.

System complexity has been widely identified to be an important problem[1,2]. However, the term "complexity" is often used loosely. There are no standard definitions of complexity or ways of measuring the complexity of large systems. As with so many complex things, complexity means different things to different people.

In this paper, we identify five aspects of distributed system complexity: Task-Structure Complexity, Unpredictability, Size Complexity, Chaotic Complexity and Algorithmic Complexity. We describe the causes of these different aspects and propose ways of measuring them. We also show how these aspects of complexity impact different classes of people – developers, administrators and end-users. Finally, we propose general methodologies that help in reducing the different aspects of complexity for the different classes of people.

So far, approaches to tackling system complexity have been rather ad-hoc in manner. The main contribution of this paper is in addressing the problem of system complexity in a more formal and scientific way. Section 2 describes the different aspects of complexity and metrics for measuring these aspects. Section 3 proposes general guidelines and patterns that can help tackle these facets of complexity. In section 4, we describe the trade-off between complexity and flexibility that system designers face. In Section 5, we make the distinction between the complexity of a distributed system and the complexity of using a distributed system for performing tasks. Section 6 briefly describes our approach to reduce the complexity of ubiquitous computing systems. Section 7 has related work and conclusions.

2. Different Aspects of System Complexity

The term complexity has been widely used in different contexts by different people. In general, though, system complexity can be described as a measure of how understandable a system is and how difficult it is to perform tasks in the system. A system with high complexity requires great mental or cognitive effort to comprehend and use, while a system with low complexity is easily understood and used. In this section, we attempt to capture some of the aspects of systems that make them difficult to understand.

2.1 Task-Structure Complexity

Task-Structure Complexity measures how difficult it is understand how to perform a task in a distributed system. This complexity aspect takes a graph or flowchart representation of a task and gives a measure of how complex the structure of this graph is. For developers, the task graph represents the structure of

the program. For administrators and end-users, the task graph represents the structure of the set of actions that they need to perform to achieve a goal.

In order to measure task-structure complexity, we extend a metric from software engineering called cyclomatic complexity[3]. Cyclomatic complexity measures the number of linearly independent paths through the task graph; i.e. it gives the number of possible ways of executing the task. The formula for cyclomatic complexity (CC) is:

$CC = E - N + p$

where E = the number of edges of the task graph

N = the number of nodes of the task graph

p = the number of connected components

The term 'p' is normally equal to 1 for a single process. 'p' may be more than 1 if several concurrent processes need to be undertaken to perform a task.

The other assumption in this formula is that there is only a single end goal state. So, if there are branches in the task graph, all branches finally merge into a single end goal state.

The cyclomatic complexity gives a measure of the number of decision points in the program. Decision points are those from where the task execution can proceed in different directions. Common decision points in distributed systems arise from choosing between multiple ways of performing the task, checking for exceptions and recovering from failures. A task with a number of decision points (such as one with a number of branches and loops) has a larger cyclomatic complexity than a task that follows a linear sequence of steps. For example, in Fig 1, the task graph at the left has a lower cyclomatic complexity than the one at the right. The task graph at the left has a cyclomatic complexity of CC = 9 - 9 + 1 = 1. The task graph at the right has a cyclomatic complexity of CC = 25 - 20 + 1 = 6.

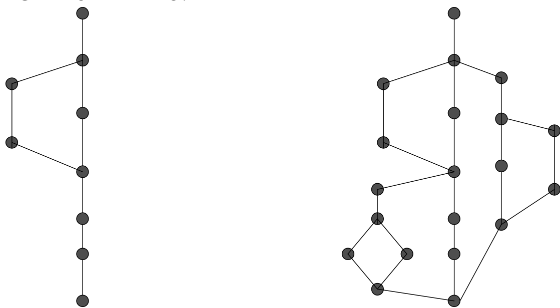

Figure 1. Task graphs with different task-structure or cyclomatic complexities

For developers, if the program they are writing has a high cyclomatic complexity, it requires greater effort for developing, testing and maintaining it. For each decision point, developers have to be aware of the various choices available and describe how the program should proceed for each choice. Cyclomatic complexity is also related to the number of test cases that have to be written for testing a program.

For administrators and end-users, a task with high structural or cyclomatic complexity requires more cognitive effort to understand and perform. Administrators and end-users may not be aware of the different choices available at decision points or what is the best choice for the current state of the distributed system and task at hand. As a simple example, when one is installing a new application in Windows with the help of a wizard, the process of installation is far simpler if the user just has to keep clicking the "next" button to go through the various steps in a linear manner, than if he has to make a number of choices during the process and he is not aware of the consequences of the different choices. Hence, decision points add to the complexity of performing a task.

2.2 Unpredictability

Unpredictability gives a measure of how difficult it is to predict the effects of an action in a distributed system. An important element that affects predictability is the amount of randomness or entropy in the system. The higher the entropy of the system, the more difficult it is to predict the state the system is in after performing an action. The entropy of a system is measured using the probability distribution of the possible states of the system. If on performing an action, the system is in one of k different states with probabilities $p_1,p_2,..p_k$, then the entropy of the system, *H*, is:

$$H = \sum_{i=1}^{k} p_i \log_2\left(\frac{1}{p_i}\right)$$

The term $log_2 (1/p_i)$ is often called the surprisal factor. Surprisal is the degree to which one is surprised to see a result. If the probability of an event is 1, there is zero surprise at seeing the result. As the probability gets smaller and smaller, the surprisal goes up. Hence, if the system may only be in a small number of states, each with relatively high probability, then the entropy is low and one is unlikely to be surprised very often. But if the system can be in a large number of rare states, then the entropy or unpredictability is high.

Unpredictability in distributed systems often results from dynamism, failures and race conditions. If an action performed by a service or application is

unpredictable, then it becomes difficult to test and maintain it for developers. Automation may also create problems of unpredictability for end-users.

There are two ways in which unpredictability of a system can be reduced. One is by reducing the number of states that the system can be in. The other is by increasing the probability of a few "desired" states and reducing the probability of other states.

2.3 Size Complexity

Another measure of system complexity is the size of the distributed system. Traditionally, the size of a distributed system is measured by the number of nodes, devices, services, applications or other components. In addition, a distributed system may have high cognitive complexity if users need to be aware of a large number of concepts in order to use the system. A concept is any logical item of knowledge defined or used by the system. A concept includes abstract notions like file-types, security policies, context information, device characteristics and QoS parameters. A large number of concepts contributes to greater difficulty in understanding the system as a whole. Hence, the size of the body of knowledge required to develop applications for the system, manage the system or use the system to perform tasks is an important measure of complexity.

2.4 Chaotic Complexity

Chaotic Complexity refers to the property of systems by which small variations in a certain part of the system can have large effects on overall system behavior. Chaotic complexity makes it difficult to understand systems. It often results from a lack of modular design and from a number of inter-dependencies between different parts of the distributed system.

As an example, policies are often a source of chaos in a system. Policies, such as access control policies, often have the power to affect different parts of the system, especially because many distributed systems do not have ways of checking the consistency of different policies. Hence, it is often fairly easy to write policies that cause unexpected behaviors. For example, it may be easy to write a policy that denies access to all resources for all people, accidentally.

An important factor contributing to chaotic complexity is coupling between different components. There are different kinds of couplings[5]. Some of these couplings (in order of increasing complexity) are:

1. Components are data coupled if they pass data through scalar or array parameters.
2. Components are control coupled if one passes a value that is used to control the internal logic of the other.
3. Components are common coupled if they refer to the same global data.
4. Components are content coupled if they access and change each other's internal data state or procedural state.

There are many reasons why low coupling between components or modules is desirable[4]. Fewer interconnections between components reduce the chance that a fault in one component will cause a failure in other components. Also, fewer interconnections reduce the chance that changes in one component will affect other components, which enhances reusability. Finally, fewer interconnections reduce administrator and programmer time in understanding the details of the system.

One way of measuring coupling between components is fan in - fan out complexity[12]. This measure maintains a count of the number of data flows into and out of a component plus the number of global data structures that the component updates. The fan in - fan out complexity of a component is given by the formula:

$$\text{Complexity} = \text{Length} * (\text{Fan-in} * \text{Fan-out})^2$$

Length is any measure of length such as lines of code.

However, coupling is just one of the factors that could lead to chaos. We are still in the process of investigating other factors that could cause small variations in one part of the system to lead to large variations in system behavior.

There is a subtle difference between chaotic complexity and unpredictability. Chaos is deterministic – i.e. it is possible to say, deterministically, what would be the overall change in system behavior on changing any part of the system, given enough knowledge about the architecture and functioning of the system. Unpredictability, however, is intrinsically non-deterministic (or probabilistic). It covers aspects that cannot be predicted deterministically, and that may occur due to random errors or race conditions, or due to insufficient knowledge about the workings of the system.

2.5 Algorithmic Complexity

The traditional definition of the complexity of an algorithm is in terms of its time and space

requirements or its relation to Turing machines or universal computers. However, there is also a cognitive aspect to algorithmic complexity, which is the effort required to understand an algorithm. There is often a trade-off between the performance and cognitive aspects of algorithmic complexity. Simple algorithms are often brute-force in nature and may have high space or time complexity. However, more sophisticated algorithms that reduce the space or time complexity have a high cognitive complexity. A simple example is the use of an $O(n^2)$ algorithm for sorting (like insertion-sort or bubble-sort) as opposed to an $O(n.\log n)$ algorithm (like quick-sort).

The cognitive algorithmic complexity can be measured using Halstead's measures[11]. These measures principally estimate the programming effort, but can be extended to measure the effort required to understand an algorithm. The Halstead measures are based on four scalar numbers derived directly from a program's source code:

n_1 = the number of distinct operators
n_2 = the number of distinct operands
N_1 = the total number of operators
N_2 = the total number of operands

From these numbers, various complexity measures are derived:

Program length (N) = $N_1 + N_2$
Program vocabulary (n) = $n_1 + n_2$
Program Volume (V) = $N * (\log_2 n)$. The program volume measures the information content of a program or the size of implementation of an algorithm.

Difficulty (D) = $(n_1/2) * (N_2/n_2)$. The difficulty of a program is also related to the error-proneness of the program.

Effort (E) = $D * V$. The effort to implement or understand a program is proportional to the volume and to the difficulty level of the program.

This measure of complexity is more relevant for programmers and administrators who write programs and scripts for performing different kinds of tasks.

Halstead's measures are related to the amount of information contained in a program. Another metric that measures the information content of any object is Kolmogorov complexity. Kolmogorov complexity is the minimum number of bits into which a string can be compressed without losing information. This is defined with respect to a fixed, but universal decompression scheme, given by a universal Turing machine. Another way of looking at it is that the Kolmogorov complexity of an object is the length of the shortest computer program that can reproduce the object.

It is a bit more difficult to measure the Kolmogorov complexity of any program since one has to devise a Turing Machine that can generate this program. The Halstead's measures, however, offer an easier way of measuring the information content of a program.

An important point about Halstead's measures is that in order to measure them, one has to decide what constitutes the set of operators and operands in a program. Commonly occurring blocks of code (like iterating through the elements of a list) may be considered as a single operator (iteration) and a single operand (the list), although the actual bloc of code may have many more operators (such as the counter or iterator) and operands (such as incrementing the counter, checking for end of list, etc.). Thus, the choice of the set of operators and operands may be made depending on the skill of the programmer.

2.6 Why these metrics?

There has not been much work in studying the intrinsic complexity of using distributed systems from the point of view of developers, administrators and end-users. Most existing measures are either inadequate or flawed.

In the case complexity for developers, complexity is normally measured in terms of metrics like lines of code or development time. However, these metrics have a number of limitations since they depend heavily on coding language, coding styles and developer skill. Our view is that the complexity metrics we have proposed capture the difficulty of building services and applications better than the normal metrics.

Also, some of our metrics have been used in the software engineering field for a while. Hence, there is a common consensus on what values of some of the metrics like Cyclomatic Complexity and Halstead's Measures are appropriate to promote understandability, programmablity and maintainability of programs. For example, the Halstead volume (V) of a function is recommended to be between 20 and 1000. Volumes greater than 1000 means function is doing too much. The Volume of a file should be between 100 and 8000. Also, the Cyclomatic Complexity of a function is recommended to be less than 15.

The other aspects, including unpredictability and cognitive size complexity, represent important features that make systems difficult to program and use.

However, there are no well-accepted metrics for evaluating these aspects. Hence, we propose our own metrics, including entropy and number of concepts, to measure these aspects of complexity. So, although there are no standard guidelines to define appropriate values for these metrics, one can still compare different systems using these measures.

For administrators and end-users, too, there are no standard ways of measuring the complexity of managing and using systems. One metric that is sometimes used in HCI is GOMS[15], which allows calculating the time it takes for achieving a goal with a certain user interface. However, this metric does not really capture the complexity of using the interface. As we have described earlier, many of the aspects that cause complexity for developers also cause complexity for administrators and end-users. Hence, it is possible to extend the software-engineering-based and other metrics for administrators and end-users as well.

3. Attacking complexity

In this section, we propose various guidelines that can help reduce the different aspects of complexity. This is not meant to be an exhaustive set of solutions, but rather, examples of some of the approaches that can be taken to reduce system complexity.

3.1 Self-Configuration and Self-Repair

One way of reducing task-structure complexity is by building autonomic systems that can configure, optimize and repair themselves. This would allow the specification and execution of tasks to be more linear, since the system can take care of making choices at various decision points automatically. Hence, administrators and end-users do not have to worry about choosing appropriate values at decision points or what actions to take upon failures.

In distributed computing, many of the decision points that contribute to cyclomatic complexity arise from checking for exceptions and failures. If the distributed system provides middleware that performs self-configuration and self-repair, then programmers, too, do not have to deal with failures, exceptions and error conditions. Hence, many decision points in programs can be eliminated.

Self-repair also reduces unpredictability since it increases the probability of a few desired states corresponding to the successful execution of a program or successful performance of a task, while reducing the probability of failure.

3.2 High-Level Programming and Interaction

Cognitive algorithmic complexity can be reduced by using high-level programming. For this, we need frameworks or middlewares that allow developers to program using high-level, abstract operators and operands. These high-level operators and operands are resolved by the framework or the middleware to appropriate low-level functions and operands either during compile-time or during run-time. Thus, the number of operators and operands used in a program will reduce, and hence the program volume and programming effort will also reduce. This will, thus, reduce cognitive algorithmic complexity.

High-level programming, along with self-configuration and self-repair, can also help reduce unpredictability. Since developers specify their programs at a high-level, this gives the distributed system middleware more flexibility in choosing an appropriate way of executing the program and recovering from failures. Thus, the unpredictability of program execution, arising from dynamism and failures, is abstracted away from developers.

Similarly, if end-users specify their requirements at a high-level, then the distributed system can, potentially, pick appropriate ways of meeting these requirements automatically. High-level interaction allows end-users to not worry about numerous low-level details of the system. Thus, the number of possible states that the system can be in is far smaller (since the end-user views the system from a higher-level of abstraction); and the entropy and unpredictability of the system is apparently lower.

3.3 Hierarchical Organization of Systems and Concepts

Size complexity can be effectively tackled with a divide-and-conquer approach. Many large distributed systems are organized hierarchically, which allows dealing with smaller parts of the system independently. For examples, hierarchical DNS servers are used to translate between names and IP addresses in the internet. However, there has not been much work in organizing the knowledge required to develop, manage and use distributed systems hierarchically. One way of doing this is to develop ontologies that define hierarchies of concepts used in the distributed system. Ontologies are a standard way of representing domain knowledge in a reusable manner. They allow different parties to become aware of the various concepts used in the system and the

relationships between these concepts. Defining hierarchies of concepts also helps users in understanding the details of the system at a high level, while allowing them to drill down to specific details if they want to do so.

4. The Complexity-Flexibility Trade-Off

An important effect of reducing complexity through high-level programming and self-configuration and self-repair is that the flexibility of the developer is also reduced. While high-level and task-oriented programming abstracts away many of the low-level details, it is also not as expressive or flexible as lower-level programming and does not allow developers to perform certain kinds of operations. Flexibility is related to the number of different states that a language or system allows to be reached, or the number of different transitions between states offered by operators of the language or system.

The complexity-flexibility trade-off plays out at other levels of programming as well. For instance, machine-level or assembly-level programming is probably the most expressive and flexible since it allows developers to do pretty much anything allowed by the instruction set of the processor. However, they are also incredibly complex to program in. In particular, the cognitive algorithmic complexity of a program in machine-level or assembly-level is very high. Higher-level programming languages like Java are less complex, but also allow lesser flexibility. For example, developers cannot cause arrays or buffers to overflow, or write data to arbitrary memory locations. Also, applets may not open sockets to arbitrary hosts or access the local filesystem.

Fig 2 shows a possible graph representing the tradeoff between flexibility and complexity. As the complexity of a programming model and the programs in it reduces, the flexibility and expressiveness of the programs also reduces. Hence, depending on the needs and requirements of the developer, an appropriate flexibility-complexity point must be chosen. The choice needs to be made depending on how finely the developer wants to program the system and the level of control and abstraction he desires.

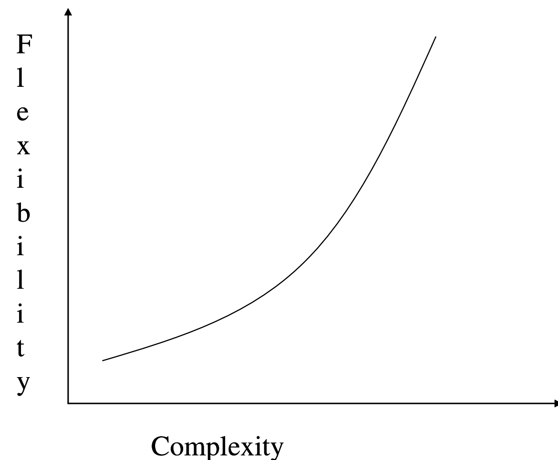

Figure 2. The Complexity-Flexibility Tradeoff

5. Distinguishing between the complexity of a system and the complexity of using a system

There is an important difference between the intrinsic complexity of a system and the complexity of using a system to perform tasks. While a system can be very complex in the inside, it may still be easy to use for performing various kinds of tasks. For example, cars are extremely complex systems internally; however, the interface exported by a car to end-users (or drivers) is fairly simple, and one can learn to drive a car reasonably easily. At the same time, cars are still very complex for developers and administrators (or car designers and car mechanics).

The main idea here is that even though a system may be very complex, internally, it is possible to hide that complexity and present relatively simple interfaces to developers, administrators and end-users. One way of doing this is through high-level programming and self-configuration and self-repair. The main challenge is to make high-level programming, self-configuration and self-repair reliable and predictable enough that users do not have to look under the hood too often. This is similar to the way in which car drivers do not worry about the internal functioning of the car most of the times. Of course, when things do go wrong, the systems must have some way of uncovering the different layers of abstraction and allow some form of debugging.

6. Reducing the complexity of ubiquitous computing systems

In our own work, we have attempted to tackle the problem of complexity of ubiquitous computing environments using these design principles. Ubiquitous computing environments feature massively distributed systems containing a large number of devices, services and applications that help end-users perform various kinds of tasks. However, these systems are very complex to configure and manage. They are highly dynamic and fault-prone. Besides, different environments have different resources and architectures and offer different ways of performing the same task.

We have developed a middleware that allows developers and administrators to program these environments using high-level, parameterized tasks. A semantic discovery process, along with a multi-dimensional utility function, is used to discover possible ways of executing a task and to pick the optimal way. The middleware can also recover from failures of one or more actions by using alternative resources or strategies to perform the task. The middleware thus allows self-configuration, self-optimization and self-repair.

Developers program the environment using a high-level programming model called Olympus[6]. The main feature of this model is that it allows certain ubiquitous computing operators and operands to be described at a high level. The middleware takes care of mapping them to appropriate low-level operators and operands depending on constraints specified by the developer and the current state and context of the system. Administrators can configure how tasks are performed by modifying high-level task scripts and task graphs as well as specifying declarative polices in Prolog that constrain the ways in which tasks can be executed. End-users interact with the environment through a Task Control GUI, where they can specify the tasks they want to perform and then specify high-level parameters that influence task execution.

The middleware uses definitions of different concepts in ontologies. Ontologies define hierarchies of different services, devices, applications, contexts, data types and other concepts. Developers, administrators and end-users use the concepts defined in the ontology while developing applications and configuring the system. This middleware runs on top of Gaia[7], our infrastructure for ubiquitous computing.

7. Related Work and Conclusions

So far, there has not been much work in defining or measuring the complexity of distributed systems. There has, however, been a lot of related work in the field of autonomic computing for tackling system complexity. Various systems such as [8,9,10] propose strategies to enable self-configuration, self-optimization and self-repair. Other work[13,14] simplify end-user interaction in complex ubiquitous computing environment by representing user tasks at a high-level and determining ways of performing these tasks at runtime. While these different systems, including ours, have their own merits, it is difficult to evaluate how well they do in reducing the complexity for developers, administrators and end-users. We hope that our proposed metrics will partly help alleviate that problem.

In conclusion, we have tried to formalize the notion of complexity of distributed systems. We have proposed a number of metrics to measure different aspects of system complexity. We believe that our work is a first step towards building a set of metrics for comparing different distributed systems, middlewares and programming frameworks in terms of complexity for all parties involved in the system, viz. developers, administrators and end-users.

8. Acknowledgements

This research is supported by grants from the National Science Foundation, NSF CCR 0086094 ITR and NSF 99-72884 EQ.

References

[1] W. Asprey, et al., “Conquer System Complexity: Build Systems with Billions of Parts,” in CRA Conference on Grand Research Challenges in Computer Science and Engineering, Warrenton, VA (2002), pp. 29-33,

[2] J. Kephart and D. Chess, “The Vision of Autonomic Computing,” IEEE Computer, Vol. 36, No. 1 (2003), pp. 41-50

[3] T.J. McCabe and C.W. Butler,. "Design Complexity Measurement and Testing." Communications of the ACM 32, 12 (December 1989): 1415-1425

[4] M. Page-Jones. The Practical Guide to Structured Systems Design. Yourdon Press, New York, NY, 1980.

[5] G. Myers. Reliable Software Through Composite Design. Mason and Lipscomb Publishers, New York, NY, 1974.

[6] A. Ranganathan, et al, “Olympus: A High-Level Programming Model for Pervasive Computing Environments,” in IEEE PerCom 2005, Kauai Island, Hawaii, 2005
[7] M. Roman, et al, "Gaia: A Middleware Infrastructure to Enable Active Spaces," IEEE Pervasive Computing Magazine, vol. 1, pp. 74-83, 2002.
[8] H. Liu et al “A Component-Based Programming Model for Autonomic Applications”. in ICAC 2004, New York, NY
[9] D.M.Chess et al “Unity: Experiences with a Prototype Autonomic Computing System” in ICAC2004, New York, NY
[10] Ponnekanti, S.R. et al. “Portability, Extensibility and Robustness in iROS”, in IEEE PerCom 2003.
[11] M. Halstead. Elements of Software Science, Operating, and Programming Systems Series Volume 7, Elsevier, 1977.
[12] S. Henry and D. Kafura. Software structure metrics based on information flow. IEEE Transactions on Software Engineering, 7(5):510--518, Sept. 1981
[13] J. P. Sousa, D. Garlan, “Beyond Desktop Management: Scaling Task Management in Space and Time” Technical Report, CMU-CS-04-160, Carnegie Mellon University
[14] U. Saif, et al “A Case for Goal-oriented Programming Semantics”. In System Support for Ubiquitous Computing Workshop at Ubicomp 2003, Seattle, WA, Oct 12, 2003
[15] Kieras, D.E. (1988). Towards a practical GOMS model methodology for user interface design. In M. Helander (Ed.), Handbook of Human-Computer Interaction (pp. 135-158). Amsterdam: North-Holland Elsevier.