An Improved Model of Configuration Complexity

Moheeb Alwarsh
Department of Computer Science, Kent State University, Kent, OHIO, USA
malwarsh@kent.edu

Abstract - *Studies have been conducted on system configurations to find metrics to calculate configuration complexity. These studies help in quantifying the complexity, estimating the required manpower and the related cost for implementing these configurations. However, a problem arises when these metrics are not precise enough to cover all aspects of system configuration. This paper is a continuation of work done by others [1][2]; we develop more precise metrics for calculating configuration complexity.*

Keywords: Configuration Complexity, Complexity Metrics

1 Introduction

Companies pay tremendous attention to data centers due to their strategic importance in the success of the companies. Operational costs for system configuration and installation in these centers is increasing day after day, and in some cases, these costs exceed those of the hardware and software [5]. Researchers are working on ways to decrease the operational costs or at least, to stabilize them from further future increases. Studies have been implemented in this area to reduce operational activities with self-healing benchmark or to develop a model for configuration complexity like [7] and [2], but the problem of determining the human costs when planning for new development is still difficult to implement without better tools and methodologies.

To understand the complexity of estimating the operational costs of configuration, we first have to know the configuration and maintenance processes used when developing a system. A system is composed of several components connected together to form an interface for a particular service. As an example, to install and configure a proxy server to provide Internet service, we need to install and configure one or more operating systems like Linux, a firewall, a Dynamic Name System (DNS), a Network Time Protocol (NTP), the proxy application, Lightweight Directory Access Protocol (LDAP) for accountability, and a load balance application or appliance to distribute user loads on the system. The human cost of implementing such a system depends on how many persons are needed and for how long. Of course, configuration complexity varies from system to system based on the time of installation and configuration. Some systems can be installed in few steps and a short time, but others need a tremendous effort to be implemented. An expert administrator can finish the job faster than a novice administrator. There is a need to develop and implement an algorithm that measures configuration complexity and that fits any expertise of manpower.

This paper is a continuation of work done by others [1][2] to quantify configuration complexity. In addition, others have proposed a mechanism to translate the result of quantification to a measure used to estimate the cost of manpower. This estimation is not precise and it could produce unexpected results. We propose improvements in the way the configuration complexity is quantified to provide a more precise mechanism.

The remaining parts of this paper are organized as follows. Section two presents the Complexity Quantification method used in [2]. Section three discusses the improved model of quantifying Configuration Complexity. The fourth Section is Related Work and the last will include the summary and concluding remarks.

2 Related Work

Ad hoc configuration is the main cause of many configuration issues, and ad hoc configuration also makes the fixing of problems discovered late more complex. This affects the effort in completing the configuration process and increases the implementation time which subsequently increases the cost [3]. Incorrect configurations cause around 90% of these problems [6]. Therefore, planning a configuration document that includes all entered parameters is necessary to help avoid these costs.

In [1], three distinct types in the configuration process lifecycle are identified. The first is an initial configuration where performance is not considered until the end of the process. The second configuration type is when the performance of a running system decreases, and in this case a configuration is needed to put it back on the track. The last type of configuration is implemented when a new performance level is required. However, an implemented configuration without a validation check would increase the complexity of debugging the system. Validation needs to be included in the configuration process lifecycle.

Configuration complexity metrics have been introduced in [2], and they are classified into three areas. They are execution complexity, parameter complexity and finally memory complexity which is human memory. [2] introduces an approach to measure the complexity; high complexity increases the probability of getting a defected system. Implementing a configuration which depends on measuring the system performance at the end of the configuration process [1] might increase the complexity of finding and fixing errors when more debugging time is required to fix problems, and this time may exceed the time needed to re-configure the entire system. Others [4] have built their cost prediction on these metrics, and this could provide a less precise estimation because problems that might be encountered after the configuration process are not considered in calculating the costs. In addition, increased memory complexity could mean an increase in human error during the configuration process by a human operator.

We introduce an improved model that uses two configuration complexity metrics from the [2] and one new category for validating the configuration. Our model provides more precise metrics to quantify the configuration complexity.

3 Complexity Quantification

In [2], the configuration complexity measure is based on three components. These components are collected from a manual configuration of an e-commerce solution; see Figure 1. The first component is execution complexity, and it consists of two metrics. The first metric is the number of actions for the configuration procedures. Borrowing an example from [2], to configure an e-commerce system (see Figure 1), we need to perform 59 human actions. The second metric is the context switch metric which increments when a user temporarily stops configuring one component and switches to configure another component.

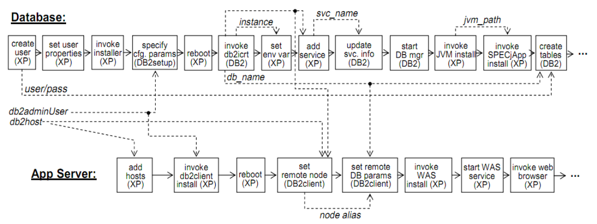

Figure 1. Partial Manual Configuration and Action Sequence for Installing e-Commerce System [2]

The second component of configuration complexity is the parameter complexity. This component consists of five metrics.

Parameters Count: This is the total number of parameters involved in the installation and configuration procedure.

Parameters Use Count: This is the total number of times the parameters are used in the procedures. For example if we use one parameter in three locations, then our Parameters Use Count value is 3.

Parameters Cross Context: If we use a parameter in configuring one component and use the same parameter in configuring another component, then we have a total of 2 Parameters Cross Contexts.

Parameters Adapting Count: This is the total number of parameters used in one form then adapted to be used in other forms. An example is the fully-qualified path name where the path name changes based on source location. Assume we have a directory /a/b/c/d as the target directory. Then if a source resides in "a" directory, the path will be "/b/c/d". If a source resides in "c" then the path is "/d". The Parameters Adapting Count in this case is 2.

Parameter Source Score: Each parameter is assigned a score from 0 to 6 based on how difficult it is to obtain the parameter. For example, a parameter that would be obtained as a result of executing several commands is more difficult than a parameter that could be entered directly like user name.

The third component is memory complexity, and it refers to the number of parameters a human operator must memorize or remember during configuration. The memory complexity is based on three metrics: Memory Size, Memory Depth and Memory Latency with the value of each being an average. The configuration approach in [2] assumes that a system administrator already knows how to configure an e-commerce solution. Memory Size means the remaining parameters which a system administrator needs to memorize and is required to use for each step in the configuration. For example, Figure 1 shows the process of configuration. At the first step a system administrator will memorize the created user and profile location. These parameters will be used later in the configuration. In this case the memory size is 2. Storing parameters is based on Last-In-First-Out LIFO with non-associative lockup. Memory size, which is the size of the stack that has all memorized parameters needed for current or future configuration, is captured prior to each configuration action. The Memory Depth is the process of measuring the depth in the stack for a targeted parameter. For example, if we have a stack of 10 parameters and the order of the required parameter is 5; then, memory depth at this stage is 5. Memory Latency is calculated based on the time between storing a parameter and using it. For example, if we use "user name" in Figure 1 at the end of the installation, then the Memory Latency will be the total time between storing the user name and using it. Since values are fluctuating up and down, only the maximum and average of each metric will be calculated. Figure 2 shows the metrics collected from the three memory complexity components after configuring an e-commerce solution; Figure 1.

	Measure	Value (manual procedure)
Exec	NumActions	59
	ContextSwitchSum	40
Param.	ParamCount	32
	ParamUseCount	61
	ParamCrossContext	18
	ParamAdaptCount	4
	ParamSourceScore	125
Memory	MemSizeMax	8
	MemSizeAvg	4.4
	MemDepthMax	12
	MemDepthAvg	1.5
	MemLatMax	55
	MemLatAvg	4.4

Figure 2. Configuration Complexity Metrics measures [2]

4 Improved Model for the Configuration Complexity

In section 2 we have demonstrated how to calculate the configuration complexity metrics using the procedure proposed in [2]. However, there are some concerns with the proposed method when it comes to memory complexity and the lack of a validation plan for configuration. We will discuss memory complexity proposed by [2]. Then, we will introduce our proposed validation metric and show how it could help in reducing the debugging time and the configuration complexity process.

4.1 Memory Complexity

Memory complexity is not applicable in all scenarios when configuring a new or an existing system. For example, after planning and designing a phase of a new system, a system administrator should write down all successful steps required to install and configure the new system. Reproducing the same steps is necessary for automation, validation and quality assurance. Configuring a complex system might require several hours or days in collaboration with other system administrators. Depending on a human operator's memory to install and configure a new system is very risky and might endanger the entire process of installation and configuration. An example is when a wrong parameter is selected or entered without discovering it until the end of the configuration process. Memory complexity is not a precise metric that should be used in all configuration scenarios. Using memory complexity with large systems would increase the probability of human errors. There is a need to document the necessary steps for configuring a new system or reconfiguring an existing system to facilitate revision of the implementation process, automation and validation.

4.2 Validation Complexity

A complex system might consist of components that have a high degree of context switching during configuration similar to Figure 1 or a low degree of context switching similar to Figure 3. However, the complexity of validating the functionality of each component increases after building the entire system. If no testing for validation of functionality is implemented right after completing the configuration of each component for a complex system whenever this is possible, then the time of debugging and verifying could increase significantly; in fact, it might exceed the time of configuration. An example is when debugging errors of a node in a Rocks cluster. Reinstalling the node could be implemented in one or two steps but debugging the errors might take more steps. This would increase the operational time and, therefore, would likely increase the cost.

Let V be a set of validation procedures that are not part of system components where V = {V1, V2, V3...Vn}; let SC be a system's components where V ϵ SC; and let S be a system where S = {SC1, SC2, SC3...SCm}. Then, when the system is complete, the potential complexity of validating the functionality of the entire system is much higher if no incremental validation of each system component has been implemented. For example, let us assume that we have a procedure of three steps to validate the functionality of a system component and we have five system components. If we build the system without putting any stop points to check what has been implemented, then we might end up with a defected system that needs to be debugged in order to find the cause of the error.

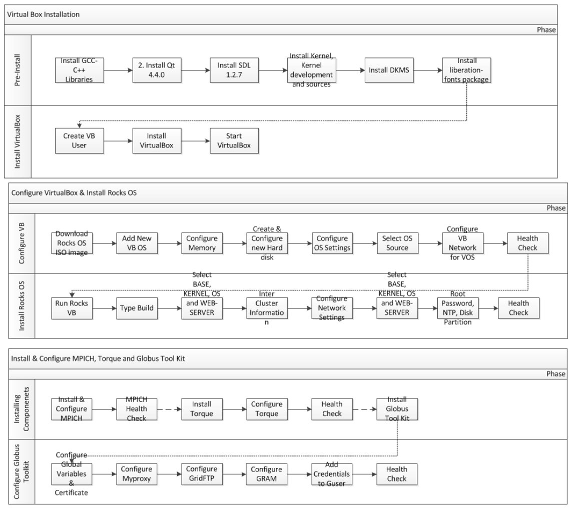

Figure 3. A semi sequential process for installing Grid on top of a virtual cluster.

We assume each system component is implemented as a black box that works independently. A system administrator combines some system components to provide a working solution. For example, a company may want to regulate Internet access for its employees and at the same time provide a layer of protection. To implement this solution, the company would need a proxy application that works as software under an operating system or in an appliance as an independent solution. The proxy needs a Domain Name System (DNS), firewall, a Network Time Protocol (NTP) and a Lightweight Directory Access Protocol (LDAP) or any other user accountability solution. Once we configure all these system components and start the system, it might work and it might not. For the latter case, we have to check each system component and make sure it is working well. For example, we can start with the DNS by issuing this command (nslookup www.domain.com). If the DNS is not working, then maybe the firewall is causing the issue or the proxy application corrupted some DNS files during installation or other system component is causing the problem. Sometimes one component might cause the problem, and in other cases the integration of two or more components might cause the problem. The system components developers didn't sit together to provide a single working solution. Instead, each one focused on providing an independent solution that could be used with other applications. In the proxy example, if we assume that the operating system is functioning without any problem, then we have five system components that need to be tested if the system fails. If we assume that each system component needs three steps to verify its functionality, then the total number of validation checks is 15 for the best case and 210 for the worst case.

If we just check the functionality of each system component, then each one will take 3 validation checks with a total of 15. There might be only one system component that causes all the issues. For example we might remove the firewall and test the system functionality. In this case we will end up with validating the remaining four system components (DNS, NTP, LDAP and the proxy Application) to make sure they are working well. This scenario might be applicable to other components, and this could produce 16 system component validations with a total of 48 validation checks. This could lead to do a total of 210 validation checks in the worst case by disassembling all system components. As an example, let's assume that there is only one system component out of 5 which is causing a problem and affecting other components. We will try to remove one by one and check all other components to make sure that a certain component is causing the problem. If we start by removing system component one {1}, then we have to test the remaining 4 components {2,3,4,5} to make sure they are free from error. If we found that the problem is still there, then we will put back {1} and remove {2}, then we will check the remaining system components {1,3,4,5}. If no is problem found we will keep this process until finding the one that is causing the problem. If system component {5} is the one that is causing the problem then it will cost 48 validation checks. What if we don't know if a combination of more than one system component is causing the problem? Then, we will end up testing all possibilities and wasting a lot of time. The calculations below show the possibility of finding the problem. First, we test the system as whole, then we might remove system components one by one or a combination of more than one.

{1, 2, 3, 4, 5} : 5 * 3 = 15

{1,2,3,4}, {1,2,3,5},{1,3,4,5}, {2,3,4,5} : 16 * 3 = 48

{1,2,3}, {1,2,4},{1,2,5},{1,3,4}, {1,4,5}, {2,3,4}, {2,3,5}, {3,4,5} : 24 * 3 = 72

{1,2}, {1,3}, {1,4}, {1,5}, {2,3}, {2,4}, {2,5}, {3,4}, {3,5}, {4, 5} : 10 * 6 = 60

{1}, {2}, {3}, {4}, {5} : 5 * 3 = 15

Total = 210 validation check

A representation of this calculation is shown on Figure 4. The worst case occurs when we encounter an error after the end of the configuration process and we start by testing system components one by one, then combining system components to figure out the combination of system components that makes this problem.

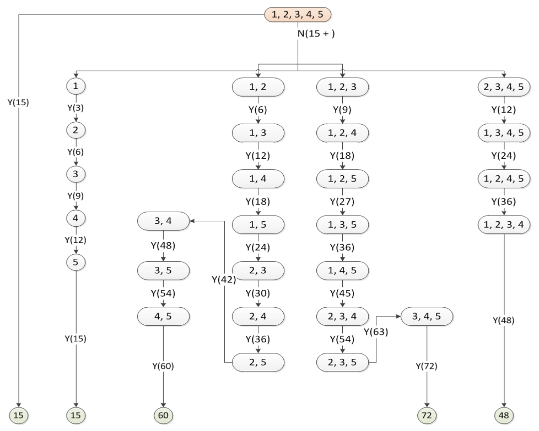

Figure 4. Probability that one or more system components are causing a problem. Worst case is when the problem occurs by combining system component {1,2, 3, 4} and we start from left of the tree to the right. In This case the worst case is 210 validation check.

We can't measure the cohesiveness in this scenario because we don't know how a system component is designed and what might affect this design. To reduce the total number of validation checks, we need to implement a stop point after assembling a system component to make sure that it is working well and doesn't cause any problems by itself and with the previously assembled components. If errors are found with a newly configured system component, then a backward validation starts building from the last SC toward the first one to find if there is any conflict. By following this procedure, we will reduce the number of validation check to the minimum possible validation check. Let's use the previous example that we illustrated early regarding finding a defective system component out of 5. We will start by configuring SC {1} and implement a validation check. If no problem found, then we will move forward to configure SC {2} and implement validation check. We continue to add SC {3}, {4} and then {5} where we discover the problem. In this case the total validation is 15 compared to 48 with the previous example. If for some reason, we found an error during the configuration process, then we implement a backward validation check between the newly added SC and the previous SC. In this case we would get 15 as a best case and 54 as a worst case compared with 210. See Figure 5.

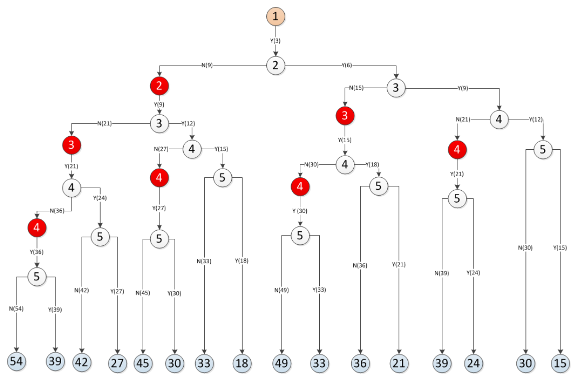

Figure 5. Probability that one or more system components are causing a problem. The minimum is 15 validation checks if there is no error. The maximum is 54, and this occurs when an error is discovered in each component which requires backward validating check.

Validation complexity can be measured with two metrics. The first one is the number of validation checks needed for each system component. The second metric is the total number of validation checks for the best case and worst case. By combining the validation complexity metrics with the execution and parameters metrics (See Table 1), we would get a more precise quantitative measure of the configuration complexity because we would be able to get more details about the problems that we might encounter during the configuration processes. Assuming the process of configuration is implemented without errors is not valid [6]. This would allow us to estimate a more accurate time for the configuration process and therefore its cost. Memory complexity is not a realistic metric because a configuration process for a complex system is documented for verifications, quality assurance, and to reproduce the configuration steps. Thus, the memory complexity is not normally applicable. Further, consider, for example, when configuring a complex system that has hundreds or even thousands of parameters, then documenting all processes of configuration in a sequential steps would reduce human errors, provide a mechanism for other administrators to review the process and configuration process can be shared and then implemented by more than one system administrator. For example, let us assume the following which admittedly is extreme. We want to improve the performance of the US aviation system and we have only 10 minutes to implement the new configuration. We have 2 minutes for entering 200 parameters and 3 minutes for restarting the system. If things go wrong, then we can roll back by returning all old parameters in the same sequence we replaced them and it will take 2 minutes for entering the parameters and 3 minutes for restarting. We have a window of 10 minutes to complete our reconfiguration; otherwise, a back-up copy of the system will be installed which will take more than 2 hours. During these two hours, large airplanes could not fly in the US. We can see that depending on a system administrator's memory for this task is not really an option. Also, it is too risky to waste 2 hours of no fly zone which will cost a lot of money. Memory complexity is not an option in this scenario and the entire configuration steps need to be documented from the start to the finish before the implementation.

Execution Complexity	Parameter Complexity	Validation Complexity
Number of Actions	Parameters Count	Number of validation for each component
Context Switch	Use Count	Total Number of Validation (worst & best case)
	Cross Context	
	Adapting Count	
	Parameters Score	

Table 1. Improved Model of Configuration Complexity

5 Concluding Remarks

We propose a new and improved incremental validation metric for quantifying configuration complexity. This metric is based on the number of validation checks for each component and the total number of validation checks. Also, we introduce an improved model to increase the precision of the output. We removed memory complexity metrics from [2] and add validation complexity; this improved model is based on work done by others. We have shown that a better approach, which requires a validation check for all components after each configuration process to make sure all components are working well before moving further and configure a new component, would decrease the debugging time and reduce the complexity of the configuration process. This would help in providing a more precise estimate of manpower cost. The validation metric consists of two parts. The first part is to calculate the number of validation checks for each system component. The second part is to calculate the probability of best and worst case scenarios for implementing the configuration process.

Acknowledgments

Thanks to Dr. Austin Melton for his inputs, thoughts and valuable suggestions in this paper.

6 References

[1] Aaron B. Brown, J.L. Hellerstein. "An approach to benchmarking configuration complexity". In Proc. ACM SIGOPS European Workshop 2004.

[2] Aaron B. Brown, Alexander Keller, Joseph L. Hellerstein. "A model of configuration complexity and its application to a change management system". Integrated Network Management 2005: 631-644.

[3] Afzal, Uzma. Hyder, S. I. "Configuration Complexity: A Layered based Configuration Repository Architecture for conflicts identification". GJCST 2010: 66-71.

[4] Yixin Diao, Alexander Keller, Sujay S. Parekh, Vladislav V. Marinov. "Predicting Labor Cost through IT Management Complexity Metrics". Integrated Network Management 2007: 274-283.

[5] T. Eilam, M. Kalantar, A. V. Konstantinou, G. Pacifici, J. Pershing, A. Agrawal. "Managing the configuration complexity of distributed applications in Internet data centers". IEEE Communications Magazine 2006:166-177.

[6] Kalapriya Kannan, Nanjangud C. Narendra, Lakshmish Ramaswamy. "Managing Configuration Complexity during Deployment and Maintenance of SOA Solutions". IEEE International Conference on Services Computing (SCC) 2009:152-159.

[7] Aaron B. Brown, Charlie Redlin. "Measuring the Effectiveness of Self-Healing Autonomic Systems". ICAC 2005: 328-329.