A Model of Configuration Complexity and its Application to a Change Management System

Aaron B. Brown, Alexander Keller, Joseph L. Hellerstein
IBM T. J. Watson Research Center
P.O. Box 704,Yorktown Heights, NY 10598 USA
{abbrown, alexk, hellers}@us.ibm.com

Abstract

The complexity of configuring computing systems is a major impediment to the adoption of new information technology (IT) products and greatly increases the cost of IT services. This paper develops a model of configuration complexity and demonstrates its value for a change management system. The model represents systems as a set of nested containers with configuration controls. From this representation, we derive various metrics that indicate configuration complexity, including execution complexity, parameter complexity, and memory complexity. We apply this model to a J2EE-based enterprise application and its associated middleware stack to assess the complexity of the manual configuration process for this application. We then show how an automated change management system can greatly reduce configuration complexity.

Keywords

Configuration Complexity, Complexity Metrics, Change Management

1. Introduction

A significant fraction of modern systems management revolves around *configuration*, a process whereby individual components are assembled and adjusted to construct a working solution. Human-driven configuration procedures occur at every stage in the lifecycle of hardware and software artifacts, from installation and deployment of new system components, to ongoing maintenance and tuning of existing components, to upgrading, replacing, or decommissioning obsolete components. For many systems, these human-driven configuration procedures represent a significant operational cost, often dominating total cost of ownership [1].

One approach to reducing the human cost of configuration is through automation, as in IBM's Autonomic Computing initiative [2]; another is to improve the tools used by human system administrators [3] [4]. A specific example of an automation-based approach is the CHAMPS system [5], which formulates Change Management (comprising the provisioning, deployment, installation and configuration steps) as an optimization problem and automatically generates—based on administrator-defined policies—change management workflows from deployment descriptors. Subsequently, these

workflows are executed by a common-off-the-shelf provisioning system [6].

In order to effectively evaluate the benefits of systems like CHAMPS—and indeed, in order to target future work to the types of configuration problems that contribute most to management complexity and cost—we need a set of metrics for quantifying the complexity and human cost of carrying out configuration tasks. This paper proposes such measures, and demonstrates their use through a quantitative validation of the reduction in configuration complexity offered by the CHAMPS system. Our case study is based on the scenario of installing and configuring a multi-machine deployment of a J2EE-based enterprise application and its supporting middleware software, including IBM's HTTP Server, WebSphere Application Server (WAS), WebSphere MQ embedded messaging, and DB2 UDB Database Server and runtime client. The specific application we use is taken from the SPECjAppServer enterprise application performance benchmark [7]. It is a complex, multi-tiered on-line e-Commerce application that emulates an automobile manufacturing company and its associated dealerships. SPECjAppServer comprises typical manufacturing, supply chain and inventory applications that are implemented with web, EJB, messaging, and database tiers.

The remainder of this paper is organized as follows. Section 2 develops a model of configuration activity and illustrates it in the context of our SPECjAppServer case study scenario. Drawing on this model, Section 3 defines a set of measures for human-perceived configuration complexity, and applies them to the case study scenario. Section 4 studies how configuration complexity can be reduced by employing automation. We discuss related work in Section 5, and conclude in Section 6.

2. A model of configuration

We begin by developing an abstract model of configuration designed to capture the manual aspects of human-driven configuration procedures. Our model includes two parts: a system model that abstracts the system being configured, and an abstraction of the activity through which configuration is performed.

We will describe our model using the running example of the SPECjAppServer scenario. As described above, SPECjAppServer emulates a multi-tier e-commerce application; it consists of a set of EJBs, databases, message queues, and web front-ends. Our configuration scenario involves installing the SPECjAppServer application and all of the supporting middleware software on a two-machine system.

2.1 System model

We model a system as a set of *containers*, each of which has an associated set of *configuration controls*. A container is either a hosting environment for other containers, or a representation of a system resource. For example, in our SPECjAppServer scenario, depicted in Figure 1, there are four levels of nested containers. One set of these nested containers represents the application server with the following nesting from outside in:

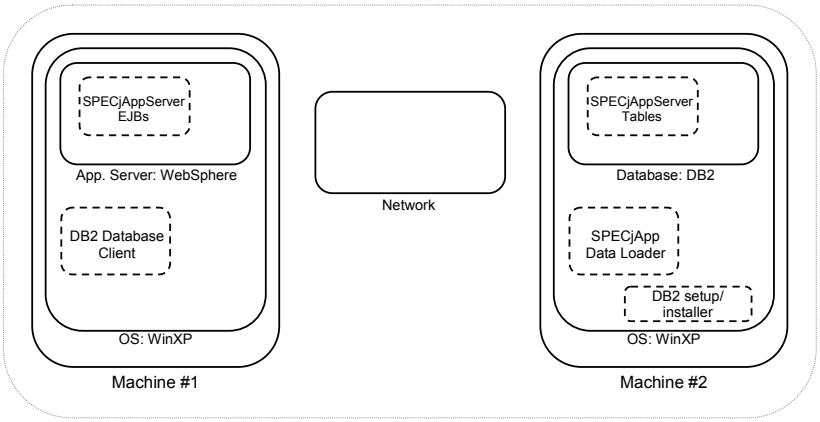

Figure 1: SPECjAppServer scenario. The figure shows the SPECjAppServer configuration scenario represented in our abstract system model of nested containers and configuration controls. Containers with solid lines are hosting environments; those with dashed lines represent system resources.

1. the server platform (Machine), which can host operating systems;
2. the operating system (OS), which can host applications;
3. the application server (App Server), which can host J2EE applications;
4. the SPECjAppServer J2EE application (EJBs).

Other sets of nested containers exist as well, resulting in a forest of trees. So far, we have not found the need for non-hierarchical container structures, but they may be required to completely model more complex systems. Since our SPECjAppServer scenario is a multi-system environment, we also have additional sets of containers representing the database server and the network (e.g., a Virtual LAN) connecting the two machines. Note that containers can be created and destroyed dynamically as the system operates or as configuration actions are performed on the system.

Each container provides a set of configuration controls. A configuration control is either an adjustable property or an invokeable method. For example, a database manager container might include configuration properties like the database's buffer pool size or access rights, and methods providing functionality like running statistics on the database or taking a backup. The configuration controls of hosting-environment–type containers include methods to create and destroy hosted containers. As an example, an operating system container provides methods to start applications and kill processes; in our SPECjAppServer scenario, the application server container has methods to install and activate J2EE applications.

2.2 Activity model

Our model of configuration activity is based on three concepts: configuration *goals*, *procedures*, and *actions*. A configuration goal describes a desired configuration state or a desired set of configuration changes. Examples of goals include low-level re-

quests for change such as “upgrade the database software on machine *m1* from version 5.0 to version 6.2”, as well as high-level goals like “decrease the average service time of application requests”. We do not constrain the level at which the goal is specified, nor do we attempt to model the process of formulating it. For our SPECjAppServer scenario, the configuration goal is “Install and configure the SPECjAppServer application as well as all needed supporting middleware software on two available machines.”

Configuration *procedures* are the means by which configuration goals are achieved. Configuration procedures are sequences of configuration *actions* performed by human operators. As we define them, configuration procedures are deterministic sequences, with no branches or decision points. Consequently, there may be many different procedures all of which achieve the same configuration goal in different contexts. The configuration *actions* that make up each procedure are manipulations of the configuration controls on containers in the system model. Different systems may implement actions in different ways—for example, setting a property on a Windows OS container may be done through a GUI console whereas the same property on a Linux OS container is set via editing a text file—but the activity model abstracts both into the same configuration action.

The level of abstraction is a key philosophical decision in our approach, and one that we hope to validate in the future through further experimentation. Our focus is on gauging procedure complexity for qualified system managers—rather than automating procedures or evaluating novices’ reactions to different user interfaces—and thus we abstract heavily to keep the model focused at the procedure level. This abstraction is also crucial in keeping the model system-independent: the same abstracted actions may have different costs (in terms of time) on different systems, but their contribution to the perceived complexity of the overall procedure should remain the same. The next section shows how we achieve this system-independence by defining configuration complexity measures in terms of how an action relates to the overall procedure, rather than the specific implementation of an action.

Figure 2 gives an example of a manual configuration procedure and action sequence for achieving the SPECjAppServer installation goal in our running scenario. This configuration procedure is designed for a specific hardware/software platform and uses a version of the SPECjAppServer application that has already been compiled and pre-packaged into the J2EE Enterprise Archive (EAR) format needed for deployment into IBM’s WebSphere Application Server (WAS). It also assumes that the target machines have only the Windows XP operating system installed, and not the required DB2 8.1 and WAS 5.1 middleware components. The procedure performs only the basic steps needed to get SPECjAppServer running and does not include any performance tuning configuration actions.

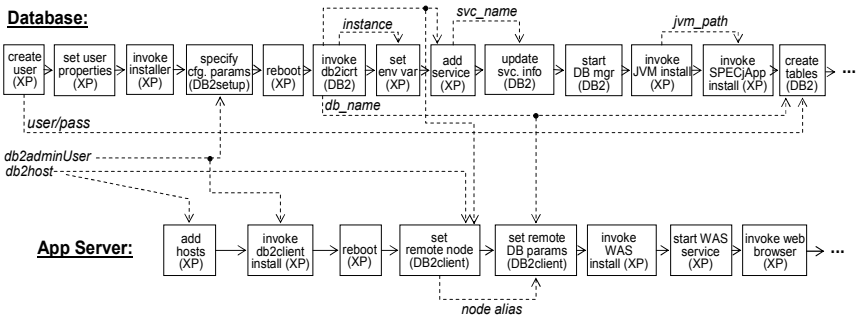

Figure 2: Partial manual configuration procedure and action sequence for SPECjAppServer scenario. This figure shows part of a manual configuration procedure for installing SPECjAppServer on a two-machine system using IBM's WebSphere Application Server v5.1 and DB2 Universal Database v8.1. Each box represents a configuration action, and the dotted arrows represent the flow of configuration parameters between actions. Each action applies to a specific container, indicated in parentheses.

3. Measures of configuration complexity

The configuration model introduced in Section 2 is a system-independent framework for describing configuration, and thus provides the basis for measures of configuration complexity that are directly comparable across different systems. We define configuration complexity as the *complexity of carrying out a configuration procedure as perceived by a human system manager*. For now, we do not consider the complexity of translating configuration goals into configuration procedures since this is (hopefully) done much less frequently than following a configuration procedure. Goal mapping is an opportunity for future research.

The three major components in our model of configuration complexity are execution complexity, parameter complexity, and memory complexity; these are analogous to the traditional software complexity categories of control flow, data flow, and space complexity, respectively [8]. In the following sections, we will define two execution complexity measures, five parameter complexity measures, and six memory complexity measures. Table 1 summarizes these measures along with their values for the SPECjAppServer installation and configuration procedure described in Figure 2.

3.1 Execution complexity

Execution complexity covers the complexity involved in performing the configuration actions that make up the configuration procedure, assuming that the actions are known in advance and that all needed configuration parameters are known and available. We use two simple measures of execution complexity. The first, *NumActions*, counts the total number of actions in the configuration procedure, weighting all equally. A more refined metric might weight different types of actions according to human-perceived difficulty or system-specific action implementation, but would re-

	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

Table 1: Configuration complexity measures and values for SPECjAppServer scenario. The complexity values are computed for the manually-driven installation procedure shown in Figure 2.

quire a significant investment in user studies to identify appropriate weights. We have found the simple version to provide significant insight, even if incomplete.

Our second measure captures the effects of *context switches*. A context switch occurs between any two consecutive configuration actions that act upon different containers. For example, in the SPECjAppServer procedure in Figure 2, a context switch occurs between the third and fourth actions in the Database flow, when the system manager must switch from operating on the OS container (XP) to the database installer container. Context switches also occur when the system manager switches between the Database and App Server flows.

Context switches introduce complexity by forcing the human administrator to swap out mental models and 'cached' rules about how a particular container and its controls behave. The complexity impact of a context switch depends on the relationship between the two containers involved; we treat the container hierarchy like a tree (adding an implicit root node if necessary) and define the *context switch distance* as the shortest path between the two containers in that tree. Our measure, *ContextSwitchSum*, is the accumulated context switch distance across all context switches in the procedure. Note that we do not take into account the *age* of existing configuration contexts and its potential impact on context switch time; we assume an initial 'clean slate' state, since we do not yet have the technology to quantify the static complexity of an existing configuration. This is clearly an important direction for future work.

Looking at the execution complexity values in Table 1, we see that our SPECjAppServer scenario has a significant amount of execution complexity, requiring 59 steps and a total context switch distance of 40. This level of complexity is surprisingly high for what is essentially a minimal two-machine middleware stack installation for a pre-packaged J2EE application, and starts to immediately suggest why

configuration complexity is such a problem with today's systems.

3.2 Parameter complexity

Next we consider parameter complexity, which measures the complexity involved in providing configuration data to the computer system during a configuration procedure—that is, providing the parameter values used when configuration actions set properties on the system's containers. *Parameters* include configuration data supplied by the human operator to the procedure, as well as configuration data generated or surfaced by the procedure and then later re-supplied by the human operator; an example of the latter case from our SPECjAppServer scenario is an auto-generated TCP/IP port number produced during the database installations and later supplied to the application server. We define five parameter complexity measures:

- *ParamCount:* the number of parameters involved in the procedure
- *ParamUseCount:* the total # of times parameters are supplied to the procedure
- *ParamCrossContext:* the total number of times parameters are used in more than one configuration context, *e.g.*, when a parameter exposed from or supplied to one container is subsequently used on a different container, weighted by the distance between contexts.
- *ParamAdaptCount:* the total number of times parameters are used in configuration actions in a different syntactic form than when they first appeared (such as a fully-qualified pathname being adapted to a relative path)
- *ParamSourceScore:* each parameter is assigned a score from 0 – 6 based on how its value was obtained; low scores represent easily-obtained values (such as those surfaced earlier in the procedure) whereas high scores represent obscure values (such as those that must be extrapolated from the system environment—a JVM path, for instance—or chosen based on experience). This metric is the sum of the source scores across all parameters.

The values in Table 1 show that our SPECjAppServer scenario has high parameter complexity as well, particularly in the fact that a non-trivial number of parameters are adapted and used across contexts, and that many of these parameters need to be supplied at more than one point during the procedure. We believe that cross-context configuration tends to be the dominant source of complexity in many computer systems, and is a key focus of our CHAMPS automation work, detailed in section 4.

3.3 Memory complexity

The final aspect of configuration complexity is memory complexity. In this context, *memory* refers to the memory of the system manager—his or her mind—not to the data storage capacity of an IT system. Memory complexity takes into account the number of parameters that must be remembered, the length of time they must be retained in memory, and how many intervening items were stored in memory between uses of a remembered parameter. The more demands that a procedure places on a human administrator's memory (or on the memory aids that he or she uses), the more complexity it manifests. Memory complexity also captures *cross-context configuration complexity*, a key form of complexity that results when a configuration procedure

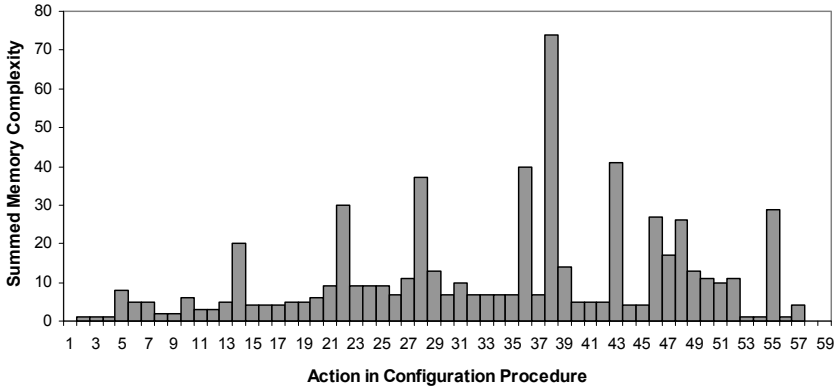

Figure 3: Per-action summed memory complexity for SPECjAppServer scenario.

includes configuration actions and parameters that appear in different contexts but are dependent on each other. For example, in our SPECjAppServer example, the database and application server must be configured to talk to each other. Configuring each end of the communication link requires configuration information from the other end (context). This information sharing creates cross-context configuration complexity; this complexity is captured by memory complexity since the shared information must be retained in memory in order to be transferred across contexts.

We use a crude model of memory as a last-in-first-out (LIFO) stack with non-associative lookup—that is, it is more difficult to remember the $(k+1)^{th}$-most-recent parameter memorized than the k^{th}-most recent. For each configuration action in a procedure, we compute the size of the memory stack just prior to the action's execution, the depth in the stack of all parameters accessed, and the latency (in number of steps) since the parameters were last accessed. Our six procedure-wide memory measures—*MemSize*, *MemDepth*, and *MemLatency*—are the averages and maximums of these per-action values across the entire procedure.

The results in Table 1 for memory complexity show that our SPECjAppServer scenario has high levels of memory complexity as well. A well-known human factors result is that people can keep about 7 items in short-term memory, so this procedure is starting to push the limits in terms of memory demands. For more detail, we can drill down into an action-by-action breakdown of memory complexity: the graph in Figure 3 shows the memory complexity for each step in the procedure, represented as the sum of the Size, Depth, and Latency measures and computed on a per-action basis. The noticeable spikes around steps #36 – 43 correspond to the time where the connection between the database and application server is being configured, the site of the greatest cross-context configuration complexity in the procedure.

4. Reducing complexity with automation

This section describes how automation can be used to reduce the complexity of installing the SPECjAppServer J2EE application. The complexity metrics introduced earlier are applied to CHAMPS, an automated change management system, and the results are compared with those for manual installation.

The automation we describe relies on an explicit representation of the dependencies between components being installed. We use the Solution Install (SI) technology to provide this explicit representation; SI is a recent technology whose goal is to facilitate the provisioning and change management of multi-tiered application systems. In SI, solutions comprise multiple levels of potentially nested *Installable Units (IU)*. Each IU has an associated *deployment descriptor (IUDD),* an XML file that describes the content of an installable unit, its checks (required system resources, prerequisites), its dependencies and its (configuration) actions [10, 11]. As an example, the overall SPECjAppServer solution, comprising the J2EE application, user data, and the underlying middleware would be described as a solution module, which contains five IUDDs for: the SPECjAppServer J2EE application, the SPECjAppServer data, WAS 5.1, DB2 8.1, and the DB2 Runtime Client. The set of IUDDs for SPECjAppServer and the underlying middleware stack reflects the structure depicted in Figure 1. Note that a set of IUDDs forms a directed, acyclic graph.

To automate the install process in our case study, we use the CHAMPS Task Graph Builder [5], which is the component that handles automated workflow creation in Solution Install. It traverses the parsed IUDDs, extracts the installable units, evaluates their dependencies and configuration actions. It then creates a Task Graph by taking into account the specified change management operation. Task graphs are fundamental to our approach for reducing configuration complexity, as they capture the configuration actions and sequencing constraints in a form that allows a configuration procedure to be executed automatically via a workflow engine. We use the BPEL4WS workflow language [12] to represent Task Graphs (cf. [5] for details).

Figure 4 depicts the major activities of a workflow that is automatically created by the CHAMPS Task Graph Builder from the set of SPECjAppServer IUDDs. It consists of two major sequences (Application Server Sequence and Database Server Sequence), which group the provisioning and configuration activities according to the systems on which the activities need to be carried out. Whenever no dependencies exist between activities in a flow, a workflow engine will carry them out concurrently, which has the potential of significant time savings, especially if they are to be carried out on different systems. Within each of these per-host sequences, we distinguish between the various change management operations as well as the containers (hosting environments) to which they apply. The former deal with deploying the software products from a repository to the targets, installing and configuring them. In order to keep the workflows at an acceptable level of granularity, we rely on a set of small configuration scripts [13], written in the PERL (for OS actions) and jacl (for WAS) scripting languages. Before installing a component, its hosting environment needs to be started.

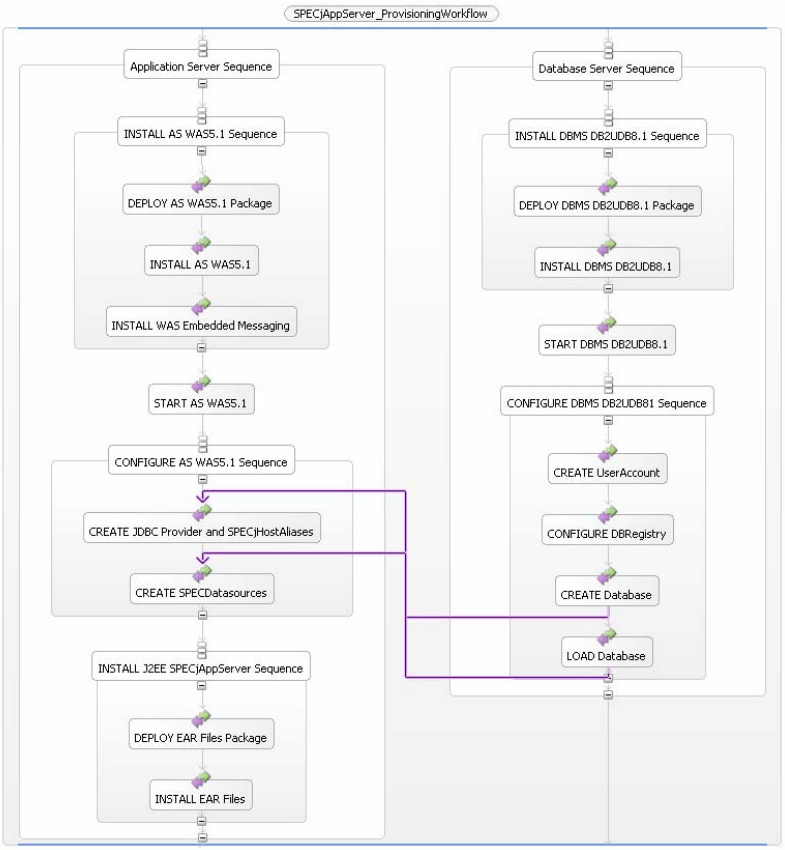

Figure 4: Automatically generated workflow for deploying, installing and configuring SPECjAppServer and its required middleware. It is generated by the CHAMPS Task Graph Builder and rendered in the WebSphere BPEL4WS Editor.

Note that complexities related to coordinating install actions between systems (reflected by high scores on the memory complexity metrics) are addressed in Figure 4, as indicated by the two bold horizontal arrows that indicate cross-system constraints. The CHAMPS automation is able to handle these considerations because of the richness of the SI IUDD specifications. For example, configuration parameters can either be *static* (its value determined up-front by the administrator) or *dynamic* (its value determined at runtime without administrator involvement), and are specified as such in the associated IUDD. Examples of static parameters include the DB2 and WAS install paths, the name and location of the database, and the user ids/passwords for database and application servers. An example of a dynamic parameter is the port number of the DBMS daemon, which needs to be passed as input into the activity that creates the JDBC provider as part of the WAS setup.

	Measure	Value (CHAMPS Automation)
Exec	NumActions	5
	ContextSwitchSum	1
Param.	ParamCount	17
	ParamUseCount	17
	ParamCrossContext	0
	ParamAdaptCount	0
	ParamSourceScore	94
Memory	MemSizeMax	0
	MemSizeAvg	0
	MemDepthMax	0
	MemDepthAvg	0
	MemLatMax	0
	MemLatAvg	0

Table 2: Configuration complexity results for CHAMPS-based SPECjAppServer installation. All metrics are reduced from those for manual installation (see Table 1).

We now evaluate the effect of the CHAMPS automation on human-perceived configuration complexity using the 13 configuration complexity metrics discussed earlier. We apply CHAMPS to the same scenario we have used throughout: installation of the SPECjAppServer application and supporting middleware stack. We assume a simple interface into the CHAMPS functionality: the user first loads an IU and associated deployment descriptor, supplies the values for the requested parameters, makes any changes they desire to the default scheduling constraints, then kicks off the automated process. Note that we do not include the complexity of generating the deployment descriptor, as we expect that to be eventually accomplished via tooling.

Table 2 shows the results of our complexity analysis. In comparison to Table 1, there is a significant reduction in execution complexity—both in number of steps and context switches—and memory complexity has dropped to zero, as the CHAMPS system automatically handles the task of distributing shared parameters across the various configuration contexts. Key aspects of parameter complexity are reduced—notably the crucial cross-context and adaptation metrics. However, the total number of parameters needed remains somewhat high, as the administrator is still required to supply information like the desired name of the SPECjAppServer database and the installation paths of various software components. However, many of these parameters can be bound to default values, significantly ameliorating the remaining parameter complexity.

We note that automating the SPECjAppServer install process provides additional benefits beyond the reduction in human-perceived complexity. It also results in a significant reduction in the time to do the install because of exploiting parallelism in the install process. Based on our measurements, the install time is reduced by 33% (from about 50 minutes and 25 seconds to 34 minutes).

5. Related work

There is little work directly related to our quantification of configuration complexity. Most standard evaluation strategies, or benchmarks, focus on performance and other dynamic aspects of a particular static system configuration. Configuration-related benchmarks exist that attempt to test properties of a static configuration, such as the CIS Security Benchmarks [14], but these do not measure the complexity of changing the configuration, and hence cannot evaluate complexity-reducing technology.

That said, there are pieces of our approach that do connect with existing work in the field. For example, our abstract system model of containers with associated controls bears some resemblance to traditional management information models, such as the various IETF MIBs or CIM: each container models a component of the computer system, and its controls represent settable properties and invocations. Our complexity metrics are analogous to those used to quantify software complexity, as in Zuse's excellent summary [8], although they are much more human-centric. In that sense, our approach bears more resemblance to the Model Human Processor (MHP) concept [9] than any software analysis approach. However, unlike MHP, we do not attempt to precisely model human response, and our primitive configuration actions are significantly higher-level than MHP primitives. Finally, while configuration complexity could be examined through human user studies such as those often seen in the human factors and human-computer interaction (HCI) communities, our approach offers several advantages, namely that it is system-independent, quantitative, and designed to generate reproducible and easily-compared results, all of which are rare in the typical system-specific format of traditional user studies.

Our approach to reducing configuration complexity fits more into the traditional work of the field, although it offers several novel advantages. The automated creation of provisioning and configuration workflows from dependency models has not been tried in the field before CHAMPS. Nevertheless, the following related work has similar goals: A constraint satisfaction-based approach to dynamic service creation and resource provisioning in data centers is described in [15], which takes the service classes as well as the technical capabilities of the managed resources into account, but does not perform additional optimization. [16] describes the combination of LCFG with the Smartfrog system to create a Grid based system with a centralized policy base. CHAMPS, in contrast, can be used in a distributed context. Another example of related work is the Workflakes system [17], which uses workflows to perform adaptation and reconfiguration tasks. It comprises an adaptation controller, where workflows describe the dynamic adaptation loop. However, Workflakes requires an administrator to manually create workflows, thus introducing additional complexity.

6. Conclusions and future work

This paper develops a model of configuration complexity and demonstrates its value for a change management system. The model represents systems as a set of nested

containers with configuration controls. From this representation, we derive metrics that indicate configuration complexity, including execution complexity, parameter complexity, and memory complexity. Applying these metrics to the SPECjAppServer J2EE application and its associated middleware, we have been able to quantify the complexity of manual configuration. We then compare this with the complexity of configuring the same system using the CHAMPS change management system. The CHAMPS automation provides dramatic reductions in configuration complexity.

The results in this paper are a starting point. They demonstrate how to construct system-independent measures of configuration complexity and how complexity can be reduced. We are now investigating the extent to which these results can be generalized to other systems. Indeed, we see our progress to date as the development of a framework for quantitative, cross-system *benchmarking* of configuration complexity and self-configuring systems, fulfilling the vision we introduced in prior work [18].

Our work also shows the natural connection between the configuration model we develop, the Solution Install technology (which uses nested containers), and the CHAMPS change management system through its use of Solution Install descriptors as a source of dependency information. With these inputs, CHAMPS automatically detects the opportunity for parallelism in configuration operations. Our empirical results suggest that exploiting parallelism can lead to substantial time savings, such as about 33% reduction in the SPECjAppServer installation time.

Further work remains in our effort to develop effective models of configuration complexity and good technologies to reduce this complexity. In particular, we hope to calibrate our 13 measures of configuration complexity with human perceptions of configuration complexity. To do this, we are developing a set of user studies that will involve system administrators of various skill levels; our intent is to produce a mapping from the measures in this paper to higher-level measures such as success probability, configuration time, and required skill level to complete configuration tasks.

References

[1] D. A. Patterson, A. B. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, et al. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. *University of California, Berkeley Computer Science Division Technical Report UCB//CSD-02-1175*, 15 March 2002.

[2] R. Murch, *Autonomic Computing*. Englewood Cliffs: Pearson Education, 2004.

[3] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. *Proc. 2003 USENIX Annual Technical Conference.* San Antonio, TX, June 2003.

[4] M. Burgess. Cfengine: A site configuration engine. *Computing Systems*, 8(3), 1995. USENIX Association, see also: http://www.cfengine.org.

[5] A. Keller and J.L. Hellerstein and J.L. Wolf and K.-L. Wu and V. Krishnan, The CHAMPS System: Change Management with Planning and Scheduling, *Proc. 9th IEEE/IFIP Network Operations and Management Symposium (NOMS'2004)*, IEEE, April, 2004.

[6] A. Keller, R. Badonnel, Automating the Provisioning of Application Services with the BPEL4WS Workflow Language, *Proc. 15th IFIP/IEEE International Workshop on Dis-*

tributed Systems: Operations & Management (DSOM 2004), Springer, November 2004.
[7] *SPECjAppServer2002 Design Document*, Version 1.00, October 2002, http://www.specbench.org/osg/jAppServer2002/docs/DesignDocument.html
[8] H. Zuse. *Software Complexity: Measures and Methods.* Berlin: Walter de Gruyter, 1991.
[9] S. Card, T. Moran, and A. Newell. *The Model Human Processor.* In Boff, K., et al. (Eds.) Handbook of Perception and Human Performance, 1986.
[10] M. Vitaletti (Editor), *Installable Unit Deployment Descriptor Specification*, Version 1.0., W3C Member Submission, IBM Corp., ZeroG Software, InstallShield Corp., Novell, July 2004, http://www.w3.org/Submission/2004/SUBM-InstallableUnit-DD-20040712/
[11] H. Chu (Editor), *Installable Unit Package Format Specification*, Version 1.0., W3C Member Submission, IBM Corp., ZeroG Software, InstallShield Corp., Novell, July 2004, http://www.w3.org/Submission/2004/SUBM-InstallableUnit-PF-20040712
[12] Business Process Execution Language for Web Services Version 1.1. Second Public Draft Release, BEA Systems, International Business Machines Corp., Microsoft Corp., SAP AG, Siebel Systems, May 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/.
[13] T. Lau, *Set up a SPECjAppServer2004 application with DB2 Universal Database*, developerWorks, IBM Corporation, July 2004. http://www-106.ibm.com/developerworks/db2/library/techarticle/dm-0407lau/
[14] Center for Internet Security. http://www.cisecurity.org.
[15] A. Sahai, S. Singhal, V. Machiraju, and R. Joshi. Automated Policy-Based Resource Construction in Utility Computing Environments. In R. Boutaba and S.-B. Kim, editors, *Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS'2004)*, pages 381 – 393, Seoul, Korea, April 2004. IEEE Publishing.
[16] P. Anderson, P. Goldsack, J. Paterson, SmartFrog meets LCFG: Autonomous Reconfiguration with Central Policy Control, *Proceedings of the 17th Systems Administration Conference (LISA 2003)*, 213-222, San Diego, CA, USA, October 2003. USENIX Association.
[17] G. Valetto and G. Kaiser. Using Process Technology to control and coordinate Software Adaptation. *Proceedings of the 25th International Conference of Software Engineering (ICSE 2003)*, pages 262 – 272, May 2003. IEEE Computer Society.
[18] A. B. Brown and J.L. Hellerstein. An Approach to Benchmarking Configuration Complexity. *Proc. 2004 SIGOPS European Workshop*. Leuven, Belgium, Sep. 2004.

Biography

Aaron B. Brown is a Research Staff Member at the IBM T.J. Watson Research Center. His research interests include understanding the role and impact of human system managers in large-scale IT infrastructures and quantifying and reducing IT management complexity. He is also one of the architects of IBM's Autonomic Computing effort. Brown received a PhD in computer science from the University of California, Berkeley.

Alexander Keller is a Research Staff Member at the IBM T.J. Watson Research Center. He received his M.Sc. and Ph.D. degrees in Computer Science from Technische Universität München, Germany, in 1994 and 1998, respectively and has published approximately 40 refereed papers in the area of distributed systems management. Dr. Keller's research interests revolve around change management for applications and services and service level agreements.

Joseph L. Hellerstein is a Research Staff Member and manager at the IBM T.J. Watson Research Center where he manages the Adaptive Systems Department. Dr. Hellerstein received his PhD from the University of California in Los Angeles. Dr. Hellerstein has published approximately 80 papers and has co-authored an Addison-Wesley book on expert systems and the Wiley book "Feedback Control of Computing Systems".