Towards Automated Deployment of Built-to-Order Systems

Akhil Sahai[1], Calton Pu[2], Gueyoung Jung[2],
Qinyi Wu[2], Wenchang Yan[2], and Galen S. Swint[2]

[1]HP Laboratories, Palo-Alto, CA
akhil.sahai@hp.com
[2]Center for Experimental Research in Computer Systems,
College of Computing, Georgia Institute of Technology,
801 Atlantic Drive, Atlanta, GA 30332
{calton, helcyon1, qxw, wyan, galen.swint}@cc.gatech.edu

Abstract. End-to-end automated application design and deployment poses a significant technical challenge. With increasing scale and complexity of IT systems and the manual handling of existing scripts and configuration files for application deployment that makes them increasingly error-prone and brittle, this problem has become more acute. Even though design tools have been used to automate system design, it is usually difficult to translate these designs to deployed systems in an automated manner due to both syntactic obstacles and the synchronization of multiple activities involved in such a deployment. We describe a generic process of automated deployment from design documents and evaluate this process for 1, 2, and 3-tier distributed applications.

1 Introduction

New paradigms, such as autonomic computing, grid computing and adaptive enterprises, reflect recent developments in industry [1, 2, 3] and research [4]. Our goal is to create "Built-to-Order" systems that operate in these new computing environments. This requires easy and automated application design, deployment, and management tools to address their inherent complexity. We must support creating detailed designs from which we can deploy systems. These designs, in turn, are necessarily based on user requirements that take into account both operator and technical capability constraints. Creating design in an automated manner is a hard problem in itself. Quartermaster Cauldron [5], addresses the challenge by modeling system components with an object-oriented class hierarchy, the CIM (Common Information Model) metamodel, and embedding constraints on composition within the models as policies. Then, Cauldron uses a constraint satisfaction approach to create system designs and deployment workflows. However, these workflows and designs are expressed in system-neutral Managed Object Format (MOF).

MOF workflows typically involve multiple systems and formats that have to be dealt with in order to deploy a complex system. For example, deploying a three-tier e-commerce solution in a virtualized environment may involve interactions with blade servers, VMWare/Virtual Servers, multiple operating systems, service containers for web servers, application servers, databases, before, finally, executing clients scripts. This problem of translating generic design in a system independent format (*e.g.*, MOF) to the multiple languages/interfaces demanded by the system environment is thus nontrivial.

The main contribution of the paper is a generic mechanism for translating design specifications written in a system independent format into multiple and varied deployment environments. To achieve this generic translation, we use an XML based intermediate representation and a flexible code generation method [6, 7] to build an extensible translator, the Automated Composable Code Translator (ACCT). Translation between the two models is non-trivial and significant result for two reasons. First, the models are quite dissimilar in some aspects; the translation is not a straightforward one-to-one mapping. For example, we describe the translation between significantly different workflow models in Section 2.3. Second, the ACCT design is deliberately generic to accommodate the multiple input and output formats encountered in multiple design and d environments. ACCT accepts MOF-based design specifications of CIM instance models and converts them into input specifications for SmartFrog, a high-level deployment tool [8]. SmartFrog uses on a high-level specification language and Java code to install, execute, monitor, and terminate applications. The generic architecture supporting multiple input/output formats is described elsewhere [6, 7].

2 Automated Design and Automated Deployment

2.1 Automated Design Environment

At the highest level of abstraction, automated design tools offer streamlined and verified application creation. Quartermaster is an integrated tool suite built around MOF to support automated design of distributed applications at this high level of abstraction [9, 10]. Cauldron, one of its key components, supports applying policies and rules at design-time to govern composition of resources. Cauldron's constraint satisfaction engine can generate system descriptions that satisfy these administrative and technical constraints. In this paper, we concentrate on deployment constraints for distributed applications. Since each component of an application often depends on prior deployment of other components or completion of other components' work, deployment is non-trivial.

To model deployment, we use a MOF Activity comprised of a number of sub-activities. Each of these activities has a set of constraints to meet before execution and also parameters that must receive values. At design time, Cauldron generates configuration templates and also pairwise deployment dependencies between deployment activities. Between any pair of activities, there are four possible synchronization dependencies.

SS (Start-Start) – activities must start together; a symmetric, transitive dependency.

FF (Finish-Finish) –activities must finish together (synchronized); also a symmetric,

transitive dependency.

FS (Finish-Start) – predecessor activity must complete before the successor activity is started, *i.e.*, sequential execution. This dependency implies a strict ordering, and the MOF must assign either the antecedent or the dependant role to each activity component.

SF (Start-Finish) – predecessor activity is started before the successor activity is finished. Similar observations on its properties follow as from FS. (As an SF example, consider producer-consumer relationships in which the producer must create a communication endpoint before the consumer attempts attachment.)

Cauldron, however, is solely a design tool and provides no deployment tools, which require software that initiate, monitor, and kill components in a distributed environment.

2.2 Automated Deployment Environment

Automated deployment tools serve to ameliorate the laborious process of preparing, starting, monitoring, stopping, and even post-execution clean-up of distributed, complex applications. SmartFrog is an open-source, LGPL framework that supports such service deployment and lifecycle management for distributed Java applications [11, 12]; it has been used on the Utility Computing model for deploying rendering code on demand and has been ported to PlanetLab [13]. Expertise gained applying SmartFrog to grid deployment [14] is being used in the CDDLM standardization effort currently underway

Conceptually, SmartFrog comprises 1) a component model supporting application-lifecycle operations and workflow facilities, 2) a specification language and validator for these specifications, and 3) tools for distribution, lifecycle monitoring, and control. The main functionalities of SmartFrog are as follows:

Lifecycle operations – SmartFrog wraps deployable components and transitions them through their life phases: *initiate*, *deploy*, *start*, *terminate,* and *fail*.

Workflow facilities – Allows flexible control over configuration dependencies between components to create workflows. Examples: *Parallel*, *Sequence*, and *Repeat*.

SmartFrog runtime – Instantiates and monitors components; provides security. The runtime manages interactions between daemons running on remote hosts. It provides an event framework to send and receive events without disclosing component locations.

SmartFrog's specification language features data encapsulation, inheritance, and composition which allows system configurations to be incrementally declared and customized. In practice, SmartFrog needs three types of files to deploy an application:

1. Java `interface` definitions for components. These serve analogously to the interface exposure role of the C++ header file and `class` construct.
2. Java source files that implement components as objects. These files correspond one-to-one with the above SmartFrog component descriptions.
3. A single instantiation and deployment file, in a SmartFrog specific language, defining the parameters and proper global deployment order for components.

2.3 Translating Between Design Specifications and Deployment Specifications

This section describes ACCT, our extensible, XML-based tool that translates generic design specifications into fully parameterized, executable deployment specifications. First, we describe ACCT's design and the implementation and then the mapping approach needed to resolve mismatches between the design tool output (MOF) and deployment tool input (SmartFrog).

There are several obstacles to translating Cauldron to SmartFrog. First, there is the syntax problem; Cauldron generates MOF, but SmartFrog requires a document in its own language syntax as well as two more types supporting of Java source code. Obviously, this single MOF specification must be mapped to three kinds of output files, but neither SmartFrog nor Quartermaster supports deriving Java source from the design documents. Finally, Cauldron only produces *pairwise* dependencies between deployment activities; SmartFrog, on the other hand, needs dependencies over the entire set of deployment activities to generate a deployment workflow for the system.

In ACCT, code generation is built around an XML document which is compiled from a high-level human-friendly specification language (MOF) and then transformed using XSLT. So far, this approach has been applied to a code generation system for information flow architectures and has shown several benefits including support for rapid development, extensibility to both new input and output languages, and support for advanced features such as source-level aspect weaving. These advantages mesh well with ACCT's goals of multiple input languages and multiple output languages, and SmartFrog deployments, in fact, require ACCT to generate two different output formats.

The code translation process consists of three phases which are illustrated in Fig. 1. In the first phase, MOF-to-XML, ACCT reads MOF files and compiles them into a single XML specification, XMOF, using our modification of the publicly available WBEM Services' CIM-to-XML converter [15].

In phase two, XML-to-XACCT, XMOF is translated into a set of XACCT documents, the intermediate XML format of the ACCT tool. During this transformation, ACCT processes XMOF in-memory as a DOM tree and extracts three types of Cauldron-embedded information: components, instances, and deployment workflow. Each data sets is processed by a dedicated code generator written in Java. The component generator creates an XML component description, the instance generator produces a set of attributes and values, as XML, for deployed components, and the workflow generator computes a com-

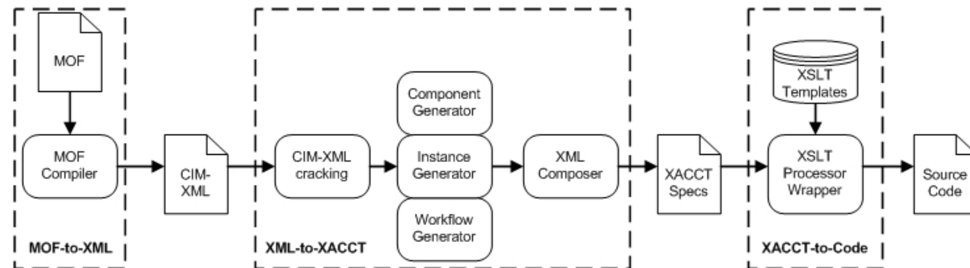

Fig. 1. The ACCT code generator.

plete, globally-ordered workflow expressed in XML. (We will describe the workflow construction in more detail later.) These generated structures are passed to an XML composer which performs rudimentary type checking (to ensure instances are only present if there is also a class), and re-aggregates the XML fragments back into a whole XML documents. This may result in multiple XACCT component description documents, but there is only one instantiation+workflow document which contains the needed data for a single deployment.

Finally, in the third phase ACCT forwards each XACCT component description, instantiation, and workflow document to the XSLT processor. The XSLT templates detect the XACCT document type and generate the appropriate files (SmartFrog or Java) which are written to disk.

XACCT allows components, configurations, constraints, and workflows from input languages of any resource management tool to be described in an intermediate representation. Once an input language is mapped to XACCT, the user merely creates an XSLT template for the XML-to-XACCT phase to perform the final mapping of the XACCT to a specific target language. Conversely, one need only add a new template to support new target languages from an existing XACCT document.

Purely syntactic differences between MOF and SmartFrog's language can be resolved using solely XSLT, and the first version of ACCT was developed on XSLT alone. However, because the XSLT specification version used for ACCT had certain limitations, we incorporated Java pre- and post- processing stages. This allowed us to compute the necessary global SmartFrog workflows from the MOF partial workflows and to create multiple output files from a single ACCT execution.

Overall system ordering derives from the Cauldron computed partial synchronizations encoded in the input MOF. As mentioned in Section 2.1, MOF defines four types of partial synchronization dependencies: SS, FF, SF, and FS. To describe the sequential and parallel ordering of components which SmartFrog requires, these partial dependencies are mapped via an event queue model with an algorithm that synchronizes activities correctly. It is helpful to consider the process as that of building a graph in which each component is a node and each dependency is an edge. Each activity component has one associated EventQueue containing list of actions:

Execute – execute a specific sub-component.

EventSend – send a specific event to other components. This may accept a list of destination components.

OnEvent – the action to wait for an incoming event. This may wait on events from multiple source components. It is the dual of EventSend.

Terminate – the action to remove the EventQueue.

Table 1. Possible event dependencies between components

C	A component, C.
$C_a \rightarrow C_b$	Component C_a sends event to C_b.
$C_a \leftarrow C_b$	Component C_a waits for event from C_b.
$C_a — C_b$	Components *must* perform action together

Given this model, any two components may have one of three synchronization relations, as shown in Table 1. Fig. 3 applies these synchronization se-

mantics to the pairwise MOF relationships. In SS, two activity components are blocked until each event receives a "start" event from the other. In ACCT, this translates to entries in the EventQueues to send and then wait for an event from the peer component. The FF scenario is handled similarly. In SF, since C_b's activity must be finished after C_a starts to deploy, C_b is blocked in its EventQueue until C_a's "start" is received at C_b. In FS, since C_b may deploy only after C_a completes its task, C_b blocks until a "finished" event from C_a is received at C_b. (For now, we assume the network delay between two components is negligible.)

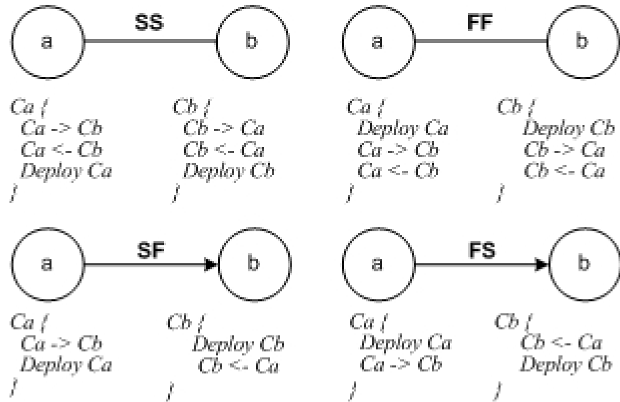

Fig. 3. Diagrams and dependency formulations of SS, FF, SF, and FS.

Fig. 2 illustrates the XACCT for the FS dependency. The SS and FF operations are parallel deployment activities while SF and FS represent sequential deployment activities.

The exact content of each EventQueue depends on its dependencies to all other activity components. Since each activity component frequently has multiple dependencies, we devised an algorithm to calculate EventQueue contents.

The algorithm visits each activity component, C_i, in the XMOF to build a global action list. If a dependency of the component is a parallel dependency (SS or FF), then the algorithm transitively checks for dependencies of the same type on related activity components until it finds no more parallel. For example, if there is a dependency in which C_i is SS with C_j, and C_j is also SS with C_k but FF (a different parallel dependency) with C_m, it records only "C_j and C_k" as SS on its action list before proceeding to check component C_{i+1}. If it is a sequential dependency (FS or SF), the algorithm adds the dependency to the global action list and proceeds to component i+1. That is, if C_i has FS with C_j, and C_j has FS with C_k, only the pairwise relation "C_i and C_j with FS" is entered into the global action list before proceeding to C_{i+1}.

```
<Instance Name="Ca" Class="Activity">
  <Workflow>
    <Work Name="--" Type="Execute">
    </Work>
    <Work Name="--" Type="EventSend">
      <To>Cb</To>
    </Work>
    <Work Name="--" Type="Terminator">
     Ca</Work>
  </Workflow>
</Instance>
<Instance Name="Cb" Class="Activity">
  <Workflow>
    <Work Type="OnEvent">
      <From>Ca</From>
    </Work>
    <Work Name="--" Type="Execute">
    </Work>
    <Work Name="--" Type="Terminator">
     Cb</Work>
  </Workflow>
</Instance>
```

Fig. 2. XACCT snippet for the FS dependency.

Then the algorithm implements deadlock avoidance by enforcing a static order of actions for each activity component C_i based on the component's role, antecedent or dependant, in each relationship. The algorithm checks

the six possible combinations of roles and dependencies as follows:

1. If C_i participates as a dependant in any FS relationship, then it adds one OnEvent action to the EventQueue per FS-Dependancy.
2. If C_i has any SS dependencies, then it adds all needed EventSend and OnEvent actions to the EventQueue.
3. If C_i functions as antecedent in SF dependencies, then per dependency it adds an EventSend action to the EventQueue followed by a single Execute action.
4. If C_i participates as a dependant in an SF dependency, then one OnEvent action per dependency is added to the EventQueue.
5. If C_i has any FF dependencies, it adds all EventSend and OnEvent actions to the EventQueue.
6. Finally, if C_i serves as an antecedent roles with FS, then it adds one EventSend action per FS occurrence.

Finally, the workflow algorithm appends the "Terminate" action to each C_i's EventQueue.

XACCT captures the final workflow in a set of per-component EventQueues, which ACCT then translates to the input format of the deployment system (*i.e.*, SmartFrog). The Java source code generated by ACCT is automatically compiled, packaged into a `jar` file, and integrated into SmartFrog using its class loader. An HTTP server functions as a repository to store scripts and application source files. Once a SmartFrog description is fed to the SmartFrog daemon, it spawns one thread for each activity in the workflow. Subsequent synchronization among activities is controlled by EventQueues.

3 Demo Application and Evaluation

We present in this section how Cauldron-ACCT-SmartFrog toolkit operates from generating the system configurations and workflow, translating both into the input of SmartFrog, and then automatically deploying distributed applications of varying complexity. In the subsection 3.1, we describe 1-, 2-, and 3-tier example applications and system setup employed for our experiments. Following that, we evaluate our toolkit by comparing the deployment execution time of SmartFrog and automatically generated deployment code to manually written deployment scripts.

3.1 Experiment Scenario and Setup

We evaluated our translator by employing it on 1-, 2-, and 3-tier applications. The simple 1- and 2-tier applications provide baselines for comparing a generated SmartFrog description to hand-crafted scripts. The 3-tier testbed comprises web, application, and database servers; it is a small enough size to be easily testable, but also has enough components to illustrate the power of the toolkit for managing complexity. Table 2, below, lists each scenario's components.

Table 2. Components of 1-, 2-, and 3-tier applications

Scenario	Application	Components
1-tier	Static web page	Web Server : Apache 2.0.49
2-tier	Web Page Hit Counter	Web Server : Apache 2.0.49 App. Server : Tomcat 5.0.19 Build System: Apache Ant 1.6.1
3-tier	iBATIS JPetStore 4.0.0	Web Server : Apache 2.0.49 App. Server : Tomcat 5.0.19 DB Server : MySQL 4.0.18 DB Driver : MySQL Connector to Java 3.0.11 Build System : Apache Ant 1.6.1 Others : DAO, SQLMap, Struts

We installed SmartFrog 3.04.008_beta on four 800 MHz dual-processor Dell Pentium III machines running RedHat 9.0; one SmartFrog daemon runs on each host.

In the 1-tier application, we deploy Apache as a standalone web server, and confirmed successful deployment by visiting a static web page. The evaluation used two machines: the first for the web server and a second to execute the generated SmartFrog workflow.

In the 2-tier Hit Counter application, we used Apache and the Tomcat application server with Ant. Each tier specified for deployment to a separate host. To verify the 2-tier deployment, we visited the web page multiple times to ensure it recorded page hits. The application simply consists of a class and a `jsp` page. The 2-tier evaluation required three machines. As in the 1-tier test, we used one machine to run the deployment script; then, we dedicated one machine to each deployed tier (Apache; Ant and Tomcat).

Finally, the 3-tier application was the iBATIS JPetStore, a ubiquitous introduction to 3-tier programming. In the 3-tier application evaluation, we used four machines. Again, we dedicated one machine for each tier (Apache; Tomcat, JPetStore, Ant, MySQL Driver, Struts; MySQL DB) and used a fourth machine to run the SmartFrog workflow.

Fig. 4 illustrates the dependencies of components in each testbed. We consider three types of dependencies in the experiment; installation dependency, configuration dependency, and activation dependency. The total number of dependencies in each testbed is used as the level of the complexity. In Figure 6, 1-, 2-, and 3-tier testbeds are considered as simple, medium, and complex cases respectively. Intuitively, the installation, configuration, and activation dependencies of each component in each testbed must be sequenced. For instance, the Apache configuration must start after Apache installation completes, and Apache activation must start after Apache configuration completes. (For space, we have omitted these dependencies from the figure.)

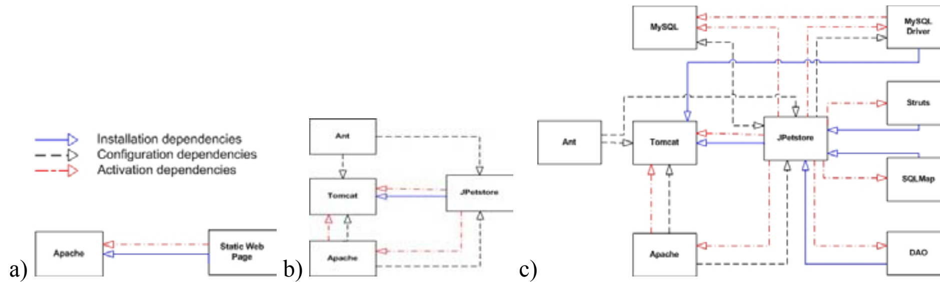

Fig. 4. Dependency diagrams of (a) 1-tier, (b) 2-tier, and (c) 3-tier application.

We modeled 1-, 2-, and 3-tier applications in Quartermaster with and Cauldron module created the configurations and deployment workflows. The resultant MOF files were fed into ACCT and yielded a set of Java class files, SmartFrog component descriptions, and a SmartFrog instances+workflow specification for each application tested. Fig. 6 on the following page illustrates the transformation process as ACCT translates the MOF file of the 3-tier application to intermediate XACCT and then finally to a SmartFrog description. For demonstration, we highlight the FS dependency between the Tomcat installation and MySQLDriver installation and how this information is carried through the transformation process.

3.2 Experimental Result

The metric we choose for evaluating the 1-, 2-, and 3-tier testbeds is deployment execution time as compared to manually written scripts. We executed SmartFrog and scripts 30 times each for each tier application and report the average. Fig. 5 shows that for simple cases (1- and 2-tier) SmartFrog took marginally longer when compared to the scripts based approach because SmartFrog daemons running in the Java VM impose extra costs when loading Java classes or engaging in RMI communication. The trend favors SmartFrog as the time penalty of the medium case becomes less (in absolute and relative terms) and for the complex case, SmartFrog took less time than the scripts based approach.

In the complex case, SmartFrog was able to exploit concurrency between application components since it had a computed workflow. The simple and medium cases contain fewer concurrent dependencies than the 3-tier case.

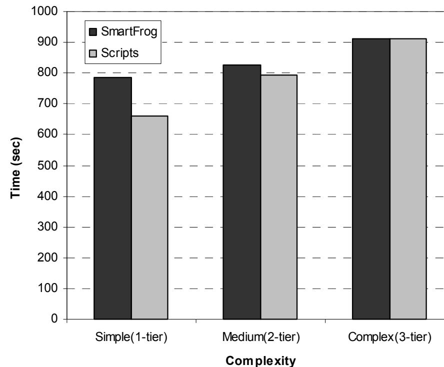

Fig. 5. Deployment Time using SmartFrog and scripts as a function of the complexity.

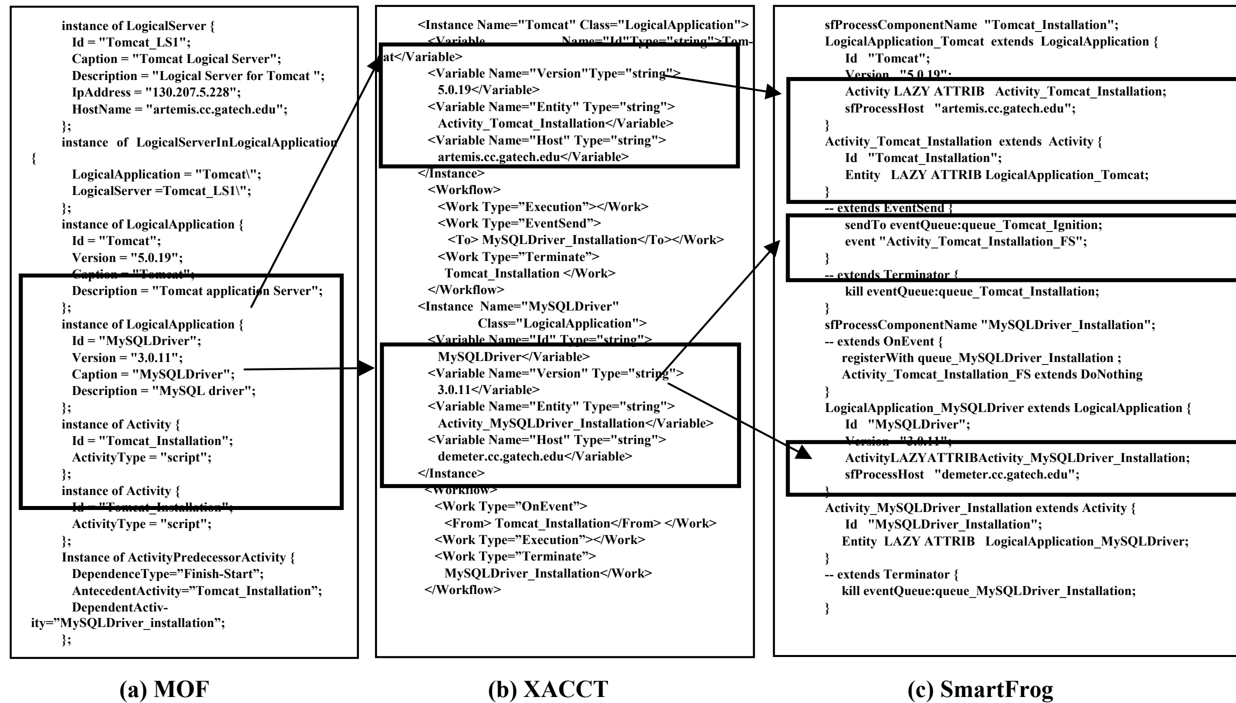

Fig. 6. (a) MOF, (b) Intermediate XML, and (c) SmartFrog code snippets. The solid line box indicates the FS workflow between Tomcat and MySQLDriver applications. Others indicate configurations. Clearly, MOF offers superior understandability for a deployment scenario as compared to the SmartFrog specification. As Vanish et al showed in [16], automating deployment via SmartFrog, for which we generate code, is generally superior in performance and more maintainable when compared to manual or *ad hoc* scripted solutions.

Nevertheless, in all cases our toolkit retains the important advantage of an automatically generated workflow, while in scripts based approach, system administrators must manually control the order of installing, configuration, and deployment.

4 Related Work

Recent years have seen the advent of wide-ranging resource management systems. For e-business, OGSA Grid Computing [17] aims to provide services within an on-demand data center infrastructure. IBM's Autonomic Computing Toolkit [1], the HP Utility Data Center [18] and Microsoft's DSI initiative [3] are examples of this. The distinction of our toolkit, however, is that Cauldron logic and a theorem prover to meet resource allocation constraints. There are several efforts related to specifying conditions and actions for policies, *e.g.*, CIM [19] and PARLAY [20]. However, to the best of our knowledge, none of them have used a constraint satisfaction approach for automatic resource construction.

Another trend is deployment automation tools. CFengine [22] provides rich facilities for system administration and is specifically designed for testing and configuring software. It defines a declarative language so that the transparency of a configuration program is optimal and management is separate from implementation. Nix [21] is another popular tool used to install, maintain, control, and monitor applications. It is capable of enforcing reliable specification of component and support for multiple version of a component. However, since Nixes does not provide automated workflow mechanism, users manually configure the order of the deployments. For deployment of a large and complicated application, it becomes hard to use Nixes. By comparison, SmartFrog provides control flow structure and event mechanism to support flexible construction of workflow.

The ACCT translator adopts the Clearwater architecture developed for the Infopipe Stub Generator + AXpect Weaver (ISG) [6, 7]. Both ACCT and ISG utilize an XML intermediate format that is translated by XSLT to target source code. Unlike ACCT, however, ISG is designed for creating information flow system code. There are other commercial and academic translation tools, like MapForce [23] and CodeSmith [24]. Similar to ISG, they target general code generation and do not support deployment workflows.

5 Conclusion

We outlined an approach for Automated Deployment of complex distributed applications. Concretely, we described in detail the ACCT component (Automated Composable Code Translator) that translates Cauldron output (in XMOF format) into a SmartFrog specification that can be compiled into Java executables for automated deployment. ACCT performs a non-trivial translation, given the differences between the XMOF and SmartFrog models such as workflow dependencies. A demonstration application (JPetStore) illustrates the automated design and implementation process and translation steps, showing the increasing advantages of such automation as the complexity of the application grows.

Acknowledgement: This work was partially supported by NSF/CISE IIS and CNS divi-

sions through grants IDM-0242397 and ITR-0219902, DARPA ITO and IXO through contract F33615-00-C-3049 and N66001-00-2-8901, and Hewlett-Packard. We thank our anonymous reviewers and M. Feridun for their very helpful comments.

References

1. IBM Autonomic Computing. http://www.ibm.com/autonomic.
2. SUN N1. http://www.sun.com/software/n1gridsystem/.
3. Microsoft DSI. http://www.microsoft.com/windowsserversystem/dsi/.
4. Global Grid Forum. http://www.ggf.org.
5. Sahai, A., Singhal, S., Joshi, R., Machiraju, V.: Automated Policy-Based Resource Construction in Utility Computing Environments. NOMS, 2004.
6. Swint, G. and Pu, C.: Code Generation for WSLAs using AXpect. 2004 IEEE International Conference on Web Services. San Diego, 2004.
7. Swint, G., Pu, C., Consel, C., Jung, G., Sahai, A., Yan, W., Koh, Y., Wu, Q.: Clearwater - Extensible, Flexible, Modular Code Generation. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2005.
8. SmartFrog. http://www.hpl.hp.com/research/smartfrog/.
9. Salle, M., Sahai, A., Bartolini, C., Singhal, S.: A Business-Driven Approach to Closed-Loop Management. HP Labs Technical Report HPL-2004-205, November 2004.
10. Sahai, Akhil, Sharad Singhal, Rajeev Joshi, Vijay Machiraju: Automated Generation of Resource Configurations through Policies. IEEE Policy, 2004.
11. Goldsack, Patrick, Julio Guijarro, Antonio Lain, Guillaume Mecheneau, Paul Murray, Peter Toft.: SmartFrog: Configuration and Automatic Ignition of Distributed Applications. HP Openview University Association conference, 2003.
12. Smartfrog open source directory. http://sourceforge.net/projects/smartfrog.
13. Peterson, Larry, Tom Anderson, David Culler, and Timothy Roscoe: A Blueprint for Introducing Disruptive Technology. PlanetLab Tech Note, PDN-02-001, July 2002.
14. CDDLM Foundation Document. http://www.ggf.org/Meetings/ggf10/GGF10%20Documents/CDDLM_Foundation_Document_v12.pdf.
15. WBEM project. http://wbemservices.sourceforge.net.
16. Talwar, Vanish, Dejan Milojicic, Qinyi Wu, Calton Pu, Wenchang Yan, and Gueyoung Jung: Comparison of Approaches to Service Deployment. ICDCS 2005.
17. Foster, Ian, Carl Kesselman, Jeffrey M. Nick, and Steven Tuecke: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Globus Project, 2002.
18. HP Utility Data Center. http://www.hp.com/products1/promos/adaptive_enterprise/us/utility.html.
19. DMTF-CIM Policy. http://www.dmtf.org/standards/cim/cim_schema_v29.
20. PARLAY Policy Management. http://www.parlay.org/specs/.
21. Dolstra, Eelco, Merijn de Jonge, and Eelco Visser: Nix: A Safe and Policy-free System for Software Deployment. 18th Large Installation System Administration Conference, 2004.
22. Cfengine. http://www.cfengine.org/.
23. Altova Mapforce. http://www.altova.com/products_mapforce.html.
24. Codesmith. http://www.ericjsmith.net/codesmith.