Towards Automated Deployment of Built-to-Order Systems

Akhil Sahai, Calton Pu[1], Gueyoung Jung[1], Qinyi Wu[1], Wenchang Yan[1], Galen Swint[1]
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2005-167
September 21, 2005*

automation, deployment, utility computing, SmartFrog, XML, XSLT

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 as multiple activities are involved in such a deployment. We describe a generic process of automated deployment and an evaluation of the tool.

* Internal Accession Date Only
[1] Georgia Institute of Technology, Atlanta, GA, USA
Published in Distributed Systems: Operations and Management (DSOM) 2005, 24-26 October 2005, Barcelona, Spain

Towards Automated Deployment of Built-to-Order Systems

Akhil Sahai, Calton Pu†, Gueyoung Jung†, Qinyi Wu†, Wenchang Yan†, and Galen Swint†
HP Laboratories,Palo-Alto, CA
Georgia Institute of Technology†, Atlanta
akhil.sahai@hp.com, [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 automated system design, it is usually difficult to translate these designs to deployed systems in an automated manner as multiple activities are involved in such a deployment.. We describe a generic process of automated deployment and an evaluation of the tool.

1. Introduction

New paradigms such as autonomic computing and adaptive enterprises reflect recent developments in industry [1][2][3] and research [4] require easy and automated application design, deployment, and management tools.

Our goal is to create "Built-to-Order" systems. In order to achieve this goal we need to create detailed designs and deploy systems based on these detailed design specification. These designs have to be based on user requirements, taking into account operator constraints and technical capability constraints, Creating design in an automated manner in itself is a hard problem. In Quartermaster Cauldron [8], this goal was achieved by modeling system components with an object-oriented class hierarchy, the CIM (Common Information Model) meta-model, and embedding constraints on composition within the models as policies. We have used a constraint satisfaction approach to create system designs and create a workflow to deploy these designs. However, these workflows and designs are expressed in system-neutral Managed Object Format (MOF). These workflows typically involve multiple systems and formats that have to be dealt with in order to deploy a complex system, for example deploying a complex three tier e-commerce system on a virtualized environment may involve, dealing with the blade server interfaces, VMWare/Virtual Server Interfaces, Operating System installations, Service container interfaces for web servers, appservers and databases and execution of client scripts. A single design thus has to deal with a large number of system actuators to automatically install a complex system The problem of translating generic design in a system indepemdent format like MOF to multiple languages/interfaces understood by various system actuators/deployment environments in a generic manner is thus non-trivial.

The main technical contribution of the paper is the generic mechanism for translating design specification written in a system independent format into multiple and varied deployment environments. In order to achieve this generic translation, we use an XML-based intermediate representation and a flexible code generation method [6] to build an extensible translator, ACCT (Automated Composable Code Translation tool), that accepts a design specification in the form of CIM instance models expressed in MOF and converts them into a high level deployment specification language called SmartFrog [5] and generates the Java code for execution. executable Java code that installs the application components on appropriate hardware and software platforms, and then starts the application execution. The SmartFrog compiler links generated design specification and Java code.

The translation between the two models is a significant result for two reasons. First, the models are quite dissimilar in some aspects and the translation is not straightforward one-to-one mapping. Specifically, the workflow models differ significantly between the design and deployment environments. Second, ACCT is designed to be more generic and be capable of handling multiple input and output formats thus making it flexible enough to handle multiple design and deployment environments.

2. Automated Design and Workflow Generation

2.1 Design Environment

Quartermaster is an integrated tool suite supporting automated design of distributed applications at a high level of abstraction. Tools in the Quartermaster suite are built around the MOF. One of its key features, Cauldron, supports applying policies and rules to govern composition of resources. Cauldron is a constraint satisfaction engine that can generate a system description that satisfies the administrative and technical constraints.

For this paper, we will concentrate on constraints for deploying distributed applications. Deployment is non-trivial, since each component of an application often depends on the pre-deployment of other components or completion other components' work. Deployment is modeled as an Activity, and an Activity can comprise a series of sub-activities. Each activity has a set of attributes parameters that must be set, and an activity can only deploy when its contraints are met. At this time, we generate configuration templates and, by associating configuration activities to the classes, also generate pair wise dependencies between the deployment activities.

Between any pair of Quartermaster activity entities, there are four types of synchronization dependencies.

SS (Start-Start) – activities must start together. This is a symmetric and transitive dependency.

FF (Finish-Finish) –activities must finish together (synchronized). This is also a symmetric and transitive dependency.

FS (Finish-Start) – predecessor activity must before the successor activity is started, that is, sequential execution. This dependency implies a strict ordering, and the MOF must assign either the Antecedent (A) or the Dependant (D) role to each component.

SF (Start-Finish) – predecessor activity is started before the successor activity is finished. Similar observations on its properties follow as from FS.

(While this seems an odd dependency at first, it is actually quite common. For example, in a producer-consumer relationship, the producer must often create a communication endpoint before the consumer starts and attempts attachment or else it will abort.)

Cauldron, however, can not actually deploy activities it is used to design, for that, we need a dedicated deployment tool that can initiate, monitor, and kill components in a distributed environment.

2.2 Deployment Environment

Automatically generated system configurations and workflow are translated to the specific formats to be used as inputs for deployment/life-cycle management environments. In order to deploy Quartermaster components without any further human interactions, we must resolve both syntactic differences and structural differences between Quartermaster MOF and deployment specifications such as SmartFrog. Especially, the global workflow usually involves many components, their complex relationships, and dependencies among components, but Quartermaster MOF only contains a list of partial dependencies between component pairs.

The ACCT translation tool addresses these impedance mismatches through the construction of a workflow expressing global dependencies among all involved components and mapping MOF syntax to an XML specification. The ACCT is also designed to generate various deployment specification formats (e.g., SmartFrog description, Radia). Thus, it can be used as a pluggable code translation component for any resource management and deployment systems. Section 2.3 describes the ACCT in more detail.

SmartFrog is a framework for service configuration specification, deployment and lifecycle management of distributed Java applications [10]. It has been used on the Utility Computing model for deploying rendering code on demand. There is a PlanetLab port [11] and the CDDLM standardization effort leverages the expertise of SmartFrog for Grid deployment [12]. The SmartFrog source code is available under the LGPL license [13].

SmartFrog consists of a component model supporting application-lifecycle operations and workflow facilities, a data description language, a validator for these descriptions, and tools for distribution, lifecycle monitoring, and control. The main features of SmartFrog are as follows:

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

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

SmartFrog runtime – Responsible for component instantiation, monitoring and security, the runtime manages daemons running on remote hosts and controls the interaction between them including providing an event framework to send and receive events without disclosing component locations.

The SmartFrog language features data encapsulation, inheritance, and composition which allow system configurations to be incrementally declared and customized. In practice, there are three types of files that SmartFrog needs to deploy an application. First, SmartFrog needs a set of component definition files that define components' Java interfaces. This is somewhat analogous to the interface exposure role of the C++ header file with respect to the `class` construct. Second, there must be Java source files that implement the components as objects. These files correspond one-to-one with SmartFrog component descriptions. Third, SmartFrog needs a single instantiation and deployment file that defines deployment parameters and proper deployment order for the components and workflows.

2.3 Translating Design Specifications to Deployment Environments

In this section, we describe ACCT, the tool used in our evaluation (Section 3). First, we describe the design and the implementation of ACCT, and then we describe the mapping approach needed to resolve mismatches between the design tool output format and deployment tool input. ACCT is based on an extensible XML-based architecture and maps the model-level MOF code to a SmartFrog Language specification document.

There are several challenges when logically connecting Cauldron to SmartFrog. First, there is the syntax problem in which Cauldron tool is unified around and generates MOF specification, but SmartFrog has its own language syntax. Furthermore, while the MOF may contain all data woven into a single description file SmartFrog needs three kinds of files, as outlined above, and neither SmartFrog nor Quartermaster supports building Java source files from the component design documents. Also, Cauldron only produces pair wise dependencies between deployment activities. SmartFrog, on the other hand, needs dependencies calculated over the entire set of deployment activities to generate deployment workflows for each component in 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 a general purpose language facility and XSLT. So far, the architecture has been applied to a code generation system for information flow architectures and has proven to support rapid development, is extensible to new target and input languages, and can support advanced features such as aspect weaving. Multiple input languages and multiple output languages are goals of the architecture, and SmartFrog deployments require ACCT to generate multiple output formats (Java and SmartFrog's language). In fact, we aim to employ ACCT as a pluggable code translation component for any resource management and deployment system pair.

The code translation process consists of three phases (see Figure 1). In the first phase, conversion, ACCT reads MOF files and compiles them into a single XML specification

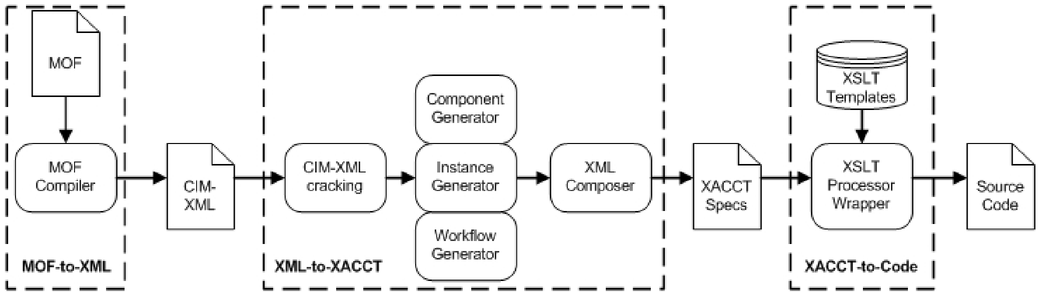


Figure 1. The ACCT code generator.

(CIM-XML specification) using the publicly available WBEM Services' CIM-to-XML converter. The WBEM Services project is an open-source Java implementation from Sun Microsystems [14]. We modified the CIM-to-XML converter to use it as MOF compiler of ACCT.

The second phase converts CIM-XML into a set of XACCT documents, the intermediate format of the ACCT tool. In this transformation, ACCT processes CIM-XML as an in-memory DOM tree and extracts the three types of information woven into the MOF by Cauldron: Components, Instances, and Deployment Workflow. The extracted data sets are each processed by three dedicated XML-to-XML generators written in Java. The component generator creates a XML component description, the instance generator produces a set of attributes and values for deployed components, and the workflow generator computes a complete, ordered workflow from Cauldron's pair wise dependency rules defined according to MOF's SS, FF, SF, and FS synchronizations. (We will describe the workflow construction in more detail later.)

These generated structures are passed to an XML composer which performs rudimentary type checking on the instances to ensure instances are only present if there is also a class, and re-aggregates the XML fragments back into a whole XML documents. At this point, there may be multiple XACCT component description documents, but there is only one instantiation+workflow document containing all data necessary for a single deployment.

Next, ACCT forwards each XACCT component description document and the XACCT instantiation and workflow document to the XSLT engine. In the engine, the XSLT templates detect the XACCT document type and generate the appropriate files (SmartFrog or Java) which are written to disk.

Using the approach of the XML-to-XACCT phase mentioned above, components, configurations, constraints, and workflows from input languages of any resource management tool can be described in the intermediate XACCT representation. Once an input language is mapped to the XACCT, the user merely creates an XSLT template perform the final mapping of the XACCT to the one of the specific target languages.

Purely syntactic differences between MOF/CIM and SmartFrog's language can be resolved using only the XSLT processor, and the first version of ACCT was developed on XSLT alone. However, because XSLT has certain limitations, we incorporated a Java/DOM pre-processing stage (XML-to-XACCT). At first, the XSLT only version was limited to a single output file, but as newer XSLT standards versions enable multi-file output, this was not available during initial development. This was solved by generating to several smaller documents. Future plans are to incorporate all XACCT XML into a single document. A second, and more important, reason is that XSLT is limited in its capability to compute workflows from the partial dependencies Cauldron supplies. This is because XSLT, again at the time we began work on ACCT, did not support on-the-fly computation of new XML structures that could be used as in-program storage. We addressed this limitation with the Java/DOM pre-processing stage that computed the overall event ordering.

Overall system ordering derives from the Cauldron computed partial synchronizations encoded in the input MOF. As mentioned in Section 3.1, there are four types of partial synchronization dependencies: SS, FF, SF, and FS. To describe the sequential and parallel ordering of components with only these partial dependencies, we implemented an event queue model with an algorithm that synchronizes activities correctly. It is helpful to consider this process as that of building a graph in which each component is a node and each dependency is an edge in the graph. Each activity component has one associated EventQueue containing list of actions:

Execute - the action to execute a specific sub component.

EventSend - the action to send a specific event to other components' EventQueues. This may accept a list of destination components.

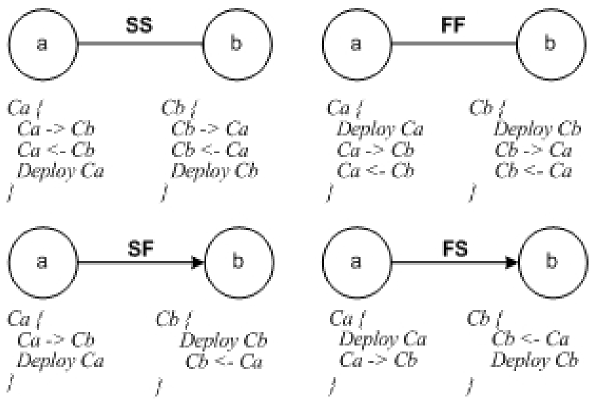


Figure 2. Diagrams and dependency formulations of (a) SS, (b) FF, (c) SF, and (d) FS.

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.

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

Table 1. Event dependencies between components.

To pass the execution turn to another activity component, an activity component sends a message as an event to a second activity component which waits, blocking, for an event in its EventQueue.

In Figure 2(a), since two activity components must start to deploy their sub-components at the approximately same time as the definition mentioned in Section 2.1, two activity components are blocked until each event receives a notification. This is achieved by making entries into the EventQueues to send then wait for events from the peered components. Similarly, Figure 2(b) illustrates the FF scenario. In Figure 2(c), since C_b's deployment must be finished after C_a starts to deploy, C_b is blocked in its EventQeue after completing deployment until the event of C_a is received at C_b's EventQueue. In Figure 2(d), since C_b may deploy only after C_a completes its task, C_b blocks until a SmartFrog "finished" event from C_a is received at C_b's EventQueue. For now, we assume the network delay of each direction between two components is ignorable.

```
<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>
```

Figure 3. XACCT snippet of FS dependency.

Figure 3 illustrates the XACCT for the FS dependency described in (d). The SS and FF operations of (a) and (b) represent the parallel deployment while the SF and FS of (c) and (d) represent the sequential deployment.

The exact content of each EventQueue depends on its dependencies to all other activity components. Combinations of actions used in four partial dependencies described in Figure 2 are the simplest cases. However, each activity component frequently has multiple dependencies. We have devised an algorithm to calculate EventQueue contents.

First, the algorithm visits the each activity component, C_i, in the CIM-XML document and builds a global action list. If a dependency of the component is a parallel dependency (i.e., SS or FF), then the algorithm transitively examines checks for dependencies of the same type on the related activity component until it finds no more parallel dependencies of that type. 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, it records "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 (i.e., FS or SF), then, the algorithm adds the dependency to the global action list and moves on to the next component. That is, if C_i has FS with C_j, and C_j has FS with C_k, only the pairwises like "C_i and C_j with FS" are entered into the global action list.

For deadlock avoidance, there is a static order of actions in each activity component based on the activity component's role (Antecedent or Dependant) in each dependency. The algorithm checks the six possible combinations of roles and dependencies as follows.

First, it checks whether the activity component participates as a Dependant of any FS dependency. If this is true, then it adds one OnEvent action to the EventQueue per each FS-Dependant dependency to the combination list.

Second, it checks whether the activity component has any SS dependencies. For SS, it adds all EventSend and OnEvent actions to the EventQueue.

Third, it checks whether the activity component functions as Antecedent in SF dependencies. Per dependency, it adds an EventSend action to the EventQueue.

Next, an Execute action is added to EventQueue. Fourth, it checks whether the activity component participates as a Dependant in an SF dependency. This maps to one OnEvent action per dependency.

Fifth, it checks whether the activity component has any FF dependencies and adds all EventSend and OnEvent actions to the EventQueue.

Sixth, it checks whether the activity component works as any Antecedent roles with FS. If so, it adds EventSend actions per occurrence.

Finally, Terminate action is appended to the EventQueue.

XACCT captures the final workflow as the set of per-component EventQueues, and those then translated 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. We also employ an HTTP server as a repository to store some scripts and application source files. Once a generated SmartFrog description is fed to the SmartFrog workflow daemon, it spawns threads to start all activities in the workflow simultaneously. From there synchronization among activities is controlled by the EventQueues.

3. Demo Application and Evaluation

We present in this section how the toolkit described in Section 2 automatically generates the system configurations and the workflow, automatically translate both the configurations and the workflow into the input of SmartFrog used as a deployment environment, and automatically deploys distributed applications with various complexities. In the sub section 4.1, we describe 1-, 2-, and 3-tier testbeds used in our experiment, and system setup for the experiment. We evaluate in the sub section 3.2 our toolkit by showing the actual result code of each phase of toolkit, and comparing the deployment execution time of SmartFrog with the automatically generated code to the manually written scripts.

3.1 Experiment Scenario and Setup

We evaluated our translator by employing it on 1-, 2-, and 3-tier applications. The 1- and 2-tier applications are simple tests that provide a baseline for comparing a generated SmartFrog description to handcrafted scripts. The 3-tier testbed comprises the web server, an application server, and the database server; it is of small enough size to be easily testable, but also has enough components to indicate the power of the toolkit in managing complexity. Table 2. Components of the 1-, 2-, and 3-tier applications.
lists the applications' components.

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

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

We installed SmartFrog 3.04.008_beta on four 800 MHz dual-processor Dell Pentium III machines running RedHat 9.0 for the evaluation. In order to evaluate the deployment of applications, we needed to run four separate SmartFrog daemons; one daemon runs on each machine.

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

In the 2-tier Hit Counter application, Apache and Tomcat application server with Ant were used in the 2-tier testbed. Each tier was deployed on a separate host. To verify the 2-tier deployment, we visited the web page to ensure it properly recorded page hits. The application simply consists of a class and a `jsp` page. In the 2-tier application evaluation, we used 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 and Tomcat).

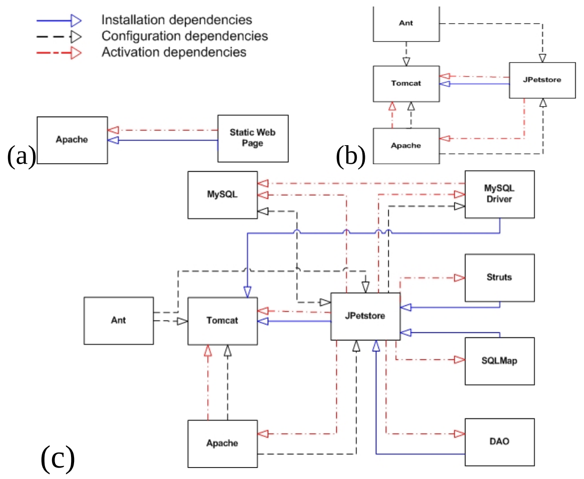


Figure 4. Dependency diagrams of (a) 1-tier application, (b) 2-tier application, and (c) 3-tier application.

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.

Figure 1 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 the, 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 properly sequenced. For instance, the Apache configuration must start after Apache installation completes, and Apache activation must start after Apache configuration completes for the 1-tier testbed. For space, we have omitted these dependencies from the figure.

3.2 Experiment Result

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. Figure 8 illustrates part of the transformation process as ACCT translates the MOF file of the 3-tier application to intermediate XACCT and then finally to a SmartFrog description. We especially highlight the FS dependency between the Tomcat installation and MySQLDriver installation, and related configurations.

The metric we choose for the 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 obtained averages.

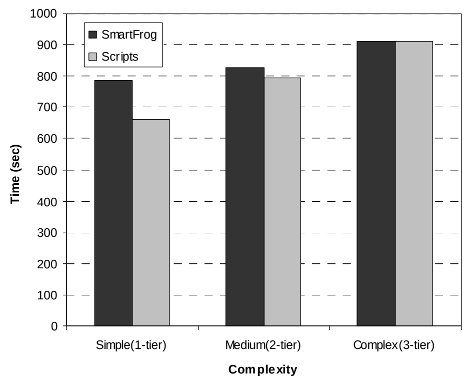
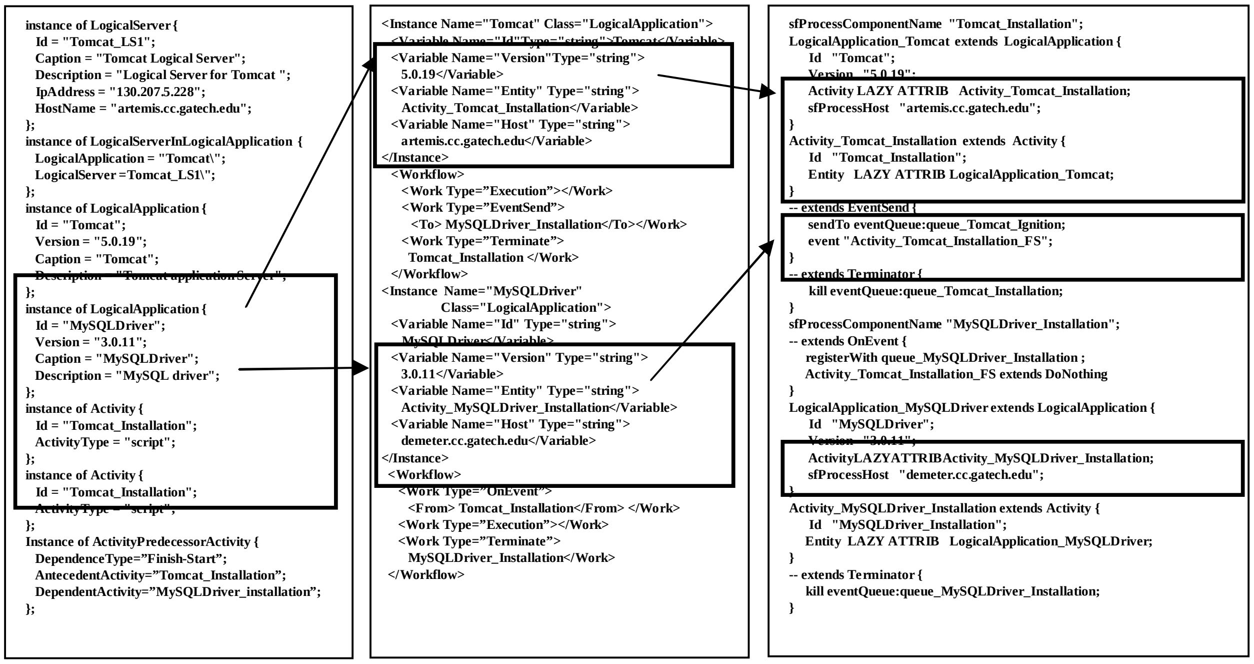


Figure 5. Deployment time using SmartFrog and scripts as functions of the complexity.

Figure 5 shows that for simple cases (1- and 2-tier) SmartFrog took longer when compared to the scripts based approach because SmartFrog daemons need the Java VM and environment and impose extra costs when loading Java classes or engaging in RMI communication. Note, however, the time penalty of the medium case is less in absolute and relative terms than the one of the simple case. In the very complex case, SmartFrog took less time than scripts based approach.

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



```
instance of LogicalServer {
  Id = "Tomcat_LS1";
  Caption = "Tomcat Logical Server";
  Description = "Logical Server for Tomcat ";
  IpAddress = "130.207.5.228";
  HostName = "artemis.cc.gatech.edu";
};
instance of LogicalServerInLogicalApplication {
  LogicalApplication = "Tomcat\";
  LogicalServer =Tomcat_LS1\";
};
instance of LogicalApplication {
  Id = "Tomcat";
  Version = "5.0.19";
  Caption = "Tomcat";
  Description = "Tomcat application Server";
};
instance of LogicalApplication {
  Id = "MySQLDriver";
  Version = "3.0.11";
  Caption = "MySQLDriver";
  Description = "MySQL driver";
};
instance of Activity {
  Id = "Tomcat_Installation";
  ActivityType = "script";
};
instance of Activity {
  Id = "Tomcat_Installation";
  ActivityType = "script";
};
Instance of ActivityPredecessorActivity {
  DependenceType="Finish-Start";
  AntecedentActivity="Tomcat_Installation";
  DependentActivity="MySQLDriver_installation";
};
```

(a) MOF

```
<Instance Name="Tomcat" Class="LogicalApplication">
  <Variable Name="Id"Type="string">Tomcat</Variable>
  <Variable Name="Version"Type="string">
    5.0.19</Variable>
  <Variable Name="Entity" Type="string">
    Activity_Tomcat_Installation</Variable>
  <Variable Name="Host" Type="string">
    artemis.cc.gatech.edu</Variable>
</Instance>
  <Workflow>
    <Work Type="Execution"></Work>
    <Work Type="EventSend">
      <To> MySQLDriver_Installation</To></Work>
    <Work Type="Terminate">
     Tomcat_Installation </Work>
  </Workflow>
<Instance  Name="MySQLDriver"
            Class="LogicalApplication">
  <Variable Name="Id" Type="string">
    MySQLDriver</Variable>
  <Variable Name="Version" Type="string">
    3.0.11</Variable>
  <Variable Name="Entity" Type="string">
    Activity_MySQLDriver_Installation</Variable>
  <Variable Name="Host" Type="string">
    demeter.cc.gatech.edu</Variable>
</Instance>
 <Workflow>
   <Work Type="OnEvent">
     <From> Tomcat_Installation</From> </Work>
   <Work Type="Execution"></Work>
   <Work Type="Terminate">
     MySQLDriver_Installation</Work>
 </Workflow>
```

(b) XML

```
sfProcessComponentName  "Tomcat_Installation";
LogicalApplication_Tomcat extends LogicalApplication {
    Id  "Tomcat";
    Version  "5.0.19";
    Activity LAZY ATTRIB   Activity_Tomcat_Installation;
    sfProcessHost   "artemis.cc.gatech.edu";
}
Activity_Tomcat_Installation extends Activity {
    Id  "Tomcat_Installation";
    Entity  LAZY ATTRIB LogicalApplication_Tomcat;
}
-- extends EventSend {
    sendTo eventQueue:queue_Tomcat_Ignition;
    event "Activity_Tomcat_Installation_FS";
}
-- extends Terminator {
    kill eventQueue:queue_Tomcat_Installation;
}
sfProcessComponentName "MySQLDriver_Installation";
-- extends OnEvent {
   registerWith queue_MySQLDriver_Installation ;
   Activity_Tomcat_Installation_FS extends DoNothing
}
LogicalApplication_MySQLDriver extends LogicalApplication {
    Id  "MySQLDriver";
    Version  "3.0.11";
    ActivityLAZYATTRIBActivity_MySQLDriver_Installation;
    sfProcessHost   "demeter.cc.gatech.edu";
}
Activity_MySQLDriver_Installation extends Activity {
    Id  "MySQLDriver_Installation";
   Entity  LAZY ATTRIB   LogicalApplication_MySQLDriver;
}
-- extends Terminator {
   kill eventQueue:queue_MySQLDriver_Installation;
}
```

(c) SmartFrog

Figure 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.

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 range of resource management systems. For e-business, OGSA Grid Computing [15] aims to provide services within a data center infrastructure that provides resources on demand. IBM's Autonomic Computing Toolkit [16], HP's Adaptive Enterprise and MicroSoft's DSI initiative [3].

Another trend is deployment automation tools. CFengine [21] 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 [20] 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 architecture adopts the architecture developed for the Infopipe Stub Generator + AXpect Weaver (ISG) [6] in that both of them utilize an XML intermediate format that is translated by XSLT to target source code. Unlike ACCT, however, ISG is oriented toward communication stubs for information flow systems. There are other commercial and academic translation tools, like MapForce [22] and CodeSmith [23]. Similar to ISG, they target general code generation and do not support deployment workflows.

5. Conclusion

We described an approach for Automated Deployment. We described in detail ACCT (Automated Composable Code Translator) that translates Cauldron output (in CIM/MOF format) into SmartFrog specification input format. A demonstration application (JPetStore) illustrates the automated design and implementation process and translation steps, showing the advantages of such automation.

6. References

[1] IBM Autonomic Computing,
http://www.ibm.com/autonomic
[2] SUN N1, http://wwws.sun.com/software/solutions/n1/
[3] Microsoft DSI, http://www.microsoft.com/management/
[4] Global Grid Forum, http://www.ggf.org
[5] SmartFrog, http://www-uk.hpl.hp.com/smartfrog/

[6] Galen Swint and Calton Pu, "Code Generation for WSLAs using AXpect", IEEE International Conference on Web Services, 2004.
[7] Salle, M., Sahai, A., .C. Bartolini, S. Singhal, "A Business-Driven Approach to Closed-Loop Management", HP Labs Technical Report HPL-2004-205, November 2004.
[8] Sahai A, Singhal S, Joshi R, Machiraju V, "Automated Policy-Based Resource Construction in Utility Computing Environments", In the proceedings of NOMS 2004.
[9] Sahai A, Singhal S, Joshi R, Machiraju V. "Automated Generation of Resource Configurations through Policies", IEEE Policy 2004.
[10] Goldsack, P., et al., "Configuration and Automatic Ignition of Distributed Applications", HP Openview University Association conference, 2003.
[11] Peterson, L, et al., "A Blueprint for Introducing Disruptive Technology", PlanetLab Tech Note, PDN-02-001, July 2002.
[12] CDDLM Charter Document,
https://forge.gridforum.org/projects/cddlm-wg
[13] Smartfrog open source directory,
http://www.smartfrog.org
[14] Open source of WBEM project,
http://wbemservices.sourceforge.net
[15] Foster, I. et al., "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration."
[16] Codesmith, http://www.ericjsmith.net/codesmith
[17] HP Utility Data Center,
http://www.hp.be/egov/en/solutions/aoii_data_center.asp
[18] DMTF-CIM Policy,
http://www.dmtf.org/standards/cim/cim_schema_v29
[19] PARLAY Policy Management,
http://www.parlay.org/specs/
[20] Eelco Dolstra, Merijn de Jonge, and Eelco Visser, "Nix: A safe and policy-free system for software deployment", In Proceeding of the Eighteenth Large Installation System Administration Conference, Atlanta, Georgia 2004.
[21] Cfengine, http://www.cfengine.org/
[22] Altova Mapforce,
http://www.altova.com/products_mapforce.html
[23] Codesmith, http://www.ericjsmith.net/codesmith