2nd International Workshop
Model-Driven Engineering

FOR HIGH PERFORMANCE AND CLOUD COMPUTING (MDHPCL)

Workshop organized as satellite event of the conference MODELS 2013, Miami, Florida, USA, September 29, 2013

Workshop Organizers

Ileana Ober, IRIT - University of Toulouse, France
Jean-Michel Bruel, IRIT - University of Toulouse, France
Akshay Dabholkar, Nimbula Inc, USA
Aniruddha Gokhale, ISIS, Vanderbilt University, USA
James Hill, Indiana University-Purdue University Indianapolis, USA
Michael Felderer, University of Innsbruck, Austria
David Lugato, CEA, France

Program Committee

Jean-Michel Bruel, IRIT - Univ of Toulouse, France
Akshay Dabholkar, Oracle, USA
Dirk Draheim, University of Innsbruck, Austria
Steven Drager, Air Force Research Laboratory, USA
Robert France, Colorado State University, USA
Michael Felderer, University of Innsbruck, Austria
Aniruddha Gokhale, Vanderbilt University, USA
David Lugato, CEA, France
William McKeever, Air Force Research Labs, USA
Ileana Ober, IRIT - University of Toulouse, France
Bernhard Rumpe, RWTH Aachen University, Germany

Table of content

$2^{nd}MDHPCL$: Model-Driven Engineering for High Performance and CLoud computing

Ileana Ober[1], Aniruddha Gokhale[2], James Hill[3], Jean-Michel Bruel[1], Michael Felderer[4], David Lugato[5], and Akshay Dabholkar[6]

[1] University of Toulouse - IRIT, France
[2] Vanderbilt University, USA
[3] Indiana University Purdue University at Indianapolis, USA
[4] University of Innsbruck, Austria
[5] CEA, France
[6] Oracle, USA

1 Objectives and Scope

The important vitality of IT, recently increased the focus on technologies such as cloud computing, high performance applications and parallelism architectures. Industry needs help from the research community to succeed in its recent dramatic shift ". None of these technologies are traditional users of modeling approaches. However, some results start to emerge on the use of modeling techniques as a mean to help addressing issues related to the complexity of applications in these fields, the need for separation of concerns, or the demand for abstracting from platform concerns.

One of the major common points between High Performance Computing and Cloud Computing is that the two technologies aim at a solution that allows to offer as the simplicity of regular desktop tools, while based on the power of massively parallel computing, respectively complex data architectures and their management. In both cases, the ultimate goal is about maximizing human productivity, by allowing non-experts to create and evaluate complex models quickly and easily. There are several ways to achieve this: by raising the level of abstraction used in the design and development of the application, by decoupling application specific from optimization driven choices, by ensuring a better separation of concerns etc. The above mentioned strategies correspond to techniques largely used and promoted by Model-Driven Engineering.

The purpose of this workshop is to bring together researchers and practitioners, with experience in HPC, cloud computing and MDE and to explore the strategies that would allow a wider use of MDE techniques in the above mentioned application fields.

In spite of the obvious need for abstraction in HPC and cloud computing, the use of MDE is marginal. This is partially due to subjective reasons, such as the lack of MDE experts amongst the HPC and cloud computing teams, insufficient communication between the two communities, reticence of the teams used to intensively combine hardware specific code with application specific code for

optimization reasons. However, there are more fundamental obstacles in using MDE, such as the problems in the scalability of the existing MDE solutions, the insufficient support for collaborative work, etc.

This workshop aims to present existing work on applying modeling techniques in cloud and high performance computing. Beyond offering a forum for current and ongoing work on these topics, the workshop aims to open a discussion on the challenges faced by these new fields, and how model-driven techniques could be adapted to meet these challenges. We intend to find individual success stories on the use of modeling on the fields addressed by this workshop and discuss on the factors that may have contributes to the positive outcome, the difficulties faced and how they were addressed.

This workshop targets researchers and practitioners who work in the area of HPC or Cloud Computing and who feel the use of modeling techniques can be beneficial to their respective fields, as well as to researchers and practitioners in in the modeling area who have an interest in adapting their work to new application domains.

2 Workshop Venue, Date and Program Committee

The workshop [1] was held as part of the IEEE/ACM MODELS 2013 conference on September 29, 2013 in Miami Beach, FL, USA.

The program committee of this workshop comprised:

- Jean-Michel Bruel, IRIT - University of Toulouse
- Akshay Dabholkar, Nimbula, Inc., USA
- Dirk Draheim, University of Innsbruck, Austria
- Steven Drager, Air Force Research Laboratory, USA
- Robert France, Colorado State University
- Michael Felderer, University of Innsbruck
- Aniruddha Gokhale, ISIS, Vanderbilt University
- James Hill, Indiana University Purdue University at Indianapolis
- David Lugato, CEA, France
- William McKeever, Air Force Research Labs, USA
- Ileana Ober, IRIT - University of Toulouse
- Bernhard Rumpe, RWTH Aachen University

3 Why a Second Edition?

This is the second edition of a workshop that focuses on the use of modeling in HPC and Cloud Computing. To our knowledge there was no other workshop on the same topic. The closest to our interest is the workshop *ICSE 2011 Software Engineering For Cloud Computing Workshop*. This workshop proved that there is an important community receptive to applying software engineering techniques in cloud computing. There are two main differences between our

[1] `http://www.irit.fr/MDHPCL2013`

workshop and the workshop held at ICSE: we intend to focus on the use of model-driven techniques (and not on software engineering issues in general) and we would like to look closer to their applicability to both Cloud Computing and High-Performance Computing.

The number of submissions received for the first edition of our workshop was not high (8), however the workshop attracted about 25 participants and generated a lot of discussions. During the discussion the participants asked explicitly for a second edition of the workshop. Our analysis of this situation is that the topic addressed by the workshop is of interest for a lot of people, there are many emerging efforts in this area, although not many are mature enough to generate submissions. Moreover, these topics are also increasingly important to the MODELS community as can be seen in the call for papers.

4 Selected Papers and Workshop Logistics

Two members of the organizing committee, Aniruddha Gokhale and James Hill, moderated the proceedings of the workshop.

Prior to the day of the workshop, we made pre-proceedings of the workshop available online for participants and `http://www.irit.fr/MDHPCL2013/Proceedings_files/preproceedings_mdhpcl2013.pdf`.

A total of 7 papers were selected to be presented of which 6 were long papers and 1 short paper. The long papers got 20 mins (15 + 5 for Q&A); short papers got 15 mins (12 + 3 for Q&A). Attendees were requested to hold off the most interesting questions to the end for the panel discussion.

This half-day workshop selected the following papers for presentation at the workshop:

1. Towards a Solution avoiding vendor lock-in to enable Migration between Cloud Platforms
2. Modeling Cloud Architectures as Interactive Systems
3. VehicleForge: A Cloud-based Infrastructure for Collaborative Model-based Design
4. A Model-driven Approach for Price/Performance Tradeoffs in Cloud-based MapReduce Application Deployment (short paper)
5. Towards Domain-specific Testing Languages for SaaS
6. Architecture Framework for Mapping Parallel Algorithms to Parallel Computing Platforms
7. Model-driven Transformations for Mapping Parallel
8. Algorithms to Parallel Computing Platforms

There were about 20 or more members in the audience. Each paper evoked significant interest in the audience giving rise to some fruitful discussions. At the end of the paper presentations, the audience discussed many long term ideas. There was enough interest demonstrated by the audience that will justify a third edition of this workshop at MODELS 2014.

Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms

Alexandre Beslic ‡, Reda Bendraou ‡, Julien Sopena ‡, Jean-Yves Rigolet †

Department of Computer Science and Engineering, Pierre and Marie Curie University, 4 place Jussieu 75017 Paris, France ‡
{alexandre.beslic,reda.bendraou.julien.sopena}@lip6.fr

IBM, 9 rue de Verdun 94250 Gentilly, France †
rigolet.j@fr.ibm.com

Abstract. The *Cloud Computing* paradigm is used by many actors, whether companies or individuals in order to harness the power and agility of remote computing resources. Because they target developers and offer a smooth and easy way to deploy modern enterprise software without dealing with the underlying infrastructure, there is a steadily increasing interest for Platforms as a Service (PaaS). However, the lock-in makes the migration to another platform difficult. If the vendor decides to raise its prices or change its security policies, the customer may have to consider to move to the competition or suffer from these changes. Assistance and tooling to move to the competition at the PaaS layer still does not exist thus requiring tremendous re-engineering effort. In this regard, we propose an approach to the challenge of software migration between PaaS using Model-Driven Engineering coupled to Program Transformation

1 Introduction

Cloud Computing is now a popular paradigm, offering computing resources on a "pay-as-you-go" basis. It allows a remote and on-demand access to a wide range of services alleviating the need to own and maintain an internal infrastructure. The service model is standardized by the NIST [16] and is divided into three major layers. These layers vary in the amount of abstraction they provide to the consumer. The more you climb this service model, the more you will face restrictions.

Infrastructure as a Service (IaaS) provides the ability for consumers to provision fundamental computing resources such as processing power, storage capacity or networks. They have control over the operating system and software stack giving them the freedom to deploy any kind of software. *Platform as a Service* (PaaS) came as an abstraction to the infrastructure layer. Because maintaining and updating a whole infrastructure requires knowledge and time, platform provides with a fully prepared runtime environment to deploy applications. It targets developers to further fasten the development process and to focus on the

product features rather than configuring the underlying infrastructure. *Software as a Service* (SaaS) is the highest level of the Cloud service model. The software itself is provided as a service to the end-user.

While Infrastructure as a Service (IaaS) and Software as a Service (SaaS) are still prevalent in the Cloud computing service model, Cloud platforms (PaaS) are becoming increasingly used. According to the Gartner study on Cloud Computing, the use of Cloud Platforms will increase at 43% in 2015 compared to 3% in 2012. With a major struggle between cloud providers to dominate the PaaS market, the use case of software migration between providers is to be considered. But this task is far from being easy. Indeed, the platform layer suffers from a well known issue: the vendor *lock-in*. Early platforms are providing tools and libraries to use during the development process to access their own features thus locking the application to this platform. The advent of NoSQL solutions with data denormalization makes it even more difficult because of choices made on the program's design to ensure best performance. As a consequence, migrating onto another platform requires tremendous re-engineering effort that a few are able to provide.

The will to migrate is explained by several factors. The price is the first one considering that computers are now a commodity that we need at the lowest price, thus explaining the popularity of the Cloud Computing paradigm. Some other factors are the *Lock-in avoidance*, an *Increased Security*, a *Better availability* (99,95% versus 100%), a *Better Quality of Service* (QoS guarantee), a *Major shift in technology trends* or *Legal issues* (forced to move) among others. As of today, no such tool exists to achieve this migration. Existing work like mOSAIC [19] is taking the approach of the middleware abstracting cloud software stacks or APIs. mOSAIC offers a thin layer of abstraction over software stacks and data storage on PaaS in order to develop applications from scratch. Thus it only supports newer applications and the user is still entangled by the compatibility list of the middleware. Even if it tries to support a wide variety of Cloud providers hence being a first step for Cloud platform interoperability, the use of a middleware just moves around the lock-in and businesses are reluctant to this.

In this regards, we present our approach to deal with this major issue of software migrations between Cloud Platforms. The idea is to provide assistance and tooling to re-engineer a software using the Model-Driven Architecture and Refactoring approach. It is divided in several stages. The first one is the discovery of a Cloud software deployed on a platform using MoDisco [12]. Follows the Transformation on the program structure/layout using Model transformation on the discovered model. Then fine grained transformations are defined between an API and its counterpart on the targeted platform using a general purpose transformation language such as TXL. Assistance on software transformation is provided by applying these rules. The final step is the migration on the targeted platform with offline and online tests suites validation to be sure that the software runs as usual. In order to achieve this, as the number of scenarios is huge (especially for the data storage part) and the cloud environment always evolv-

ing, we aim to provide the ability to add new knowledge for processing source to target transformations using a dedicated DSL.

The paper is organized as follows. In section 2, we give examples of challenges to be tackled when migrating a software tied to the provider's data storage solution or APIs. In section 3, we introduce our detailed approach to deal with software migration between Cloud platforms. Section 4 discusses related work and the limits of using middlewares to deal with this issue. Section 5 concludes the paper and discusses future work.

2 Vendor lock-in issue

The Platform as a Service (PaaS) appeared shortly after the Infrastructure (IaaS) and Software (SaaS) layers of Cloud Computing. Heroku, which has been in development since 2007 is a owned subsidiary of Salesforce.com and is one of the very first Platform as a Service provider. It provides a fully prepared stack to deploy automatically Ruby, Node.js, Clojure, Java, Python and Scala applications and runs on top of Amazon Web Services (AWS). Since then many new providers have entered the market as Google with its App Engine (GAE) platform or Microsoft with Windows Azure both in 2008. These are three of the very early platform providers but there are many others as of today creating a large ecosystem of PaaS solutions. The particularity of Cloud platforms is that every provider has its own set of features, frameworks or language supported.

With those early Cloud platforms, the customer is using a well defined set of tools and libraries originating from the provider. Achieving best performance is a result of using the providers data storage solution, not supported on other platforms. Google App Engine is using BigTable while Windows Azure is using Azure Table to store data. Both are categorized as NoSQL solutions meaning that they differ drastically from classical RDBMS as MySQL or PostgreSQL which are relational.

Both have a strikingly similar data structure, even if they have also subtleties in their design like the way they handle fragmentation on servers, thus impacting the design of the program. Both are schema-less and both are using two keys with a timestamp to quickly query data using a distributed binary index. Both are also categorized as Key-Value stores in their documentation (even if the right definition for BigTable as defined in the white paper is a sparse, distributed, persistent multi-dimensional sorted map [13]). Achieving best-performance on both systems requires effort and knowledge. Thus and because they are exotic in design compared to other platforms supported databases, you end-up being locked in. These types of NoSQL databases are not represented elsewhere as a true Storage as a Service offering. Your choices are to use similar solutions like DynamoDB from Amazon, HBase (the open source implementation of BigTable) on EMC (an IaaS provider), or try a migration from Azure Table to BigTable or vice versa. The fact is that DynamoDB and HBase are not included as built-in PaaS features but only at the Infrastructure layer. Moreover, they have different properties in the way they handle the data compared to BigTable or Azure

Table. The data denormalization broadens the possibilities but trying to move from a solution to another is a difficulty introduced recently with the advent of NoSQL databases. Every NoSQL is used for a purpose and differs slightly or completely from another solution. But as they offer the scaling properties that relational databases couldnt offer, they are mandatory on PaaS to leverage best performance and scalability.

Considering those design decisions, what if one of these two platforms on which you deployed your application decides to raise its prices or do whatever you disagree with as a customer? Either you accept these changes or you consider moving onto another Cloud. But this will require tremendous efforts to adapt your software that is locked-in by specific APIs and data storage proprietary implementations.

As the lock-in became a sore point for customers willing to move on a platform, companies are now taking the bet to offer *Open Platforms as a Service*. Initiatives like TOSCA [7] to enhance the portability of Cloud applications have been supported by several partners like IBM, Red Hat, Cisco, Citrix and EMC. Red Hat with *OpenShift*, VMWare with *CloudFoundry* and IMBs *SmartCloud Application Services* are three of the most known projects for portable PaaS solutions. As of writing, CloudFoundry is still in beta while OpenShift has been released. The idea behind Open PaaS is that being restricted to a framework or library to develop an application is not offering the flexibility desired by the developers. Instead they offer the widest range of languages, frameworks and data storage to vanish the lock-in still present on older Platforms. These platforms are extensible with new technologies or patterns (in the case of IBMs PaaS solution).

Still older Platforms have their benefit. Because the architecture is mature and that they improve by offering the latest technologies to attract new customers. There are a lot of examples of successful websites with a huge amount of requests per day while newer platforms lacks such examples that could appease new customers in their choice. Moreover, with the example of OpenShift, the support is large but still limited to older versions of languages such as Python used in its 2.6 version (while there is a 2.7 and 3 version of the language supported on many other platforms). It also misses built-in services present in other PaaS like Redis, Memcached or RabbitMQ (some of them are available on carts, which are pluggable components on OpenShift but are not straightforward to use). Also proprietary implementations of database systems are still more scalable and BigTable is known to scale to petabytes of data with highly varied data size and latency requirements which makes it particularly powerful for high traffic websites. Given the diversity of configurations for developed applications, Open platforms are not the general response to the vendor lock-in problem. Because the lock-in is traded off with the support of newer technologies and possible increased availability and security on other platforms. A dominant Platform crushing the competition is unlikely to happen because of the wide range of customers needs. As a consequence: *How to offer the possibility to migrate from a platform to another considering this ecosystem of PaaS providers?*

While on the infrastructure layer, the migration was restricted in the study of moving Virtual Machines from an Hypervisor to another [11], here the issue is much larger. Because the diversity of modern applications is huge and that finding a "one-fits-all" answer is rather impossible. At least a way could be found to assist the shift from a platform to another as for the migration of legacy software to the Cloud.

3 Our Approach

In this section, we describe the details behind our approach to migrate applications between Cloud platforms. Given the similarities that could be found on technologies of the same kind, we could define source to target transformations by leveraging knowledge amongst the Cloud developer community as well as Cloud vendors themselves. We strongly believe that assisting the migration of applications between Cloud platforms using the re-engineering approach is more flexible than the middleware approach because of the independence to any intermediate technology that could be harmful in the future, in terms of performance and support.

3.1 Overall Design

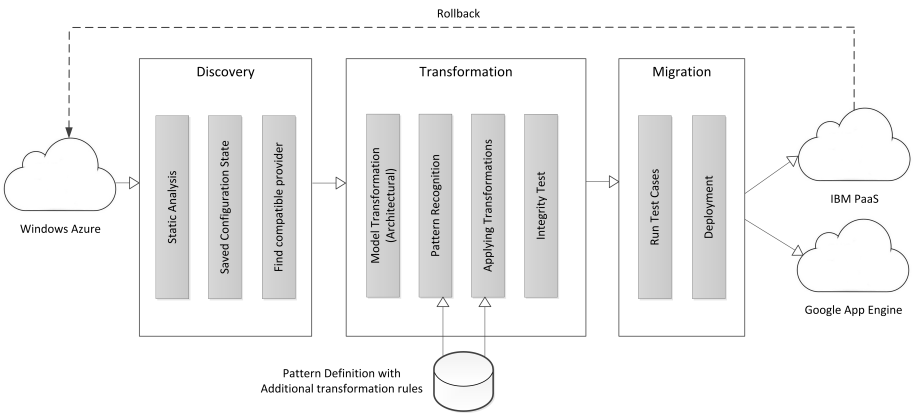


Fig. 1. PaaS migration workflow

Figure 2 shows our approach which contains three steps in the lifecycle of a software about to be migrated on a new platform: *Discovery*, *Transformation* and *Migration*. It also contains pluggable components: *Pattern Definition with Additional transformation rules*.

The *Discovery* step includes the discovery on sources taken back from the source platform. Because platforms generally have version control deployment options using git, getting the sources is straightforward at this step. By having a high level representation of sources, we have all the informations to transform and adapt the software accordingly at an architectural level.

Transformation step is going to adapt the software regarding the targeted platform. It includes a Model Transformation step to address architectural (coarse grained) violations that could prevent a successful migration. Because fine grained transformations on sources from a given pattern to its counterpart on the targeted platform are necessary as well, we include a Pattern recognition step. Patterns are defined portions of code with a given semantic. Every pattern comes with its counterpart for the targeted platform and mappings between method calls and attributes are made to bridge the two representations. These mappings are part of the *Additional transformation rules* that are provided by Cloud users with a dedicated *Domain Specific Language*.

Migration is the last phase of the process, it tests the transformed application prior to the deployment on the new platform and gets feedbacks at runtime. If nothing goes wrong in the deployment and at runtime, we take back resources on the ancient provider. But if something goes wrong after the resources are taken back, rollback is applied on the ancient platform with the help of saved sources and configurations. Specific insights on each phase are covered on subsections.

Discovery phase As we are treating of the use case of migration between platforms, we assume that an existing project is already deployed on a PaaS. Whatever the provider, as long as the sources are accessible by any mean, the process can go further. With these sources, we get back a higher level representation of the software. For this, we use MoDisco [12] which is a reverse engineering tool (the term of discoverer is more appropriate) that traduces sources to UML models or even an Abstract Syntax Tree (AST). MoDisco is extensible by new definitions of legacy software to discover. However in our case, the OMG's Knowledge Discovery Metamodel (KDM) [18] already provides the required metadata for the source discovery. KDM is a standard providing ontology (a set of definitions) for system knowledge extraction and analysis. This *KDM discoverer* is a built-in feature of MoDisco which takes away the complexity of the process. The target KDM model gives insights of the technology used by the application. It represents the project structure and overall design. At this state, the target model as well as configuration options are saved for future purposes (Rollback which is further explained in the migration section).

For a given configuration we could guide the user to a new platform that has the same kind of technologies to reduce the work on the transformation phase. This could be done by defining a configuration pattern for each provider and match our configuration among these definitions. This choice could also be guided by the price and other parameters taking inspirations from customer needs. Given the diversity of platform providers, a right behavior is to forbid the choice to a target provider with a completely different language with an opposed data representation. Such a case will make the transformation step weaker and likely to break the integrity of the sources. Otherwise, we could also find a perfect matching configuration for our software. Not every software are locked in by a specific API or data solution (especially on an Open Platform which promote

the use of open source software) thus enabling an easy migration to another provider.

```
1  // Create a new student entity.
2  StudentEntity student1 = new StudentEntity("Foo", "Bar");
3  student1.setEmail("foo.bar@upmc.com");
4  student1.setPhoneNumber("0102030405");
5
6  // Create Insert operation
7  TableOperation insertStudent1 = TableOperation.insert(student1);
8
9  // Submit the operation
10 tableClient.execute("people", insertStudent1);
```

Listing 1.1. New entity on Azure Table

```
1  // Create a new student entity
2  Entity student1 = new Entity("Student");
3  student1.setProperty("name", "Foo");
4  student1.setProperty("lastName", "Bar");
5  student1.setProperty("email", "foo.bar@upmc.com");
6  student1.setProperty("phoneNumber", "0102030405");
7
8  // Add the student entity
9  datastore.put(student1);
```

Listing 1.2. New entity on BigTable

Transformation phase Transformation process takes place after the Discovery step and benefits from the KDM model discovery. KDM provides a set of abstraction layer to understand existing softwares. There are three main layers:

- *Program Elements Layer*: Code primitives and actions
- *Runtime Resource Layer*: Data, Events, UI and Platform
- *Abstractions Layer*: Conceptual, Build and Structure

All these layers are helpful to define the context of the migration, giving us insights of the specific code portions, frameworks and APIs used, to higher level details about runtime environment. The KDM model is used for two goals: finding the best target platform and apply architectural changes to the project. By architectural changes, the idea is to address project varying layouts forced by some frameworks (mostly for web applications). Examples are project layouts of Struts, JSF or the webapp lightweight framework of Google App Engine. Changes to the project configuration and deployment scheme are guided to comply to the new provider as each offer different options of deploying on their platform.

However, these architectural changes are not sufficient to adapt the software for the new platform. As such, our approach focuses on the use of pattern-based techniques so that fine grained transformations are realized on the sources. Patterns are defined as a provider's code portion with a meaningful semantic. As every pattern comes with an associated transformation definition bridging the

methods and attributes to their counterparts in the targeted platform, transformation are made in a fine grain fashion. Thus the architectural changes mixed to the program transformation depicts an hybrid way to adapt an application to be migrated on a new platform, covering most of the re-engineering effort.

Listing 1.1 and 1.2 are showing Java code portions to store an entity respectively on Azure Table and BigTable. The similarity between both source code is striking, and a move from the one to the other is straightforward with manual intervention. But doing this on large sources is tedious and error prone. Both code are storing the same entity using the same scheme. Defining a strategy of transformation between those two elementary operations is possible, and could help to this re-engineering effort to enable a migration. Although differences still remains between the two datastores by design (mostly on the way to handle fragmentation over multiple servers), this variance could also be dealt within the program transformation process. Developers are the first source of knowledge for the huge amount of technology used across all this platform ecosystem. Thereby, providing a DSL helping to define those sets of transformation could leverage the potential of this approach. The DSL would come as an abstraction of tools such as TXL and other general purpose transformation languages.

Fine grained transformations on specific method calls and class wise modifications could be made using TXL [14], a Language processing transformation on sources. It provides with rewrite rules, strategies to apply those rules with support for context-sensitive transformations.

Migration phase Migration is the last step of the workflow. It moves the transformed sources on the new provider. Prior to the deployment, some tests are going to be applied on sources to avoid the case of applications that are not providing a correct behavior. At this time, we are uncertain on the manner tests are going to be handled. They will need to fit the targeted Cloud provider, then adding complexity to the process for the developer that is going to write those tests. After the application being validated, deployment on the target platform could be launched. Runtime informations are caught after the deployment process to validate the success of the migration. Then, we take back resources on the ancient provider. Still, if something goes wrong we could *rollback* to the ancient state, that was saved during the discovery phase.

4 Related Work

Existing work addresses the problem by using middlewares. mOSAIC is one of those projects. However there are several reasons for us not to use the middleware approach. First the *Overall complexity*: In order to support the widest amount of technologies, middlewares are strongly tied to the software and often provides with configuration files and eventual cluttered logic. Also, it could cause a *Performance overhead* during program execution. Cloud software are especially developed to be accessed by a huge amount of users, generating an enormous quantity of data to write and read. This makes considering a middleware to

support the portability for a potential migration a huge tradeoff. Finally, the *Compatibility list*: The user is tied to the technology supported by the middleware. In the case of a major shift in technology trends, the customer will always face this transformation process if the middleware still dont offer support for the targeted technology. Existing work are already relying on models to adapt legacy software to be migrated on the Cloud through modernization (from the companys infrastructure to a cloud infrastructure or platform) such as REMICS [17], MODAClouds [9] and artist [1] amongst notable cloud migration projects. These solutions are focusing on the provisioning or on the migration of legacy software to the Cloud. None of these are actually dealing with the migration between PaaS. Another existing work is CloudMIG Xpress [15] which is also using KDM to reverse engineer cloud software to check violations for the deployment but does not try to correct those.

5 Conclusions and Future work

In this paper, we presented our approach to deal with software migration between Cloud platforms. We introduced the overall design depicted in three phases: Discovery, Transformation and Migration. We rely on program transformation to enable the migration between PaaS solutions. The discovery on sources is performed with MoDisco, which has a built-in KDM (Knowledge Discovery Metamodel) discovery feature. This discovery provides insights on the software configuration to help choosing the best target platform regarding numerous parameters (price, availability, etc.). Instead of providing yet another middleware much likely to cause a performance hit, we will build a system to provide the ability to define transformation rules on cloud software. Those rules will be defined by developers (and we may imagine, Cloud providers to attract new customers), with the help of a dedicated Domain Specific Language. Transformations are made at an architectural level using models and on source code using tools like TXL, which is a language to define transformations.

Future work includes the realization of a dedicated Eclipse plugin. This plugin will directly points to the changes to be made on a Java project to adapt for the migration on a targeted provider. This work is also going to include insights from the discovery phase to choose wisely platform that fits the best in terms of technology and customer needs. Finally, we will go onward with the program transformation support, validating the approach on real use cases. This part being the heart of the contribution.

6 Acknowledgement

The author's work is funded by the MERgE project (ITEA 2 Call 6 11011).

References

1. artist project. `http://www.artist-project.eu/`.
2. Google app engine. `https://appengine.google.com/`.
3. Heroku. `https://www.heroku.com/`.
4. Ibm smartcloud application services. `http://www.ibm.com/cloud-computing/us/en/paas.html`.
5. Microsoft windows azure. `http://www.windowsazure.com/`.
6. Red hat openshift. `https://www.openshift.com/`.
7. Tosca. `http://cloud-standards.org/wiki/index.php?title=Main_Page`.
8. Vmware cloud foundry. `http://www.cloudfoundry.com/`.
9. Danilo Ardagna, Elisabetta Di Nitto, P Mohagheghi, S Mosser, C Ballagny, F D'Andria, G Casale, P Matthews, C-S Nechifor, D Petcu, et al. Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In *Modeling in Software Engineering (MISE), 2012 ICSE Workshop on*, pages 50–56. IEEE, 2012.
10. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
11. David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *Internet and Web Applications and Services, 2009. ICIW'09. Fourth International Conference on*, pages 328–336. IEEE, 2009.
12. Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: a generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 173–174. ACM, 2010.
13. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.
14. James R Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.
15. Sören Frey and Wilhelm Hasselbring. Model-based migration of legacy software systems to scalable and resource-efficient cloud-based applications: The cloudmig approach. In *CLOUD COMPUTING 2010, The First International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 155–158, 2010.
16. Michael Hogan, Fang Liu, Annie Sokol, and Jin Tong. Nist cloud computing standards roadmap. *NIST Special Publication*, page 35, 2011.
17. Parastoo Mohagheghi and Thor Sæther. Software engineering challenges for migration to the service cloud paradigm: Ongoing work in the remics project. In *Services (SERVICES), 2011 IEEE World Congress on*, pages 507–514. IEEE, 2011.
18. Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman, and Mario Piattini. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532, 2011.
19. Dana Petcu. Portability and interoperability between clouds: challenges and case study. In *Towards a Service-Based Internet*, pages 62–74. Springer, 2011.
20. Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crăciun. Portable cloud applications-from theory to practice. *Future Generation Computer Systems*, 2012.

Modeling Cloud Architectures as Interactive Systems

Antonio Navarro Perez and Bernhard Rumpe

Department of Software Engineering,
RWTH Aachen University,
Aachen, Germany

Abstract. The development and maintenance of cloud software is complicated by complex but crucial technological requirements that are tightly coupled with each other and with the software's actual business functionality. Consequently, the complexity of design, implementation, deployment, and maintenance activities increases. We present an architecture description language that raises the level of technological abstraction by modeling cloud software as interactive systems. We show how its models correspond to an architecture style that particularly meets the requirements of cloud-based cyber-physical systems. The result provides a basis for an architecture-driven model-based methodology for engineering cloud software.

1 Introduction

The development and maintenance of cloud software is complicated by complex but crucial non-functional requirements, for instance: deployment, distribution, scalability, robustness, monitoring, multi-tenancy, and security [1]. These requirements are tightly coupled with each other and with the software's actual business functionality. Consequently, the complexity of design, implementation, deployment, and maintenance activities increases.

Arguably, one of the most complex representatives of cloud-based systems are cyber-physical systems [2] with cloud-based components. In such systems, cloud software interacts with the physical world by monitoring and controlling numerous physical states and processes. Contemporary examples of such systems come from domains such as smart homes [3], smart grids [4], and connected cars [5]. In these systems, cloud software acts as the "brain" of the system by organizing multitudes of different clients, data streams, processes, and calculations. Significantly, all these physical states of affairs are constrained by inherent concurrency, reliability issues, and soft or hard time constraints [6].

In this paper, we introduce an architecture description language (ADL) [7] as the core element of a *model-based* methodology for engineering cloud software. This methodology understands and describes these systems as *interactive systems* [8]. In this context, our ADL describes the *logical software architecture* of such systems. Thereby, it achieves a system representation that abstracts from its mentioned technological requirements. The description of those is, in turn,

outsourced to other distinct modeling languages that integrate with the logical software architecture model (e.g., infrastructure models that describe a concrete cloud-based infrastructure setup). Moreover, our ADL supports an architecture style similar to actor-based systems [9] that refines the notion of interactive systems and is tailored to the domain of cloud software.

We hypothesize that (a) interactive systems adequately describe a relevant class of cloud-based systems, that (b) our ADL and its underlying architecture style adequately describes the essence of interactive cloud software, (c) that the resulting overall system model is an adequate representation of the actual system and can be used as a basis for generative techniques, and (d) that our methodology thereby reduces the complexity of the engineering process of such systems.

We begin with a brief overview of related work in Section 2. Subsequently, we introduce the requirements of cloud-based cyber-physical systems in Section 3. We present our ADL and its supported architecture style in Section 4 and discuss it briefly. We conclude with an outlook to future work.

2 Related Work

Interactive systems are a special class of reactive systems [10] and commonly described as compositions of independent components (e.g. software, target hardware, external devices) that interact with each other to achieve a common goal. The literature on such systems is extensive. The FOCUS method provides a general formal methodology to describe the architecture of interactive systems through the input/output behavior of components that are statically connected via communication channels. [8, 11] Practical model-based methodologies have been developed that model the architecture of such systems, especially in the domain of real-time embedded systems, for instance, ROOM [12], AADL [13], and MARTE [14]. The ADL MontiArc [15] also models the architecture of such systems according to the FOCUS semantics.

In general, development of cloud-software lacks a set of established standards and tools. [16] In the particular context of cyber-physical systems, Lee et al. discuss a similar need for better specification methods and tools, mentioning modeling and actor-based systems among others. [2, 6] The actor-based paradigm for specifying distributed systems [9, 17, 18] nicely mirrors the essence of interactive systems, addressing in particular the challenge of distribution and scale. Its application in the domain of cloud-based services is recently rising in popularity with frameworks as Scala Akka [19] and Microsoft's Orleans [20].

However, the integration of model-based methodology, actor-based architecture styles, and cloud-based software has not yet been tackled.

3 Interactive Cloud Software

We define interactive cloud software as a class of software that drives cloud-based systems. It shares many essential properties and requirements with software in

reactive systems, for instance, software in real-time distributed systems. Those properties requirements are:

- *Reactiveness:* the software's main logic is driven by reactions to events from its environment (e.g., sensor signals) or internal events (e.g. interactions between software units). Moreover, it must satisfy given time constraints (e.g., to control an actuator device in time to react to an environmental event).
- *Statefulness:* the software maintains a conversational state with the environment it interacts with (e.g., the current operational state of a device).
- *Concurrency:* events in the real world occur simultaneously and at any given time. Consequently, the software has to have the capability to deal with these events in time of their occurrence, that means, to deal with them concurrently.
- *Distribution:* the software communicates with highly distributed clients over networks with uncertain properties. Moreover, the software itself is running on distributed cloud-based infrastructure.
- *Scalability*: to assure the requirements of timely and concurrent reactiveness, the software has to able to scale dynamically to meet load demands.
- *Reliability:* the software controls aspects of the physical world and, hence, has to meet high standards of reliability. Moreover, the quality of large, distributed, heterogeneous systems is difficult to assure. This is especially true for stateful systems as erroneous states might multiply over time in absence of resetting or cleaning countermeasures.

4 Architecture Modeling

The cloudADL (cloud architecture description language) is a textual *Components & Connectors* ADL [7] that describes architectures through hierarchically decomposed, interconnected components. It is an extension of MontiArc [15] and developed with the language workbench MontiCore [21]. It supports an architecture style tailored to systems as described in Section 3.

Architecture models in the cloudADL describe the *logical architecture* of the system. The logical architecture describes the essential structure of the system and the abstract, generalizable properties of its elements. However, it abstracts from specific, non-generalizable implementation details. Thus, cloudADL models do not explicitly define or constraint the technological implementation of modeled concepts or the granularity of how modeled elements are mapped to their counterparts in the real system. For instance, individual components (as introduced in the following section) may represent small software modules (e.g., Java classes), applications (e.g. web applications), or an entire integrated systems. Regardless of that, a component's realization (which in the following is referred to as a runtime component instance) always reflects the same basic, abstract properties expressed by the architecture model.

Figure 1 shows the architecture of a simple cloud service that receives incoming data streams from internet-connected physical sensor devices and stores

them to a database. The Figure shows a graphical representations as well as the textual syntax. We will describe this example in the following.

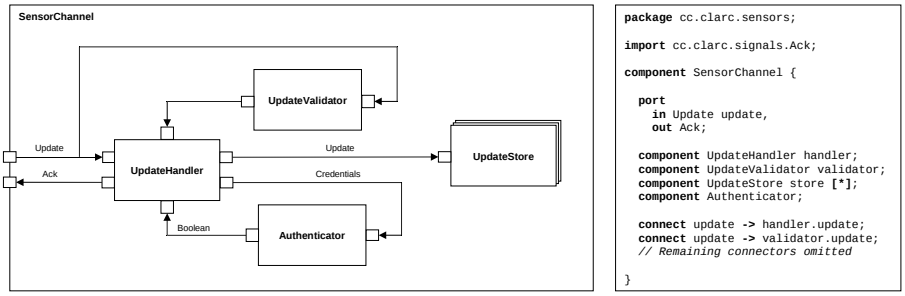


Fig. 1. The architecture of a cloud service that receives and stores data streams from sensors.

4.1 Concepts and Semantics

In this Section we present the structural elements of the cloudADL. We also informally describe their semantics. As an extension of MontiArc, it shares with it the concepts of component types, subcomponents, and connectors, as well as the underlying formalism from FOCUS [11]. Its most important additions to these are replicating subcomponents, and contexts.

Components and Interfaces *Components* are the central building blocks of architectures and represent distinct and independent units of functionality. They interact with each other by *asynchronous [22] message passing* over directed *message channels* that connect an outgoing and an incoming *port* of two components. Ports are typed endpoints for channels and can either receive or send typed messages. Taken together they form the import and export interface of a component.

Channels are associated with *streams*. The notation of streams depicts sequences of messages that have been sent over these channels as seen from a given point in time. Streams, hence, represent the histories of interactions a component has conducted with other components over its incoming and outgoing ports. Their entirety constitutes the visible *behavior* of a component.

The behavior of a component is the specification that defines how the component reacts to incoming messages. It can react by changing an internal state, by sending outgoing messages, by raising an error, or by doing nothing at all. A component's behavior can be defined through additional models (e.g., code of a traditional programming language, or through automaton models) or through *decomposition*. A decomposed component is internally divided into a network of

subcomponents that defines its behavior. Subcomponents are connected via directed *connectors* between two ports, representing the message channels between them.

In Figure 1 `SensorChannel` is a component with an incoming port that receives messages of type `Update` and an outgoing port that sends messages of type `Ack`. It receives updates from sensor devices and responds with positive or negative acknowledgements. This behavior of `SensorChannel` is defined through its decomposition into four subcomponents that are interconnected with each other and with their parent component's ports. `UpdateHandler` takes incoming updates and organizes their processing by the other components. It passes the update's credentials to the `Authenticator` which, in turn, sends back an approval or a rejection. The `UpdateValidator` also receives the update message, checks its payload for validity, and sends the result to the `UpdateValidator`. Finally, the `UpdateStore` takes accepted updates and writes them into a database. Through collaboration, the network of subcomponents implements the functionality of `SensorChannel`.

Components are defined as types, each in its own model. The type definition contains the declaration of the component's ports, its subcomponents, and its connectors. Subcomponent declarations are *prototype instances* of component types which, in this case, are defined in other models. Likewise, port declarations also refer to externally defined types from numerous sources (e.g., types defined by Java classes, or types from UML/P class diagrams [23]).

The use of component types, with the clear distinction between a type's definition and usage, as well as the mechanism of decomposition allows the modeler to compose a concrete architecture by reusing components. This is further supported by the strong encapsulation of a component's behavior. A component type only exports its interface to other component. Its internal structure, be it a network of subcomponents or any other behavior specification, is kept private. Thus, component realizations can be replaced as long as the component's interface does not change.

Replication Subcomponents can be tagged as *replicating* subcomponents. By default, a subcomponent correlates to exactly one runtime instance of that subcomponent's realization in the actual system. Replicating subcomponents, however, may have a variable number of runtime instances. In this way, the architecture model accounts for parts of the software that can dynamically scale at runtime according to dynamic parameters (e.g., the system's current load) by increasing or decreasing the number of those parts that correlate to a replicating subcomponent. The actual replication strategy is left unspecified.

In Figure 1, the `UpdateStore` subcomponent is tagged as replicating. As I/O operations on a database potentially block a system, it can dynamically replicate itself, for instance, based on the frequency of incoming messages with payload to be stored. Note that the component's replicability remains an abstract property and does not imply a particular technical replication mechanism from instantiating new threads to launching new virtual machines.

Message channels attached to replicating components guarantee that every message is received by exactly one replica. Consequently, the sending endpoint of that channel is agnostic about the eventual replicability of the receiver as, in any case, a message is received by one component instance. However, outgoing ports may be tagged as *replicating ports*. Such ports reflect the number of receivers on the other end of the channel and allow for the explicit selection of receiving component instances. Thus, a component's behavior specification is able to implement alternate communication patterns like broadcasting.

Contexts The introduction of replicating components leaves the semantics of message channels underspecified. Channels that connect one or two replicating subcomponents do not give a concrete mechanism for selecting the receiving replica of individual messages. This mechanism can be based on many viable strategies (e.g., round robin), some of which cannot be specified on the level of the logical software architecture (e.g., the selection of the first idle component).

In many scenarios, however, the selection of a particular replica is crucial. For instance, cyclic paths of components and channels might represent a causal chain of interactions in which messages originating at the end of the chain are required to be received by the same component instance that caused the chain in the first place. To give an example, in a web system a set of replicas might be associated to a user session and, thus, require mutually exchanged messages to be only delivered to other replicas of the same set.

Consider the example in Figure 2. A message being received by component A might cause a chain of logically related messages being sent from A to B to C, and from C back to A. Depending on the desired semantic of the message channel between C and A, messages from C might be required to be received by that concrete instance of A that caused the chain of messages in the first place. Without further information, the selection of the concrete replica of A to receive an individual message is ambiguous.

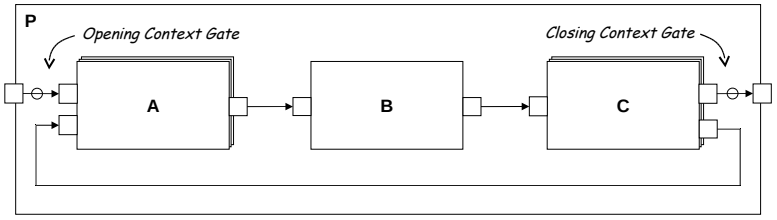


Fig. 2. A set of subcomponents with receiver selection ambiguities.

Contexts are a concept to address these ambiguities. A context is a class of *context tokens* that are valid in the scope of a decomposed component. It is defined by a name and a set of *opening* and *closing context gates*. A message being

sent through an opening context gate is *contextualized* by having a new token of that context assigned to it. Conversely, a message being sent over a closing context gate has its respective token removed. Whenever a contextualized message is received by a component instance, this instance is as well contextualized with the message's context token. Messages and component instances can be contextualized in multiple contexts at the same time.

Contexts can be seen as a generalized mechanism to maintain *conversational states*. A conversational state is a shared state between two or more communicating entities that is defined by the history of communication between them. In our case the entities are components and their communication history is the set of messages they exchanged prior to a certain point in time.

Figure 2 shows two context gates attached to the "first" and "last" connector in P. In this example, all messages received by P are assigned to a token of that context. All messages sent by P have the respective context token removed.

Components that process messages and produce new messages in response are supposed to *handover* context tokens from incoming messages to those outgoing messages that are logically caused by them. In other words, every outgoing message may be the logical result of a set of previously received incoming messages. If this is the case, the outgoing message is contextualized to all contexts of all the incoming messages that logically caused it to be sent.

Usually, the handover can be defined by a simple handover function. For instance, if the behavior specification of a component defines a distinct sequence of actions that are executed for every incoming message, all outgoing messages that are caused by those actions can automatically be contextualized with the context of that incoming message.

To summarize, contexts can be understood as a generalization of the session and session token mechanism in most web systems with client interaction.

4.2 Architecture Style

The concepts of the cloudADL are reflected in the actual system in the form of a particular *architecture style*. In essence, the hierarchy of components in the model corresponds to a hierarchy of system parts—the runtime components—, each of which is associated to a component in the model. Similarly, channels between components correspond to interfaces between those system parts. Moreover, those system parts and interfaces all share common properties that are all results of the concepts of the cloudADL.

Distribution By Default Runtime components are executed in their own thread of control. Thus, they cannot mutually block or crash each other. They influence each other only through asynchronous message passing over explicit message channels with FIFO-buffers at the receiving endpoint. Their internals are encapsulated and cannot be accessed directly.

Moreover, the behavior of runtime components is, with the exception of replicating ports, agnostic about the interaction partners of its component and about

the technical realization of the component's adjacent message channels. A component's behavior implementation receives and sends messages via ports but cannot reflect on what is beyond its ports.

Consequently, the interaction semantics are, by default, of a distributed nature. Functionally, it does not make a difference whether two interacting components are actually distributed or whether they are executed in the same runtime environment. Message channels can be implemented in numerous ways, from local inter-thread communication to dedicated message queue servers. In any case, interactions are always implemented as discrete, asynchronously exchanged typed messages on channels connecting two runtime components. Moreover, the preservation of the order of messages on an individual message channel is guaranteed.

As a result, the architecture style makes physical distribution and networking transparent. The system can be functionally understood without taking its distribution into consideration. Moreover, the physical distribution can be changed without influencing the functional properties of the system.

Supervisor Hierarchy The hierarchy of components in the model translates to a hierarchy of *supervision* in the runtime system. Decomposed components represent management logic, so called *supervisors*. Supervisors monitor and manage the component instances associated to their component's immediate subcomponents. These, in turn, can again be other supervisors or functional components. Supervisors manage, in particular, the replication of runtime components associated to replicating subcomponents, and error handling.

The supervisor's error handling aims to deal with errors as locally as possible. To this end, component instances can implement strategies to deal with internal errors. If they fail, they escalate the error up the hierarchy to their supervisor which, again, implements a strategy to deal with the error. This pattern applies until the error is resolved (e.g., by resetting a component instance) or the root of the hierarchy is reached.

Discussion This architecture style bears resemblance to actor-based systems. [9] However, there are differences. For instance, actors can dynamically instantiate other actors, whereas the general architecture of component instances is, apart from replicating component instances, static. Moreover, actors have one queue for incoming messages, whereas component instances have as many typed queues as they have incoming ports.

Apart from that, our architecture style reflects the properties and requirements mentioned in Section 3 in a similar way. Distribution is addressed through distribution transparency. Statefulness is implemented by the encapsulated state of components. Concurrency is inherent to the system due to each component's independent thread of control. Scalability and reliability are given through supervision. Finally, reactiveness is achieved as a combination of a concurrent, non-blocking, fault-tolerant, and scalable software architecture.

5 Conclusion and Future Work

We have presented an architecture description language that describes the logical software architecture of cloud software in terms of interactive systems. This language is a Components & Connectors language that, most importantly, introduces replication, contexts, and service interfaces as cloud-specific architectural concepts. Furthermore, we have shown how this description maps to the architecture style of distributed, hierarchically supervised components with asynchronous, buffered message passing. We have argued that this architecture style meets the requirements of the class of cloud software that drives cloud-based cyber-physical systems.

We are employing this language in the context of a broader model-based, generative methodology for engineering cloud software named *CloudArc*. This methodology is developed in the context of the SensorCloud project [24] funded by the German Federal Ministry of Economics and Technology, as well as in the context of other projects.

This methodology centers around a logical architecture model and augments it with additional modeling languages specific to other aspects of the overall system. It employs generative techniques in the form of a code generator product line. These code generators synthesize middleware that implements the architecture style described in this paper on various technological infrastructures.

Deployment models are a crucial aspect of the modeling languages in CloudArc. These models describe an infrastructure architecture independent of the logical software architecture and use a mapping model to relate the two. The resulting deployment description is leveraged by the code generators to produce a middleware implementation that reflects the software's target infrastructure. The system's business logic is subsequently implemented through handwritten code.

Test specification models allow for the model-based testing and simulation of the system without the need to deploy it onto production infrastructure. Scenario models and input/output specifications are employed to test the business functionality on a particular variant of middleware that simulates a distributed system but runs locally.

These concepts are currently under development. We plan to evaluate them systematically in the future in the context of our projects.

References

1. Armbrust, M., Stoica, I., Zaharia, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A.: A View of Cloud Computing. Communications of the ACM **53**(4) (April 2010) 50
2. Lee, E.A.: Cyber-Physical Systems - Are Computing Foundations Adequate? October (2006)
3. Harper, R., ed.: The Connected Home: The Future of Domestic Life. Springer London, London (2011)

4. King, C., Strapp, J.: Software Infrastructure and the Smart Grid. In: Smart Grid Integrating Renewable Distributed Efficient Energy. Elsevier Inc. (2012) 259–288
5. Iwai, A., Aoyama, M.: Automotive Cloud Service Systems Based on Service-Oriented Architecture and Its Evaluation. In: 2011 IEEE 4th International Conference on Cloud Computing, IEEE (July 2011) 638–645
6. Lee, E.A.: Cyber Physical Systems : Design Challenges. Electrical Engineering (2008)
7. Medvidovic, N., Taylor, R.N.R.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering **26**(1) (2000) 70–93
8. Broy, M., Stø len, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg (2001)
9. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. Dissertation, Massachusetts Institute of Technology (December 1986)
10. Harel, D., Pnueli, A.: On the development of reactive systems. (February 1989) 477–498
11. Ringert, J.O., Rumpe, B.: A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. Int. J. Software and Informatics **5**(1-2) (2011) 29–53
12. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. Wiley professional computing. Wiley (1994)
13. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI series in software engineering. Addison-Wesley (2012)
14. OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems - Version 1.1. Technical report, Object Management Group (2001)
15. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical report, RWTH Aachen University, Aachen (2012)
16. Hogan, M., Liu, F., Sokol, A., Tong, J.: NIST Cloud Computing Standards Roadmap. Technical report, National Institute of Standards and Technology (2011)
17. Armstrong, J., Virding, R., Williams, M.: Concurrent programming in ERLANG. Prentice Hall (1993)
18. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java - PPPJ '09, New York, New York, USA, ACM Press (August 2009) 11
19. Haller, P., Odersky, M.: Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2-3) (February 2009) 202–220
20. Bykov, S., Geller, A., Kliot, G., Larus, J., Pandya, R., Thelin, J.: Orleans: A Framework for Cloud Computing. (2010)
21. Krahn, H., Rumpe, B., Völkel, S.: MontiCore: A Framework for Compositional Development of Domain Specific Languages. International Journal on Software Tools for Technology Transfer **12**(5) (March 2010) 353–372
22. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. ACM SIGPLAN Notices **23**(7) (July 1988) 260–267
23. Schindler, M.: Eine Werkzeuginfrastruktur zur Agilen Entwicklung mit der UML/P. Dissertation, RWTH Aachen University (2011)
24. : SensorCloud. http://www.sensorcloud.de/

VehicleFORGE: A Cloud-Based Infrastructure for Collaborative Model-Based Design

Laszlo Juracz, Zsolt Lattmann, Tihamer Levendovszky, Graham Hemingway, Will Gaggioli, Tanner Netterville, Gabor Pap, Kevin Smyth, Larry Howard

Institute for Software Integrated Systems
Vanderbilt University
Nashville, TN

Abstract. Recent increases in industrial adoption of Model-Based Engineering has created demand for more advanced tools, environments, and infrastructures. As a response to the Defense Advanced Research Project Agency's (DARPA) initiative in the Adaptive Vehicle Make (AVM) program, we have designed and built VechicleFORGE, a collaborative environment tightly integrated with the AVM design tools. This paper describes VehicleFORGE concepts and services facilitated by the cloud computing foundation of the infrastructure.

1 Introduction

The VehicleFORGE [5] platform is designed and maintained to host the DARPA Fast, Adaptable, Next-Generation Ground Vehicle (FANG) series of prize-based design competitions as part of the AVM [1] portfolio. In the first FANG challenge, the AVM program set out to apply crowdsourcing practices to involve a larger group of individuals in the design work. After registration, competitors can form teams and gain access to the modeling and analysis tools and modeling components being developed by the program. Although subsequent competitions may be less open in terms of public participation, operating the competition requires a centralized platform where participants can collaborate and which serves as a location for distributing tools, design requirements and components, documentation, sharing results and accessing compute resources.

VehicleFORGE provides the virtual environment and cloud infrastructure which enables the management of the competitions, competitors, and the collaboration of geographically distributed design teams, as well as various cloud-based analysis services and tools for the design work.

We develop and operate VehicleFORGE and the underlying cloud infrastructure using open source technologies. Both the web application and the monitoring tools are designed to enable streamlined deployability and scalability to meet changing utilization profiles and to satisfy security requirements set by DARPA.

2 The Forge Framework

In the past decade, the success of crowdsourcing and the model of distributed problem-solving and software production has been enabled by widely popular open source software forges such as SourceForge.net [4]. After investigating the available technologies, we decided to use the Allura application developed by SourceForge.net as a basis for the development of VehicleFORGE.

2.1 Concepts of the Platform

Although the architecture of the application has greatly evolved, the organization of the fundamental forge concepts in VehicleFORGE is derived from the core Allura application.

Projects embody the collaborative spaces where members of a team of users manage design work, create and share design artifacts and analysis results. Registered users can form new Projects or acquire membership in an existing one. Projects are created based on pre-configured templates but in general, each team controls how it utilizes the Project for its work.

Neighborhoods are collections of projects, usually representing a higher-level real-word organizational concept (eg. competition) with which the teams of the member projects are affiliated. Neighborhoods also offer similar collaboration functionalities to the project spaces: they can have members, customized roles and selected tools installed for neighborhood-level collaborative work.

Tools are provisioned in the project space and house the data and interfaces for engaging in collaborative design work. Privileged project administrators can freely install and administer new tools. Objects created during the collaborative and design work in a tool are referred to as *artifacts*.

Among the various out-of-the-box tools, VehicleFORGE offers *Subversion* [15] (SVN) and *Git* [25] repositories for sharing files created in desktop-based design tools. Through a set of *post-commit hooks*, the forge processes newly added content to update its understanding of a project's designs. Project members can access web-based previews of each other's work, and files and design artifacts recognized this way can be cross-referenced with artifacts created in other VehicleFORGE tools. Thus, repositories work as the bridge between design work done on the desktop and the web-based collaboration environment.

A major extension that VehicleFORGE offers to the basic forge concepts found in software forges is the *Component Exchange*. It offers a globally readable shared space in which users can publish, categorize, share and discover formally characterized reusable design components outside of the scope of an individual project.

VehicleFORGE implements customizable *role-based access control*: each project can create *permissions* and *user groups* to match its requirements. The combination of groups and permissions are used to determine the access to artifacts contained in a tool.

Every project (and neighborhood) space has an *Admin Tool* where team-leaders can provision new project tools and configure the permissions. Each

VehicleFORGE deployment has a dedicated ForgeAdmin project which contains services related to the entire forge. There are various tool-specific aggregation and statistical interfaces available to monitor collaboration and design activities in the team and neighborhood scope.

2.2 Customizability

The basic forge concept developed by software forges was designed primarily for supporting work with source-code, however, VehicleFORGE is easily customizable for collaboration in arbitrary domains, from cyber-physical system design [3] to policy authoring [29]. Beyond the flexible project configuration and access-control administered through web interfaces, VehicleFORGE supports multiple levels of extensibility.

Visualizers provide a means for third party developers to implement new visualization of domain-specific repository content. Visualizers are executed in the user's browser and they are not part of the main application code base, however, they can be deployed along with custom post-commit hooks to do server-side preprocessing of the files containing the information to be displayed.

The forge application and the project tools are written in the Python-based TurboGears framework. Experienced developers can make significant capability extensions by developing new forge tools or modifying parts of the open source forge framework.

3 Services in the cloud

The VehicleFORGE cloud infrastructure facilitates the creation and operation of multiple forge deployments and provides the flexibility to scale deployments to changing loads. Its extensible pool of resources is available for virtualizing various operating environments to extend VehicleFORGE platform capabilities and to offer compute and analysis as a service through the forge to the AVM community.

3.1 Scalability of Forge Deployments

The deployment architecture is designed so that every significant, load-bearing service on the Forge is horizontally scalable. Various strategies are employed on a service-by-service basis to enable this. The web server is run on multiple processes across multiples instances. All requests are initially routed to a server running a fast, lightweight load balancer service that distributes the requests intelligently to the available web servers. A similar strategy is used to scale the compute and repository services. To scale the database, index, and cache services we use replication support built in to the specific software implementations.

The service infrastructure is designed to minimize response time and optimize cloud resource utilization. Figure 1 depicts the service architecture for a VehicleFORGE deployment. Most requests begin and end through a web server

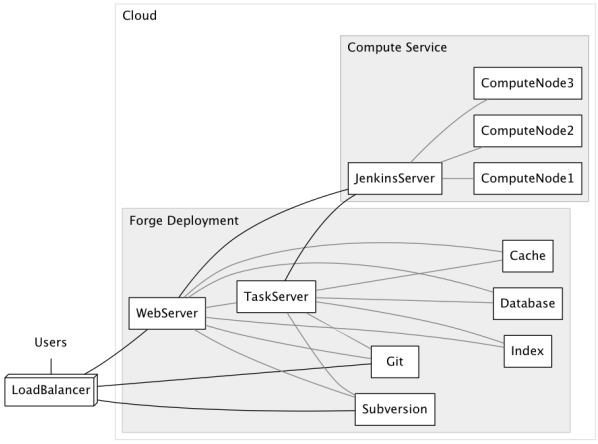


Fig. 1. VF Service Artchitecture.

gated by the load balancer. To minimize latency, the web servers delegate longer-running, more resource-intensive jobs to *Task Servers* that execute the jobs asynchronously. In a similar vein, web servers communicate with the META *Compute Service* to conduct the Testbench analyses. The Compute Service nodes are separated from the main VehicleFORGE deployment to ensure that they have access to the necessary resources for their intensive analysis tasks.

3.2 Test Bench Execution

Test benches, design spaces, components Within the AVM program, a design tool chain (META tools) [23,38] is being developed for exploring design alternatives and analyzing designs under certain conditions. The META tools provide the integration framework for components, designs, design spaces, requirements and test benches. *Components* are atomic building blocks that describe the dynamics behavior and the structural aspect of physical objects. *Designs* and *design spaces* are built up from components, where a design has a fixed architecture (composition of components) and a design space can encode different component alternatives as well as different design architectures.

After a design or design space is created, test cases are defined against the given requirement set. The test cases, which we term *test benches*, are executable versions of the system requirements. From the test bench models, the META tools can compose analysis packages over a design space for different domains such as simulation of DAEs (differential algebraic equations), formal verification, static analysis, and structural analysis. Examples include vehicle model simulation using different drive cycles such as highway speed profile or urban speed profile, cost of the design by aggregating the cost of the components, and structural stress analysis on the composed CAD (3D) model of the design.

Resource considerations Executing the test benches may require different platforms; the execution time varies based on model size and analysis type. Furthermore, the number of test bench executions depends on the number of test benches and the design space size (i.e. number of designs). During this project we used approximately 30 test benches and 400 design variations, which evaluates to about 12k test bench executions. Additionally, we had to take into account the size of the generated analysis results and provide for their storage.

Implementation and cloud usage Initially, all test benches were executed sequentially within the same process, while we had only just 1-2 test benches and 1-5 designs. As we increased the number of test benches and the complexity of the designs, we switched to another solution. We implemented a client side job manager (called META Job Manager) for running the test benches on multiple CPUs using a single machine and limited the number of maximum parallel jobs to the number of CPUs. As the execution time for a single test bench started increasing because the complexity of the design increased, the 1-3 hour simulation times were an unacceptable encumbrance on the user's machine. For this reason, we extended the META Job Manager with a remote execution service that authenticated with VehicleFORGE.

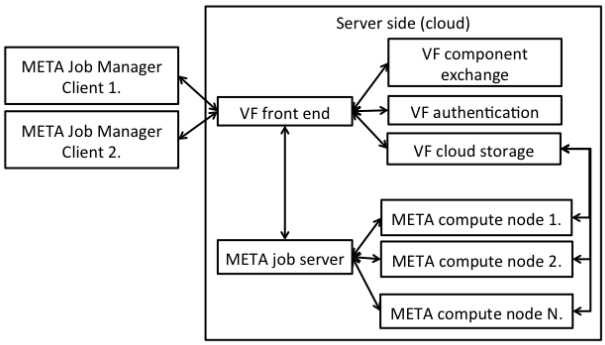


Fig. 2. META VF infrastucture.

Figure 2 depicts the communication path between the client side job manager and the server side services. The META Job Manager can be configured to execute the test benches locally (in parallel) or post them to VF to be executed on compute nodes. If the job manager is configured for remote execution, the user provides the login information and logs in through the VF authentication service. Then, the job manager is ready to accept jobs and post them to VF. When a job is posted to the job manager, it calls a VF service to register the job, uploads the analysis package to a cloud storage server, then indicates the job is

ready to execute. The job gets posted to the META server side job dispatcher and is added to the queue. If there is a free executor on a compute node that has all the tools required to execute the job, the job gets shipped to the compute node and the execution begins. A compute node for a job is chosen based on the version of the desktop tools that the user employed to create his/her design. For example, we had different version of the META tools deployed on the client machines and some of our compute nodes supported one version of the tool chain, while others supported another version of the tools. The compute node downloads the analysis package from the cloud storage, executes the analysis, uploads the results to the cloud storage, and indicates that the job is complete (successfully or with failures). The META Job Manager monitors the pending jobs regularly. When the analysis is done, it downloads the results to the client's machine.

Optimization and cloud benefits Since the component models and their artifacts are stored on VF, which are accessible through the VF component exchange, the data transfer between the server and clients was reduced by sending only links to components rather than the actual content. This significantly improved the turnaround time for the CAD jobs.

Users do not need to install the analysis tools, if they only use remote execution. The remote compute nodes have all the tools set up and configured correctly to perform the test bench executions. As a result of having compute nodes, the load on the users machine was decreased and it requires only communication with the server to send and receive test bench bundles.

Within 3 months the remote compute nodes served 1000 users (200 teams) and ran 50000+ analysis jobs with 92 percent success[1].

3.3 Monitoring the infrastructure

Operating a complex distributed application demands constant monitoring and maintenance. The dynamic nature and virtualized deployment of the VehicleFORGE application further complicate the problem of providing a cohesive understanding of its status. We have deployed a number of monitoring and management tools into both the VehicleFORGE application and its underlying cloud infrastructure in order to automate and simplify the tasks of monitoring and managing operations.

The foremost requirement for application monitoring is simply to understand the current state of the application. This requirement spans from low-level checks, such as disk and memory usage on the virtual machines, to higher-level needs, such as checking database flush times. The VehicleFORGE team selected Nagios3 [33] for status monitoring, though excellent alternatives, such as Munin [32] and Ganglia [28], exist. We chose Nagios because it provides a very large library of built-in checks, is easily extensible for custom checks, and is resource

[1] The job execution largely did *not* fail on server side, but the analysis results may or may not have passed the requirements.

efficient enough to execute all of our checks in very short intervals (less than a minute in our case). A key aspect of status monitoring is the ability to reflect the process topology of the deployment, i.e. which process is running on which machine. As our entire deployment process is based on modeling and automation, it was a natural extension for us to develop a mapping from the deployment blueprint to the Nagios monitoring configuration. As alterations are made to a given deployment a new monitoring configuration can be easily synthesized. In the production deployment of VehicleFORGE, Nagios performs 374 independent status checks every minute. If any of these checks fail, an administrator is notified immediately and can begin to remediate the issue.

While status monitoring helps to ensure the uptime of the application, very frequently administrators and support personnel need to understand the historical context of a specific application operation or event. For example, why was a particular user's registration request rejected, or how many users logged in on a particular day. For these types of operational questions it is best-practice to instrument the application so that it logs all relevant events with some pertinent information. Typically these logs reside on the local filesystem of each machine. Standard Linux processes make extensive use of logging, too. It can be a non-trivial problem in distributed applications to collect all of the desired logging information, centralize it, process and analyze it, and archive it. Similar to status monitoring, several open source alternatives exist for log handling, notably LogStash [37] and Scribe [19]. After evaluating the alternatives, we chose to use LogStash in conjunction with ElasticSearch [22] for log indexing, and Kibana [20] for analysis and visualization. This combination is very easy to deploy and configure and require minimal resources during operation. Every event occurring in both the VehicleFORGE application and the underlying virtual machine it logged, collected, indexed and archived. This provides our operations team with tremendously powerful tools to understand both what is happening at any given moment, and past historical trends. The volume of data generated by our logging approach is non-trivial though. In an average hour of operation, the production VehicleFORGE deployment generates over 4.3 million log records which consume almost 5MB of disk space. In one year of operation, that equates to nearly 38 billion records and 40GB of data. A dedicated cluster of virtual machines is needed to index the data and execute searches across these records.

Finally, both developers and operators need to understand what is happening "inside" of the application. This understanding is at a deeper level than is typically provided by tools such as Nagios, Munin or Ganglia. The need is also more "real-time" than is provided by analysis of historical log information. A typical use of such real-time statistics is a operations dashboard. On this dashboard may be a number of statistics that allow an operator to assess the internal state of the deployment in a glance, for example, a real-time plot of the number of active users. An event such as a DNS failure that sends a large share of traffic away from the site would not be detected by either the monitoring system or log analysis. Another example would be a software update roll-out that causes users to start receiving errors. In both of these cases it necessary to have a deeper and

more immediate view into the state of the application. StatsD [27,26] is just such a tool. The code of an application must be modified in order to support StatsD collection, but once done, as specific events occur within the application a UDP packet is sent to a daemon that receives and aggregates it over time. Within 10 seconds of an event occurring, operators see it appear on a plot that provides significant insight into the internal state of the application. The VehicleFORGE infrastructure makes use of StatsD and in the near future its support will be built into the VehicleFORGE application too.

3.4 Operator tools

The maintenance of a distributed application can be greatly simplified through the development of autonomic tools for configuring and managing a cloud deployment. The complex inter-dependencies between subcomponents of the VehicleFORGE application and the services required to provision the application, as well as the complexities inherent to managing a distributed application of this scale, guided our development of *Da Capo*, a cloud provisioning and orchestration framework that we develop and maintain internally. Da Capo provides an extensible framework into which we have injected a VehicleFORGE-specific component that delineates the particulars of orchestrating our application. We use Da Capo extensively in the development of VehicleFORGE to create and manage both publicly accessible production deployments and short-lived, private sandbox deployments for development and testing.

Da Capo greatly facilitates the creation and configuration of distributed applications. Using its API through a web interface contained in the lightweight VehicleFORGE tool *ForgeCloud*, we are able to configure new VehicleFORGE deployments. Configuration is performed through the definition of app-specific parameters and *services*, persistent/long-running background processes (e.g. the database) upon which the application depends. Through the ForgeCloud configuration interface, we can specify the number of instances in the deployment, the size of those instances, the service(s) contained on each instance, and various configuration parameters that define how the application will function. Da Capo is aware of which services are required, limits to the number of a particular service type that can exist in a functional deployment, and any inter-dependencies between services. In short, it will ensure that a deployment specification will result in a valid deployment before it is initialized.

Once the deployment specification is submitted, Da Capo provisions the necessary resources by communicating with the OpenStack API [31]. When the instances are ready, it installs and configures the application, services, and their dependencies on the designated instances. It handles any requisite deployment-specific and user-specified configuration. Finally, Da Capo initializes the services and the application. Any errors that occur during this process are logged and reported.

Da Capo additionally offers tools for monitoring and managing a running deployment. It can create and restore backups for a deployment by running the appropriate commands on the appropriate services (in our case Solr [35],

MongoDB [13], Swift [6], Git, SVN and Redis [21]). It can execute an operator-inputted command on all instances of a deployment, or only instances running specific services. It can stop, start, and restart specified services, display logs, check the status of all services, detect and run necessary application migration scripts, and perform an operating system upgrade. Further, Da Capo's command infrastructure is easily extensible, ensuring that any future automation needs will be met.

4 Related Work

VehicleFORGE adopts cutting edge FORGE components that have proven themselves in textual language-based projects. Wiki pages [24] offer a convenient way of sharing hypertext-based information, which are aided by other established tools, many based on those of the Allura Project [2], to provide a collaboration hub for distributed design teams.

Although there are several model-based collaboration environments in the field of CAD and general domain-specific languages,[8,34,17,16,18,7]– literature overview with analysis can be found in [11] and [9]–, VehicleFORGE offers a number of novel concepts, including the Component Exchange, among others. The tight integration with the META toolchain and the efficient use of state of the art cloud computing and collaboration technologies also make it a unique infrastructure. The vision of combining model-driven engineering and cloud computing has been proposed in existing publications [10,12,14], but as of yet no publication details the creation of a functioning deployment.

A distributed collaborative design evaluation platform called DiCoDEv [30] uses virtual reality techniques to implement real-time manufacturing design. The focus of VehicleFORGE is broader than manufacturing and it provides several offline services. In [36], the authors describe a collaboration environment with source control management, forum, mailing list web sites, news, and project categorization. As opposed to VehicleFORGE, it is restricted to textual content written in the language R.

5 Conclusions and plans

In this paper, we have introduced VehicleFORGE, a cloud-based application for collaborative model-based development. VehicleFORGE utilizes cutting edge cloud computing technologies in order to maintain scalability for resource-intensive design work. Various domain-specific tools that benefit from high computational power and centralized resources, such as design space explorers, can use the VehicleFORGE cloud to great advantage. VehicleFORGE has been used in United States-wide competitions, which was made possible by the monitoring and operator tools. Development and maintenance on VehicleFORGE deployments is enabled by custom platform as a service software developed in house.

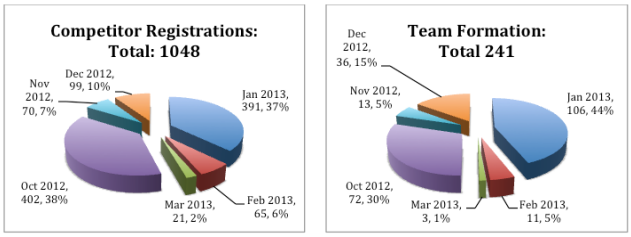


Fig. 3. User and Team registrations throughout FANG-I

There were 1048 competitors and 241 design teams registered on the main VehicleFORGE website which served as the home for the FANG-I Design Challenge (see Figure 3). Besides this deployment, the *VehicleFORGE Production Cloud* hosted the testbench analysis services, a Beta website for staging and testing FANG Challenge resources and an internally used platform for managing the development of the system itself. There will be further forge instances deployed for the AVM program in the upcoming period to the FANG-II Design Challenge.

The *Development Cloud* hosts the Alpha VehicleFORGE website which is maintained for educational purposes and several, on-demand sandbox-deployments created for development and testing purposes.

In the meantime, we are working on the first release of the refactored, new forge framework, which is designed for utilization by a greater open source community–outside of the immediate scope of the AVM program.

6 Acknowledgments

This work was sponsored by DARPA, under its Adaptive Vehicle Make Program. The views and conclusions presented are those of the authors and should not be interpreted as representing official policies or endorsements of DARPA or the US government.

References

1. *Adaptive Vehicle Make*. `http://www.darpa.mil/Our_Work/TTO/Programs/Adaptive_Vehicle_Make__(AVM).aspx`.
2. *Allura*. `http://sourceforge.net/projects/allura/`.
3. *Cyber-physical system*. `http://en.wikipedia.org/wiki/Cyber-physical_system`.
4. *SourceForge*. `http://sourceforge.net`.
5. *VehicleFORGE*. `http://www.vehicleforge.org`.
6. J. Arnold. *Software Defined Storage with OpenStack Swift*. SwiftStack, Inc., April 2013.

7. O. Berger, C. Bac, and B. Hame. Integration of libre software applications to create a collaborative work platform for researchers at get. *International Journal of Information Technology and Web Engineering (IJITWE)*, 1(3):1–16, 2006.
8. R. Bidarra, E. Van Den Berg, and W. F. Bronsvoort. Collaborative modeling with features. In *Proceedings of DET*, volume 1, page 2001, 2001.
9. G. Booch and A. W. Brown. Collaborative development environments. volume 59 of *Advances in Computers*, pages 1 – 27. Elsevier, 2003.
10. H. Bruneliere, J. Cabot, F. Jouault, et al. Combining model-driven engineering and cloud computing. In *Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud'10: Workshop's 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*, 2010.
11. J. Cabot, G. Wilson, et al. Tools for teams: A survey of web-based software project portals. *Dr. Dobbs*, pages 1–14, 2009.
12. J. Castrejón, G. Vargas-Solar, C. Collet, and R. Lozano. Model-driven cloud data storage. *Proceedings of CloudMe*, 2012.
13. K. Chodorow. *MongoDB: the definitive guide*. O'Reilly, 2013.
14. C. Clasen, M. D. Del Fabro, M. Tisi, et al. Transforming very large models in the cloud: a research roadmap. In *First International Workshop on Model-Driven Engineering on and for the Cloud*, 2012.
15. B. Collins-Sussman, B. Fitzpatrick, and M. Pilato. *Version control with subversion*. O'Reilly, 2004.
16. R. Frost. Jazz and the eclipse way of collaboration. *Software, IEEE*, 24(6):114–117, 2007.
17. J. Gallardo, C. Bravo, and M. A. Redondo. A model-driven development method for collaborative modeling tools. *Journal of Network and Computer Applications*, 35(3):1086–1105, 2012.
18. C. Herrmann, T. Kurpick, and B. Rumpe. Sselab: A plug-in-based framework for web-based project portals. In *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, pages 61–66, 2012.
19. R. Johnson. Facebook's scribe technology. October 2008.
20. R. Khan. *Kibana*. `http://kibana.org/`.
21. J. A. Kreibich. *Redis: The Definitive Guide: Data Modeling, caching, and messaging*. O'Reilly, 2013.
22. R. Kuc and M. Rogozinski. *Elasticsearch Server*. Packt Publishing, 2013.
23. Z. Lattmann, A. Nagel, J. Scott, K. Smyth, J. Ceisel, C. vanBuskirk, J. Porter, T. Bapty, S. Neema, D. Mavris, and J. Sztipanovits. Towards automated evaluation of vehicle dynamics in System-Level designs. In *CIE*, 2012.
24. B. Leuf and W. Cunningham. The wiki way: quick collaboration on the web. 2001.
25. J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.
26. I. Malpass. Measure anything, measure everything. http://codeascraft.com/2011/02/15/measure-anything-measure-everything/, February 2011.
27. I. Malpass. Statsd. StatsD Repository, July 2013.
28. M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
29. A. Nadas, L. Juracz, J. Sztipanovits, M. E. Frisse, and A. J. Olsen. Policyforge: A collaborative environment for formalizing privacy policies in health care.

30. M. Pappas, V. Karabatsou, D. Mavrikios, and G. Chryssolouris. Development of a web-based collaboration platform for manufacturing product and process design evaluation using virtual reality techniques. *International Journal of Computer Integrated Manufacturing*, 19(8):805–814, 2006.
31. K. Pepple. *Deploying OpenStack*. O'Reilly, 2011.
32. G. Pohl and M. Renner. *Munin - Graphisches Netzwerk- und System-Monitoring*. Open Source Press, April 2008.
33. M. Schubert, D. Bennett, J. Gines, A. Hay, and J. Strand. *Nagios 3 enterprise network monitoring: including plug-ins and hardware devices*. Syngress, 2008.
34. N. Shyamsundar and R. Gadh. Internet-based collaborative product design with assembly features and virtual design spaces. *Computer-aided design*, 33(9):637–651, 2001.
35. D. Smiley. *Solr 1.4 Enterprise Search Server*. Packt Publishing, 2009.
36. S. Theußl and A. Zeileis. Collaborative software development using r-forge. 2008.
37. J. Turnbull. *The LogStash Book*. Amazon, 2013.
38. R. Wrenn, A. Nagel, R. Owens, H. Neema, F. Shi, K. Smyth, D. Yao, J. Ceisel, J. Porter, C. vanBuskirk, S. Neema, T. Bapty, D. Mavris, and J. Sztipanovits. Towards automated exploration and assembly of vehicle design models. In *CIE*, 2012.

A Model-driven Approach for Price/Performance Tradeoffs in Cloud-based MapReduce Application Deployment

Shashank Shekhar, Faruk Caglar, Kyoungho An, Takayuki Kuroda, Aniruddha Gokhale[1], and Swapna Gokhale[2]

[1] Institute for Software Integrated Systems, Dept of EECS
Vanderbilt University, Nashville, TN 37235, USA
Email: { sshekhar,caglarf,kyoungho,kuroda,gokhale}@isis.vanderbilt.edu
[2] Dept of CSE, Univ of Connecticut
Storrs, CT, USA
Email: ssg@engr.uconn.edu

Abstract. This paper describes preliminary work in developing a model-driven approach to conducting price/performance tradeoffs for Cloud-based MapReduce application deployment. The need for this work stems from the significant variability in both the MapReduce application characteristics and price/performance characteristics of the underlying cloud platform. Our approach involves a model-based machine learning capability that trains itself from executing a variety of MapReduce applications on different cloud service providers, and in turn providing useful price/performance tradeoff information to MapReduce application users. Additionally, the model-based platform serves to automate the deployment of a MapReduce application to the cloud by incorporating the tradeoff choices.

1 Introduction

Emerging trends and technical needs: With the ever growing size of data to be processed, the need for advances in processing very large data sets keeps growing. MapReduce [1] is a widely-used framework for Big Data, which has a parallel programming model and executes on large clusters. Since the number of *map* and *reduce* tasks used for a MapReduce application depends on a variety of factors, such as the size of the data set, the degree of reliability, presence of combiners and number of partitions, executing a MapReduce application requires elastic computing, storage and networking resources. These requirements makes cloud computing a perfect fit to execute MapReduce applications due to its support for elastic computing.

There are two significant challenges faced by MapReduce users. First, Cloud-based provisioning incurs a cost, which is hard to precisely estimate *a priori*. The reason is that the cost varies depending on both the MapReduce job characteristics stemming from the factors outlined above, and also on which cloud service

provider (CSP) is used to execute the job. The latter is due to each CSP having their own pricing model and Service Level Agreements (SLAs). Second, the performance of applications running in the cloud also varies significantly from one CSP to another [6]. This problem is not limited to MapReduce applications alone but can apply to many other types of applications that illustrate variability in the resources they consume. For this paper, however, we scope out the research investigations in the context of MapReduce applications.

Research contributions: In summary, MapReduce application users face significant challenges in making the right price/performance tradeoffs when deciding to use the Cloud to execute their applications, not to mention the learning curve involved in deploying their applications on a particular CSP's platform. There is an urgent need for a solution that can help a MapReduce user choose from various CSPs by making appropriate price/performance tradeoffs, and subsequently automate the deployment of their application on the chosen cloud provider. To address this need we present preliminary ideas on a model-driven engineering (MDE) approach to making the price/performance tradeoffs and using these decisions to automate the deployment of the application to the cloud. We provide this capability as a web-based service to the end user.

The web service utilizes the MDE approach in the following manner. First, the web service engineers use their MDE tool – developed using domain-specific modeling – to train a machine-learning tool by executing a variety of MapReduce applications with different price and performance characteristics on a variety of CSPs. The deployment of the applications used in the training phase is automated via yet another MDE-based generative tool that can synthesize deployment scripts for the underlying CSP.

The web service offers the end user (i.e., the MapReduce application user) with an interface – essentially an abstract model of a MapReduce application and its properties developed using a MDE approach through the use of domain-specific modeling – so that the user is shielded from details of the CSPs; instead they focus on providing the key requirements of their MapReduce application to the web service. The machine learning tool owned by the web service can then classify an incoming MapReduce application supplied by the end user into a known class of MapReduce application categories. We leverage our prior experience on job classification [10] for this work. Note that the end user is not cognizant of the machine learning tool. The web service subsequently consults its trained engine and provides the end user with different price/performance tradeoffs. The end user in turn uses the web interface to make a deployment choice at which point the web service reuses its automated cloud deployment capabilities to deploy and execute the end user's MapReduce job on the CSP.

2 Related Work

There has been some existing work to solve the problems we are handling in this research. Ganapathi et al. [3] have used Kernel Canonical Correlation Analysis (KCCA) to predict the performance of MapReduce application in the cloud

environment. Kadirvel et al. [4] have evaluated and compared wide range regression techniques for MapReduce applications. However, these research have not provided any guidance or tool for estimating the cost of deployment and execution. Liew et al. [8] have developed a tool suite called CloudGuide for estimating cloud deployment cost and performance for legacy web applications. Li et al. [7] have also proposed a framework, CloudProphet for achieving the same objective. However, they have not provided anything specific to MapReduce based applications.

SPACE4CLOUD [2] follows a model-driven approach for estimating cost and performance for different cloud systems and defines three meta-models: Cloud Independent Model (CIM), Cloud Provider Independent Model (CPIM) and Cloud Provider Specific Model (CPSM) for different levels of abstractions. The solution tries to address any type of cloud application, but the results are not encouraging for heavy workloads which is the primary criteria for any MapReduce solution. Thus, we need a solution which could help the customers and to cater to the variabilities and commonalities amongst the CSPs for which a model driven approach seems an appropriate choice.

3 Project Status and Ongoing Work

In present form we have developed an MDE-based cloud deployment framework that automates the deployment and execution of MapReduce applications used for training our machine learning-based engine as well as deploying a user-supplied application. We focus on the Hadoop framework, which is the most widely used MapReduce framework. For developing the modeling capabilities, we have used the Generic Modeling Environment [5] to model and interpret the deployment configurations and executing the MapReduce jobs.

For training our machine learning engine that can illustrate the price/ performance tradeoffs, we have used a Java-based open source machine learning software called Weka and the algorithm used is multilayer perceptron [9]. The training is thus far conducted on cloud platforms that include an OpenNebula-based private cloud infrastructure, and public clouds such as Amazon EC2 and Windows Azure.

Our ongoing work involves the process of finding the right set of factors that affect the performance of a MapReduce application. Kadirvel et al. [4] have identified and divided the factors into six categories i.e. resource, data, program, configuration, faults and environment. We are expanding the list of these factors by including the variabilities within and across CSPs. Some of these include performance variabilities due to different workload at different time of the day and at different zones of the CSPs' datacenters. Some of the factors which we are still investigating to how to take into account are network topology of the datacenter, variability in processor speed/ architecture and data locality. We are also performing sensitivity analysis to figure out the factors which affect the performance the most and leave out the factors which do not, which will reduce

the complexity. This will also help to refine the models and make the MDE tool more effective.

Our future work will involve using the predicted resources to estimate the cost of deploying and executing MapReduce jobs, thus helping the user to select the best CSP for the job. The framework can then deploy and execute the MapReduce job on the CSP without the user needing to deal with the intricacies of the CSP's infrastructure. We also need to develop the modeling abstractions for different CSPs and expanding the models for various factors that impact the performance.

4 Conclusion

In this work, we proposed a framework for conducting price/performance trade-offs in executing MapReduce jobs at various CSPs, selecting the best option and deploying and executing the job on the selected CSP infrastructure. All of these capabilities are driven by a model-driven engineering (MDE) framework that shields the users from the variabilities in the cloud service providers (CSPs) and also automates the deployment of the application on the CSP platform. We propose to offer such a solution as a web-hosted service, possibly as a software-as-a-service, by a providers who is willing to incur the initial cost of training the tool for various configurations on different CSPs. They can then provide services to customers who will pay a fraction of the cost of the saving they get by using the solution. We have conducted preliminary work including training a system on a private cloud platform as well as public clouds. The MDE abstractions are being developed and the realization as a web-hosted service is still under development.

5 Acknowledgment

This work was supported in part by NSF CNS SHF 0915976 and CNS CAREER 0845789. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
2. Davide Franceschelli, Danilo Ardagna, Michele Ciavotta, and Elisabetta Di Nitto. Space4cloud: a tool for system performance and costevaluation of cloud systems. In *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, pages 27–34. ACM, 2013.
3. Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. Statistics-driven workload modeling for the cloud. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 87–92. IEEE, 2010.

4. Selvi Kadirvel and José AB Fortes. Grey-box approach for performance prediction in map-reduce based platforms. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–9. IEEE, 2012.
5. Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing Domain-Specific Design Environments. *Computer*, 34(11):44–51, 2001.
6. Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.
7. Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. Cloudprophet: towards application performance prediction in cloud. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 426–427. ACM, 2011.
8. Siew Huei Liew and Ya-Yunn Su. Cloudguide: Helping users estimate cloud deployment cost and performance for legacy web applications. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 90–98. IEEE, 2012.
9. Sankar K Pal and Sushmita Mitra. Multilayer perceptron, fuzzy sets, and classification. *Neural Networks, IEEE Transactions on*, 3(5):683–697, 1992.
10. Dili Wu and Aniruddha Gokhale. A self-tuning system based on application profiling and performance analysis for optimizing hadoop mapreduce cluster configuration. In *Proceedings of the 20th IEEE International Conference on High Performance Computing, to appear*, 18-21 Dec, 2013, Bangalore, India.

Towards Domain-Specific Testing Languages for Software-as-a-Service

Dionny Santiago, Adam Cando, Cody Mack, Gabriel Nunez,
Troy Thomas, and Tariq M. King

Ultimate Software Group, Inc.
2000 Ultimate Way, Weston, Florida 33326, USA
{dionny_santiago,adam_cando,cody_mack,gabriel_nunez,
troy_thomas,tariq_king}@ultimatesoftware.com
http://www.ultimatesoftware.com

Abstract. There continues to be a trend towards using the power of cloud computing to tackle inherently large and complicated problem domains. Validating domain-intensive cloud applications presents a significant challenge because of the complexity of both the problem domain and the underlying cloud platform. In this paper, we describe an approach that leverages model-driven engineering to improve testing domain-intensive cloud applications. Our approach combines a set of abstract test commands with various domain and configuration models to define a domain-specific testing language. We have developed a prototype of our approach that provides language editing and platform configuration tools to aid test specification, execution and debugging.

Keywords: Testing, Model-Driven Engineering, Domain-Specific Languages, Cloud Computing, Human Capital Management Software

1 Introduction

Validating software-as-a-service applications is difficult due to the large size of the problem domain, coupled with the complexity of the underlying cloud platform. Adequate functional testing is heavily dependent on domain expertise from each product area, and typically requires extensive data setup. Since the software is delivered as a service over the Internet, functional UI testing must be performed using different browsers to ensure a good user experience. In addition, engineers need to be able to set up tests to run on specific configurations of the underlying cloud infrastructure.

Model-driven engineering (MDE) seeks to simplify software development by raising the level of abstraction through domain modeling, while promoting communication between groups working on the same system [1]. Researchers and practitioners have developed a number of MDE tools and techniques, most of which have focused on exploiting domain models for automatic code generation. However, there has also been research on MDE approaches to enhance software

testing [2, 3]. A more recent interest that has arisen is the use of MDE to support emerging paradigms such as adaptive and cloud computing [4].

In this paper, we describe an approach that leverages MDE to improve the specification, execution, and debugging of functional tests for domain-intensive cloud applications. Our approach is the result of investigating new and innovative ways to test UltiPro, a comprehensive cloud-based human capital management (HCM) solution [5]. Domain models and abstractions for common UI interactions, data setup, environment and platform configurations are combined with highly extensible testing frameworks [6–8]. The result is a powerful domain-specific language (DSL) for creating automated functional tests. Our test authoring DSL is supported by an editor that provides syntax checking and highlighting, intelli-sense, tooltips, and debugging features.

The major contributions of this research paper are as follows: (1) describes a novel approach that integrates various domain and configuration models into a test case specification language for cloud applications; (2) presents the design of a prototype used to demonstrate the feasibility of the approach; and (3) discusses our experience developing the prototype, focusing on the lessons learned. The rest of this paper is organized as follows: the next section motivates the research problem. Section 3 describes our domain-specific test case specification approach for cloud applications. Section 4 presents a prototype that implements the proposed approach. Section 5 discusses the lessons learned from building the prototype. Section 6 is the related work and Section 7 concludes the paper.

2 Motivation

Our research has been motivated by the challenges faced when testing UltiPro [5]. Delivered on-demand as software-as-a-service in the cloud, UltiPro provides HCM functionality including recruitment, onboarding, payroll, payment services, benefits, compensation management, performance management and reviews, succession planning, and more. Data is available across all areas of HCM, and can be accessed by department, division, or country. Several reporting and analysis features are also available through UltiPro's web-based portal.

The large size and complexity of the problem domain makes testing the functionality of UltiPro challenging. Individual product areas (e.g., recruitment, payroll) encompass so many features that each area could be considered as a product itself. Adequately testing UltiPro requires each product area to be validated, which is impossible without domain expertise. Although each product area is large, UltiPro has been designed and developed as a unified solution which seamlessly integrates all aspects of HCM. Validation of the overall product therefore relies heavily on the collaboration of domain experts across all product areas. This ensures that changes to one product area does not have an adverse effect on other product areas.

Testing UltiPro is further complicated because of its development and delivery as a cloud application service. Cloud application services are hosted on complex, distributed infrastructures with multiple servers and architectural lay-

ers that extend from the underlying network up to the web-based user interface. To ensure a good user experience, functional UI testing must be performed using different web browsers. Other non-functional factors such as high performance and security requirements also make it difficult to test cloud applications. However, this paper limits the scope of the testing problem for cloud applications to functional UI and platform compatibility testing.

3 Approach

Our approach defines a test specification language that can be used to develop automated tests for a particular application domain. As shown in Figure 1, we leverage abstract test commands, domain and platform models, and test automation frameworks for the purpose of creating a domain-specific language (DSL) for testing cloud applications. The DSL allows us to provide test case editing, execution, and debugging tools tailored for domain experts, test engineers, and end users. Abstract tests defined using the DSL are translated into executable scripts that run on the underlying testing tools and frameworks. For the remainder of this section, we describe the various test abstractions and models used in our approach. Transformation of abstract tests into tool-specific testing scripts is discussed as part of the prototype design in Section 4.

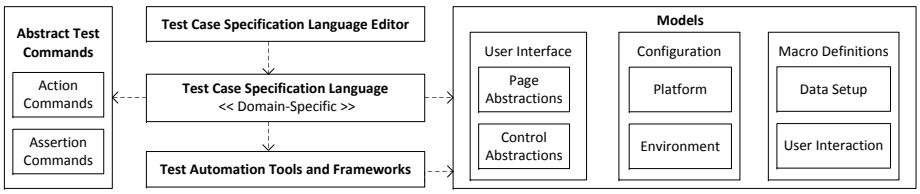


Fig. 1. A Domain-Specific Testing Language for Cloud Applications

3.1 Abstract Test Commands

At the core of the language is a set of abstract test commands. There are two types of commands: `Action Commands` and `Assertion Commands` (left of Figure 1). Action commands apply inputs that exercise the system under test (SUT). This includes stimulating UI controls such as textboxes, dropdowns, and buttons, as well as database-related actions. On the other hand, assertion commands perform UI and database checks to verify the behavior of the SUT. Table 1 describes some of the key test commands defined under our approach.

3.2 Application Domain Models

Domain concepts are introduced into our testing language through two types of models: *User Interface* and *Macro Definitions* (right of Figure 1).

	Command	Description
Actions	Blur	Loses focus of an element
	Clear	Empties the value of an element
	Click	Presses and releases mouse button while hovering over an element
	Mouse Over	Hovers the mouse pointer over an element
	Mouse Out Of	Move the mouse pointer from within the hover area of an element
	Set	Assigns a value to an element
Assertions	Is	Checks if the value of an element equals a given value
	Is Like	Checks if the value of an element contains a given value as a substring
	Is Visible	Checks if an element is visible
	Is Enabled	Checks whether an element used for input is enabled
	Exists	Checks if an element is present
	Has Options	Checks if a dropdown provides a specified list of values
	Has Number Of Options	Checks if the size of a dropdown list is equal to a specified value

Table 1. Web UI Abstract Test Commands

User Interface This model is a generalization of the user interface of the SUT. For example, in the case of UltiPro the UI model consists of abstractions representing its web pages and controls. These page and control objects encapsulate the CSS selectors used to identify web elements in the HTML source document tree. Automated tests then reference these abstractions instead of the implementation details, which makes tests easier to maintain as the application changes [8, 9]. Furthermore, page and control objects are named using domain-specific concepts. For example, a grid control used for entering pay data would be named `PayDataEntryGrid`. Using such terms allows domain experts and end users to easily identify and specify various aspects of the SUT.

Macros A macro in computer science is a pattern that specifies how a sequence of inputs is mapped to a replacement input sequence. Macros are often used to make programming tasks less repetitive and less error-prone. Our approach leverages the benefits of macros to improve test specification. Testers can define frequently used test setup, input, or assertion command sequences, and store them in a central location. These macros are then named using domain-specific terms, and integrated into our testing language. Similar to our abstract test commands, test macros can target user or database interactions.

3.3 Configuration Models

Several abstractions for configuring the underlying platform and environment of the SUT are integrated into our language. These include abstractions for: *Server Environment Configuration* – Application servers, web servers, database servers, reporting servers that make up the test environment; *OS/Platform Configuration* – Operating systems on which to test the desktop and mobile versions of the cloud application; *Web Browser Configuration* – Clients to use during cross-browser compatibility testing, e.g., Chrome, Firefox, Internet Explorer or a combination thereof; and *Test Harness Configuration* – Modes and settings that allow users to tweak aspects of test execution including timing characteristics, logging, among others.

```
1  Summary
2      Purpose   Validate Run Payroll Feature
3      Authors   Dionny Santiago, Adam Cando
4      Configs   .NET, Payroll-14
5
6  Declarations
7      UltiPro UltiPro
8
9  Setup
10     Given I Setup a Payroll (...)
11     And launch UltiPro
12     And login as a Payroll Administrator
13       By setting the UserNameTextBox to "admin",
14                     PasswordTextBox to "ulti"
15       And clicking on the LoginButton
16
17 Tests
18     Scenario: Payroll Should Run to Completion
19     Given I Navigate to the Payroll Overview
20     When I click the StartPayrollButton
21     Then the ProcessStatusLabel is "Completed."
22     And the CreateBatchesButton is enabled
```

Fig. 2. Example Test Case Specification

3.4 Illustrative Example

Figure 2 presents an example test case specification defined using our approach. The example test consists of four main blocks: Summary, Declarations, Setup, and Tests. The summary block (Lines 1-4) holds meta information about the test, which includes a purpose, authors, and various configurations. In Line 4, `.NET` is a configuration that runs tests against the UltiPro desktop web application using three browsers for compatiblity testing. On the same line, `Payroll-14` sets up the test to run on a server environment configured for payroll processing.

Applications and data that will be used throughout the test must appear in the declarations block (Lines 6-7). In this block, automatic word completion popups (i.e., intelli-sense) is filtered to a list of available application models and database types. Line 7 of Figure 2 specifies that the UltiPro application model will be referenced whenever the name `UltiPro` appears in the test.

The setup block (Lines 9-15) contains a set of preconditions for the test, written using a behavioral-driven development (BDD) style syntax. Line 10 illustrates the use of a database macro to `Setup a Payroll`. Note that elipses are used to mask the actual parameters passed into the macro. Inline, users can declare high-level test steps (Line 12), and define how those steps are implemented as actions or assertions on the application model (Lines 13-15).

Test cases appear within the block named `Tests` (Lines 17-22). Line 18 provides a name for the single test in the example, while Line 19 demonstrates the usage of a UI navigation macro. The `click` action command is illustrated in Line 20, while Lines 21 and 22 show the `Is` and `Is Enabled` assertion commands.

4 Prototype

This section presents the setup, design and implementation of the Legend prototype, which was developed to demonstrate the feasibility of the proposed approach. Legend primarily consists of tools for authoring, executing and debugging tests written to validate UltiPro [5].

Legend has been developed in C# as a Visual Studio (VS) extension. The VS 2010 SDK provides components for extending the VS Editor with a custom language service. The Legend language service supports many of the VS SDK features including syntax coloring, error highlighting, intelli-sense, outlining, tooltips, and debugging. Integration of application and configuration models using domain concepts is a novel feature of Legend that separates it from other test specification languages and tools.

The underlying testing framework used to run Legend tests is an in-house tool called Echo [8]. Echo was developed based on Selenium, a cross-browser web UI automated testing framework [7]. Page and control abstractions for the application domain model are defined using Echo, in accordance with the page object pattern [9]. Frameworks such as MbUnit [6] are used to provide capabilities such as test fixture setup and teardown, data-driven testing, and reporting. The custom tooling developed for the prototype is divided into two categories: *Editing Tools* and *Configuration Tools*.

Editing Tools Figure 3 provides a UML package diagram showing the design of Legend DSL Editor. As shown in Figure 3, the editor is comprised of three major packages: `EditorExtension`, `ApplicationModelIntegration` and `CodeGeneration`. Key classes from each package, along with their interdependency relationships are also shown in the diagram.

The `EditorExtension` subsystem (top of Figure 3) contains the components that implement token colorization, syntax checking, block outlining, and intellisense. This subsystem is the main point of interaction between the Visual Studio editor and the Legend code extensions. The classes with the stereotypes `Providers`, `Controllers`, `Taggers`, and `Sources` are derived from the Visual Studio SDK, and directly interact with the .NET Managed Extensibility Framework (MEF) [10]. Classes stereotyped `Augmentors` and `Services` represent our custom extensions. The `LanguageService` class coordinates several of the interactions between the editor and the application models.

Integration with the application model is achieved via the `ApplicationModelIntegration` subsystem (bottom-left of Figure 3). It contains two types of classes: `ModelProviders` and `Models`. The `ModelProvider` classes use reflection to read the page objects, control objects, macros and elements that make up the `Models`. It is also responsible for filtering intelli-sense on the model, given a specific test context. For example, at the point where a test declares access to a particular web page, the `ModelProvider` scopes the word completion picklist for elements to include only elements that appear on that page.

Lastly, the `CodeGeneration` subsystem (bottom-right of Figure 3) provides logic for translating the abstract tests written in Legend into code executable

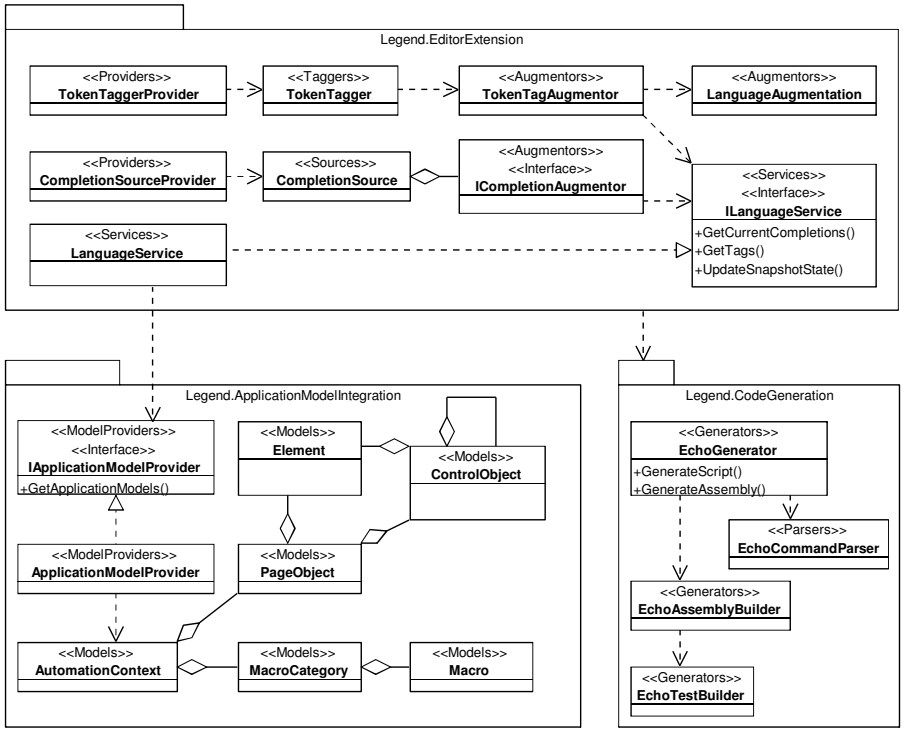


Fig. 3. Diagram showing the package structure and class design of the prototype

by the Echo framework. In order to support debugging at multiple levels of abstraction, the `EchoGenerator` has two modes of generation: *Script* – generates test scripts written in the Echo syntax; and *Assembly* – generates a Common Intermediate Language (CIL) representation of the Echo test. Our prototype maps the CIL to the domain-specific test steps. A test can therefore be executed and debugged at the level of the domain-specific test language, or the script language of the underlying Echo testing framework.

Configuration Tools Abstractions for the test environment and platform configurations are implemented in two distinct XML files called `Environments.xml` and `Parameters.xml`. The environments XML contains a list of all the test environments that are connected to a tool that can automatically request the latest UltiPro build. The file is auto-generated and populated with the unique identifiers used by each team to refer to their test environments. Information related to the specific network and database servers, along with any credentials for authentication are also stored in the `Environments.xml` file.

The `Parameters.xml` allows users to specify a range of configurations ranging from desktop web settings such as browsers and languages, to mobile web settings such as device screen size and orientation. Although each team can create their own configurations, there are a set of fixed configurations that are available for

use across all teams during regression testing. The data in both the environments and parameters XML files are passed directly to the Echo testing framework prior to test execution.

5 Lessons Learned

A key factor that contributed to the successful development of the prototype was having robust, highly extensible and configurable underlying testing frameworks. Echo and Selenium provided the implementations to support many of the abstractions described in the approach. Building the application models required a collaborative effort among developers, testers and domain experts. Developers would create the page objects and control objects, while testers and domain experts made sure they were named and exposed appropriately for testing. Macro creation was primarily done by testers and domain experts, with occasional assistance from the developers if necessary.

One of the more complex aspects of the prototype implementation was the module for keeping track of the test context in order to filter intelli-sense. This required the creation of a state-based rules engine to allow us to perform different editor actions based on previously specified lines in the test. Although challenging to implement, this feature was necessary to provide meaningful intelli-sense that guides testers during test creation. In other words, testers are only presented with commands, model elements, and other keyword suggestions if they are applicable in the current context.

Based on initial responses to prototype demonstrations, a major benefit of Legend is the ease with which test cases can be specified and reviewed by non-technical users. The tool can therefore be leveraged by domain experts and end users during acceptance testing. It also allows these stakeholders to assist in debugging issues using a language they understand, and without being concerned with the low-level implementation details of the test automation. Since tests are specified in an english-like syntax, using Legend could reduce or eliminate the need to maintain a separate inventory of test documentation. However, further evidence through a case study or empirical evaluation is needed to be able to fully validate these claims.

A limitation of the current prototype is the lack of an externalized point of extension for the test commands and their syntax. Since domain experts, testers and developers from several teams will be defining new model elements as the application evolves, we need to provide a mechanism that allows new page or control-specific commands to be easily added to the language. Web UI elements with dynamically generated identifiers are also not supported by the prototype, or the underlying Echo testing framework, but are planned for future releases.

6 Related Work

Although the use of domain modeling to support software engineering is not new, only a few researchers have leveraged MDE and DSLs to support software

testing [2, 3, 11]. Kanstren and Puolitaival [3] present the OSMOTester approach and tool that is very related to our work. OSOMOTester automatically generates tests with domain-specific concepts. A domain expert is used to construct a test model of the system, which is combined with a domain-specific modeling language that constrains and guides test case generation.

Hernandez et al. [2] describe a model-driven technique for designing platform independent tests for validating web-based applications. These platform independent tests are then combined with a model of the web technologies used to implement the application, and generate platform-specific tests [2]. Kuhn and Gotzhein [11] present an approach that uses configurable simulations to do platform-specific testing. They extend the UML testing profile to include platform models for real deployments, and describe how to use these models to test embedded systems by simulating various hardware configurations.

Some researchers have proposed model-driven approaches that aid the development of high-performance and cloud computing systems [12]. Palyart and Lugato define a high-performance computing modeling language (HPCML). HPCML provides constructs for specifying different concerns in high-performance scientific computing such as mathematics, parallelism, and validation. Nagel et al. [13] introduce a meta-model for specifying bindings between business processes and cloud services considering service-level agreements. They further extend that meta-model to support dynamic adaptation of cloud-based services.

There are several general purpose behavioral-driven development (BDD) testing tools that help to tie acceptance tests to business requirements [14–16]. Similar to Legend, such tools aim to bridge the gap between domain experts, developers, and testers [16]. These tools typically work by creating and linking two sets of files – specifications and step definitions [15]. Legend combines these two activities into a single, domain-specific test authoring experience. Our research on Legend extends previous work on the Echo Web UI Test Automation Framework [8]. Echo provides a thin layer of abstraction on top of Selenium [7], and adds several features including command timeouts, wait throttling between commands, database interaction, and environment and test configuration.

7 Conclusion

The work in this paper presented an approach that applied model-driven engineering to the development of a domain-specific test case specification language. Our approach is general in the sense that it can be applied to any domain, but in terms of technologies we focus on web-based applications which are deployed on cloud computing platforms. We have implemented a prototype of the proposed approach for a cloud-based human capital management solution. Developing the prototype gave us first-hand experience on some of the benefits and challenges associated with creating a DSL for testing UltiPro. Feedback from interactive prototype demonstrations has been positive. Our next steps are to develop a full implementation of Legend, and perform a case study using data from UltiPro.

Acknowledgments. The authors would like to thank Jorge Martinez, Michael Mattera, and members of the Virtual Team at Ultimate Software for their contributions to this work. We also give thanks to the judges and participants of the Summer 2012 Ultimate Software 48 Hours Project for their valuable feedback. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the Ultimate Software Group, Inc.

References

1. Stahl, T., Voelter, M., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons (2006)
2. Hernandez, Y., King, T.M., Pava, J., Clarke, P.J.: A meta-model to support regression testing of web applications. In: SEKE. (2008) 500–505
3. Kanstrén, T., Puolitaival, O.: Using built-in domain-specific modeling support to guide model- based test generation. In: MBT. (2012) 58–72
4. Brunelière, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: Modeling, Design, and Analysis for the Service Cloud - MDA4ServiceCloud'10, Paris, France (June 2010)
5. Ultimate Software: Human Capital Management Solutions: Ultipro Enterprise (July 2013) `www.ultimatesoftware.com/solution`.
6. Jeff Brown: MbUnit Test Framework `http://mbunit.com/` (July 2013).
7. Stewart, S., Huggins, J.: Selenium - Web Browser Automation `http://docs.seleniumhq.org/` (July 2013).
8. Virtual Team: Echo Web UI Test Automation Framework. Technical report, Ultimate Software Group, Inc. (October 2010)
9. Wilk, J.: Page Object Pattern (March 2012) `http://blog.josephwilk.net/cucumber/page-object-pattern.html` (July 2013).
10. Microsoft: MSDN - Visual Studio: Extending the Editor (July 2013) `http://msdn.microsoft.com/en-us/library/dd885242.aspx`.
11. Kuhn, T., Gotzhein, R.: Model-driven platform-specific testing through configurable simulations. In: Model Driven Architecture - Foundations and Applications. Volume 5095 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 278–293
12. Palyart, M., Ober, I., Lugato, D., Bruel, J.M.: HPCML: a modeling language dedicated to high-performance scientific computing. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and Cloud computing. MDHPCL '12, New York, NY, USA, ACM (2012) 6:1–6:6
13. Nagel, B., Gerth, C., Yigitbas, E., Christ, F., Engels, G.: Model-driven specification of adaptive cloud-based systems. In: Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and Cloud computing. MDHPCL '12, New York, NY, USA, ACM (2012) 4:1–4:6
14. Chelimsky, D., Myron Marston, M., Lindeman, A., Rowe, J.: RSpec - BDD framework for the Ruby Programming Language (December 2010) `http://rspec.info` (July 2013).
15. Hellesoy, A., Wynne, M.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. Pragmatic Programmers. Pragmatic Bookshelf (2012)
16. Nagy, G., Bandi, J., Hassa, C.: SpecFlow: Pragmatic BDD for .NET (November 2009) `http://www.specflow.org/specflownew/` (July 2013).

Architecture Framework for Mapping Parallel Algorithms to Parallel Computing Platforms

Bedir Tekinerdogan[1], Ethem Arkın[2]

[1]Bilkent University, Dept. of Computer Engineering, Ankara, Turkey
bedir@cs.bilkent.edu.tr
[2]Aselsan MGEO, Ankara, Turkey
earkin@aselsan.com.tr

Abstract. Mapping parallel algorithms to parallel computing platforms requires several activities such as the analysis of the parallel algorithm, the definition of the logical configuration of the platform, and the mapping of the algorithm to the logical configuration platform. Unfortunately, in current parallel computing approaches there does not seem to be precise modeling approaches for supporting the mapping process. The lack of a clear and precise modeling approach for parallel computing impedes the communication and analysis of the decisions for supporting the mapping of parallel algorithms to parallel computing platforms. In this paper we present an architecture framework for modeling the various views that are related to the mapping process. An architectural framework organizes and structures the proposed architectural viewpoints. We propose five coherent set of viewpoints for supporting the mapping of parallel algorithms to parallel computing platforms. We illustrate the architecture framework for the mapping of array increment algorithm to the parallel computing platform.

Keywords: Model Driven Software Development, Parallel Programming, High Performance Computing, Domain Specific Language, Modelling.

1 Introduction

It is now increasingly acknowledged that the processing power of a single processor has reached the physical limitations and likewise serial computing has reached its limits. To increase the performance of computing approaches the current trend is towards applying parallel computing on multiple nodes typically including many CPUs. In contrast to serial computing in which instructions are executed serially, in parallel computing multiple processing elements are used to execute the program instructions simultaneously.

One of the important challenges in parallel computing is the mapping of the parallel algorithm to the parallel computing platform. The mapping process requires several activities such as the analysis of the parallel algorithm, the definition of the logical configuration of the platform, and the mapping of the algorithm to the logical configuration platform. Based on the analysis of the algorithm several design decisions for allocating the algorithm sections to the logical configurations must be made. To support the communication among the stakeholders, to reason about the design decisions during the mapping process and to analyze the eventual design it is important to adopt the appropriate modeling approaches. In current parallel computing approaches there does not seem to be standard modeling approaches for supporting the mapping process. Most approaches seem to adopt conceptu-

al modeling approaches in which the parallel computing elements are represented using idiosyncratic models. Other approaches borrow for example models from embedded and real time systems and try to adapt these for parallel computing. The lack of a clear and precise modeling approach for parallel computing impedes the communication and analysis of the decisions for supporting the mapping of parallel algorithms to parallel computing platforms.

In this paper we present an architecture framework for modeling the various views that are related to the mapping process. An architectural framework organizes and structures the proposed architectural viewpoints. We propose five coherent set of viewpoints for supporting the mapping of parallel algorithms to parallel computing platforms. We illustrate the architecture framework for the mapping of parallel array increment algorithm to the parallel computing platform.

The remainder of the paper is organized as follows. In section 2, we describe the background on software architecture viewpoints and define the parallel computing metamodel. Section 3 presents the viewpoints based on the defined metamodel. Section 4 presents the guidelines for using the viewpoints. Section 5 presents the related work and finally we conclude the paper in section 6.

2 Background

In section 2.1 we provide a short background on architecture viewpoints which is necessary for defining and understanding the viewpoint approach. Subsequently, in section 2.2 we provide the metamodel for parallel computing that we will later use to define the architecture viewpoints in section 3.

2.1 Software Architecture Viewpoints

To represent the mapping of parallel algorithm to parallel computing platform it is important to provide appropriate modeling approaches. For this we adopt the modeling approaches as defined in the software architecture design community. According to ISO/IEC 42010 the notion of *system* can be defined as a set of components that accomplishes a specific function or set of functions[4]. Each system has an architecture, which is defined as "the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution". A common practice to model an architecture of a software intensive system is to adopt different architectural views for describing the architecture according to the stakeholders' concerns [2]. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern. An architectural viewpoint defines the conventions for constructing, interpreting and analyzing views. Architectural views conform to viewpoints that represent the conventions for constructing and using a view. An *architectural framework* organizes and structures the proposed architectural viewpoints [4]. The concept of *architectural view* appears to be at the same level of the concept of *model* in the model-driven development approach. The concept of *viewpoint*, representing the language for expressing views, appears to be on the level of metamodel. From the model-driven development perspective, an architecture framework as such can be considered as a coherent set of domain specific languages [9]. The notion of architecture framework and the viewpoints plays an important role in modeling and documenting architectures. However, the existing approaches on architecture modeling seem to

have primarily focused on the domain of traditional, desktop-based and sometimes distributed development platforms. Parallel computing systems have not been frequently or explicitly addressed.

2.2 Parallel Computing Metamodel

Fig. 1 shows the abstract syntax of the metamodel for mapping parallel algorithms to parallel computing platform. The metamodel consists of four parts including *Parallel Algorithm, Physical Configuration, Logical Configuration,* and *Code.* In the *Parallel Algorithm* part we can observe that an *Algorithm* consists of multiple *Sections,* which can be either *Serial Section* or *Parallel Section.* Each section is mapped on *Operation* which on its turn is mapped on *Tile.*

Physical Configuration represents the physical configuration of the parallel computing platform and consists of *Network* and *Nodes. Network* defines the communication medium among the *Nodes. Node* consists of *Processing Unit* and *Memory*. Since a node can consist of multiple processing units and memory units we assume that different configurations can be defined including shared memory and distributed memory architectures. *Logical Configuration* represents a model of the physical configuration that defines the logical communication structure among the physical nodes. *LogicalConfiguration* consists of a number of *Tiles*. *Tile* can be either a (single) *Core*, or *Pattern* that represents a composition of tiles. Patterns are shaped by the operations of the sections in the algorithm. *Pattern* includes also the communication links among the cores. The algorithm sections are mapped to *CodeBlocks.* Hereby, *SerialSection* is implemented as *SerialCode*, and *ParallelSection* as *ParallelCode.* Besides of the characteristic of *ParallelSection* the implementation of *ParallelCode* is also defined by *Pattern* as defined in the logical configuration. The overall *Algorithm* is run on *PhysicalConfiguration.*

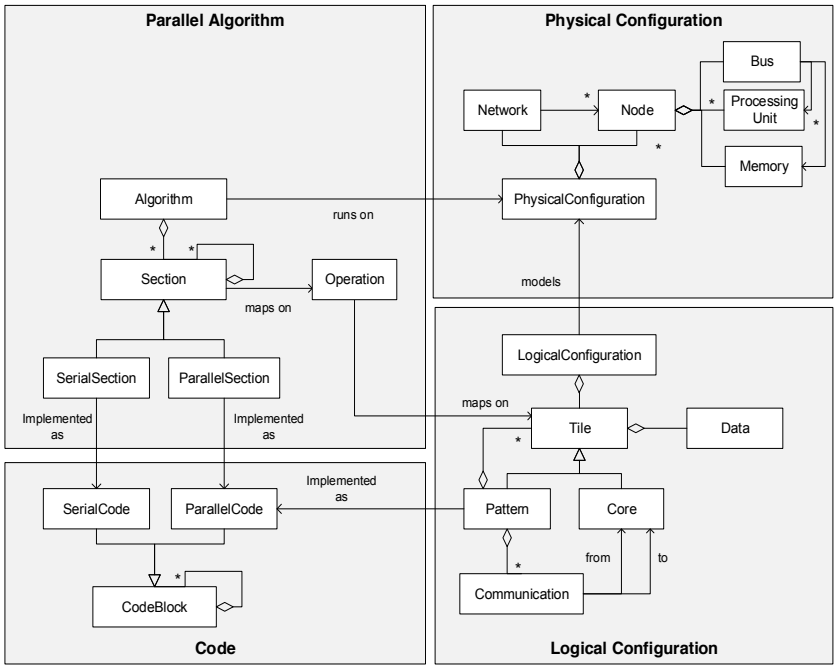


Fig. 1. Metamodel for Mapping Parallel Algorithm to Parallel Computing Platform

3 Architecture Viewpoints for Parallel Computing

Based on the metamodel of Fig. 1 we define the architecture framework consisting of a coherent set of viewpoints for supporting mapping of parallel algorithms to parallel computing platform. In section 3.1 we will first describe an example parallel algorithm and the corresponding logical configuration. In section 3.2 we will present the *algorithm decomposition viewpoint*. In section 3.3 we will present *the physical configuration viewpoint*. Section 3.4 presents the logical configuration viewpoint, section 3.5 the algorithm-to-logical configuration viewpoint, and finally, section 3.6 will present the algorithm-to-code viewpoint.

3.1 Case Description

To illustrate the problem we will use the array increment algorithm as shown in Fig. 2 that will be mapped on a 4x4 physical parallel computing architecture. Given an array the algorithm recursively decomposes the array into sub-arrays to increment each element with one. The algorithm is actually composed of two different parts. In the first part the array element is incremented with one if the array size is one (line 3). If the array size is greater than one then in the second part the array is decomposed into two sub-arrays and the algorithm is recursively called.

```
1.  Procedure ArrayInc(A[], n):
2.  if n=1 then
3.    *A += 1
4.  else
5.    ArrayInc(A, n/2)
6.    ArrayInc(A+n/2, n)
7.  endif
```

Fig. 2. Array Increment Algorithm

3.2 Algorithm Decomposition Viewpoint

In fact the array increment algorithm is a serial (recursive) algorithm. To increase the time performance of the algorithm we can map it to a parallel computing platform and run it in parallel. For this it is necessary to decompose the algorithm into separate sections and define which sections are serial and which can be run in parallel. Further, each section of an algorithm realizes an operation, which is a reusable abstraction of a set of instructions. For serial sections the operation can be custom to the algorithm. For parallel sections in general we can identify for example the primitive operations *Scatter* for distributing data to other nodes, *Gather* for collecting data from nodes, *Broadcast* for broadcasting data to other nodes, etc. Table 1 shows the algorithm decomposition viewpoint that is used to decompose and analyze the parallel algorithm. The viewpoint is based on the concepts of the *Parallel Algorithm* part of the metamodel in Fig. 1. An example algorithm decomposition view that is based on this viewpoint is shown in Fig. 3A. Here we can see that the array increment algorithm has been decomposed into four different sections with two serial and two parallel sections. Further, for each section we have defined its corresponding operation.

3.3 Physical Configuration Viewpoint

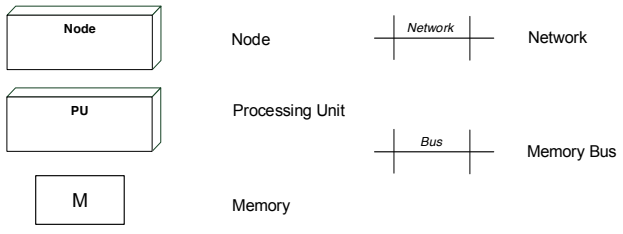
Table 2 shows the physical configuration viewpoint for modeling the parallel computing architecture. The viewpoint is based on the concepts of the *Physical Configuration* part of

the metamodel in Fig. 1. As we can see from the table the viewpoint defines explicit notations for *Node, Processing Unit, Network, Memory Bus* and *Memory.* An example physical configuration view that is based on this viewpoint is shown in Fig. 3B. Here the physical configuration consists of four nodes interconnected through a network. Each node has four processing units with a shared memory. Both the nodes and processing units are numbered for identification purposes.

Table 1. Algorithm Decomposition Viewpoint

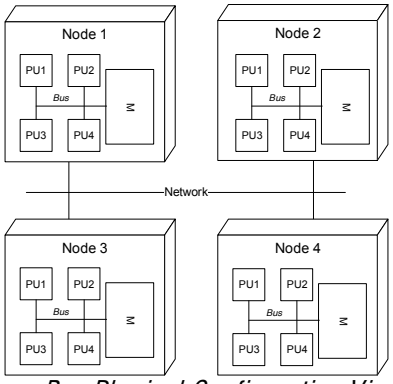
Name	Algorithm Decomposition Viewpoint
Concerns	Decomposing an algorithm into different sections which can be either serial or parallel. Analysis of the algorithm.
Stakeholders	Algorithm analysts, logical configuration architect, physical configuration architect
Elements	• Algorithm – represents the parallel algorithm consisting of sections. • Serial Section – a part of an algorithm consisting of a coherent set of instructions that needs to run in serial • Parallel Section – a part of an algorithm consisting of a coherent set of instructions that needs to run in parallel • Operation – abstract representation of the set of instructions that are defined in the section
Relations	• Decomposition relation defines the algorithm and the sections
Constraints	• A section can be either SER or PAR, not both
Notation	*Index* | *Algorithm Section* | *Section Type* | *Operation*

Table 2. Physical Configuration Viewpoint

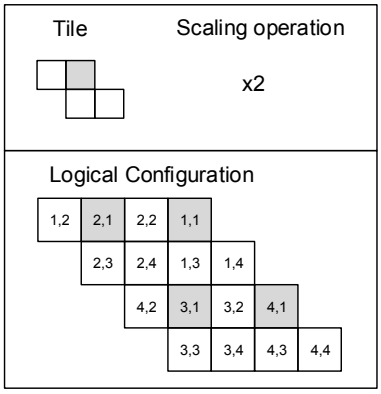
Name	Physical Configuration Viewpoint
Concerns	Defining physical configuration of the parallel computing platform
Stakeholders	Physical configuration architect
Elements	• Node – A standalone computer usually comprised of multiple CPUs/ /cores, memory, network interfaces, etc. • Network – medium for connecting nodes • Memory Bus – medium for connecting processing units within a node • Processing Unit – processing unit that reads and executes program instructions • Memory – unit for data storage
Relations	• Nodes are networked together to comprise a supercomputer • Processing units are connected through a bus
Constraints	• Processing Units can be allocated to Nodes only • Memory can be shared or distributed
Notation	

Ind.	Algorithm Section	Section Type	Operation
1	A1 = [A, A+n/2] A2 = [A+n/2, A+n]	SER	*Decompose*
2	Distribute the sub-arrays	PAR	*Scatter*
3	*A += 1	SER	*Increment*
4	Collect increment-ed array results	PAR	*Gather*

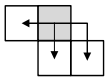
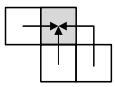
A. Algorithm Decomposition View



B. Physical Configuration View



C. Logical Configuration View

Ind.	Algorithm Section	Plan
1	A1=[A, A+n/2] A2=[A+n/2, A+n]	Run on each node
2	Distribute the sub-arrays	
3	*A += 1	Run on each node
4	Collect incre-mented array results	

D. Algorithm-to-Logical Configuration View

NO	Algorithm Section	Code Block
1	A1=[A, A+n/2] A2=[A+n/2, A+n]	A1 = [A, A+n/4] A2 = [A+n/4, A+n/2] A3 = [A+n/2, A+3n/4] A4 = [A+3n/4, A+n]
2	Distribute the sub-arrays	Will be generated
3	*A += 1	*A += 1
4	Collect incremented array results	Will be generated

E. Algorithm-to-Code View

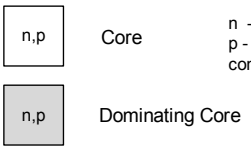
Fig. 3. Views for the given case using the defined viewpoints

3.4 Logical Configuration Viewpoint

Table 3 shows the logical configuration viewpoint for modeling the logical configuration of the parallel computing architecture. The viewpoint is based on the concepts of the *Logical Configuration* part of the metamodel in Fig. 1. The viewpoint defines explicit notations

for *Core, Dominating Core, Tile, Pattern* and *Communication.* An example logical configuration view that is based on this viewpoint is shown in Fig. 3C. The logical configuration is based on physical configuration as shown in Fig. 3B and shaped according to the algorithm in Fig. 3A. As we can observe from the figure the 16 cores in the four nodes of the physical configuration are now rearranged to implement the algorithm properly. Each core is numbered based on both the node number and core number in the physical configuration. For example, core (2,1) refers to the processing unit 1 of physical node 2. Typically, for the same physical configuration we can have many different logical configurations each of them indicating different communication and exchange patterns of data among the cores in the nodes. In our earlier paper we define an approach for deriving the feasible logical configurations with respect to speed-up and efficiency metrics [1]. In this paper we assume that a feasible logical configuration is selected. For very large configurations such as in exascale computing [5] it is not feasible to draw this on the same scale. Instead we define the configuration as consisting of a set of tiles which are used to generate the actual logical configuration. In the example of Fig. 3C we can see that the configuration can be defined as a tile that is two times recursively scaled to generate the logical configuration. For more details about the scaling process we refer to our earlier paper [1].

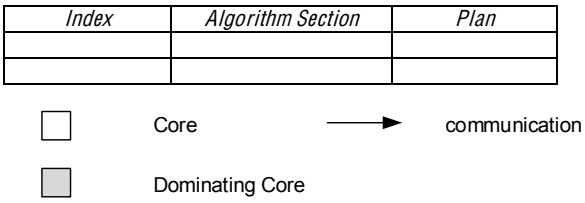
Table 3. Logical Configuration Viewpoint

Name	Logical Configuration Viewpoint
Concerns	Modeling of the logical configuration for the physical configuration
Stakeholders	Logical configuration architect
Elements	• Core – model of processing unit • Dominating Core – the processing unit that is responsible for exchanging data with other nodes
Relations	• Cores can be composed into larger tiles • Tiles can be used to define/generate logical configuration
Constraints	• The number of cores should be equal to the processing units in the physical configuration • The numbering of the cores should match the numbering in the physical configuration
Notation	

3.5 Algorithm-to-Logical Configuration Mapping Viewpoint

The logical configuration view represents the static configuration of the nodes to realize the parallel algorithm. However, it does not illustrate the communication patterns among the nodes to represent the dynamic behavior of the algorithm. For this the algorithm-to-logical configuration mapping viewpoint is used. The viewpoint is illustrated in Table 4. For each section we describe a plan that defines on which nodes the corresponding operation of the section will run. For serial sections usually this is a custom operation. For parallel section the plan includes the communication pattern among the nodes. A communication pattern includes communication paths that consist of a source node, a target node and a route between the source and target nodes. An algorithm-to-logical configuration mapping view is represented using a table as it is, for example, shown in Fig. 3D.

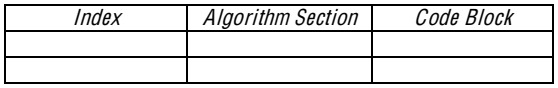
Table 4. Algorithm-to-Logical Configuration Mapping Viewpoint

Name	Algorithm-to-Logical Configuration Mapping Viewpoint
Concerns	Mapping the communication patterns of the algorithm to the logical configuration
Stakeholders	System Engineers, Logical configuration architect
Elements	• Section – a part of an algorithm consisting of a coherent set of instructions. A section is either serial or parallel • Plan – the plan for each section to map the operations to the logical configuration units • Core – model of processing unit • Dominating Core – the processing unit that is responsible for exchanging data with other nodes • Communication – the communication pattern among the different cores
Relations	• Mapping of plan to section
Constraints	• Each serial section has a plan that defines the nodes on which it will run • Each parallel section has a plan that defines the communication patterns among nodes
Notation	

3.6 Algorithm-to-Code Viewpoint

Once the logical configuration and the corresponding algorithm section allocation plan has been defined, the implementation of the algorithm can be started. The corresponding viewpoint is shown in Table 5. The viewpoint is based on the concepts of the *Parallel Algorithm* and *Code* parts of the metamodel in Fig. 1. An example algorithm decomposition view that is based on this viewpoint is shown in Fig. 3E.

Table 5. Algorithm-to-Code Mapping Viewpoint

Name	Algorithm-to-Code Mapping
Concerns	Mapping the algorithm sections to code
Stakeholders	Parallel Programmer, System Engineer
Elements	• Algorithm – represents the parallel algorithm consisting of sections. • Section – a part of an algorithm consisting of a coherent set of instructions • Code Block – code for implementing the section
Relations	• Realization of the section to code
Constraints	• Each section has a code block
Notation	

4 Guidelines for Adopting Viewpoints

In the previous section we have provided the architecture framework consisting of a coherent set of viewpoints for supporting the mapping of parallel algorithms to parallel com-

puting platforms. An important issue here is of course the validity of the viewpoints. In general evaluating architecture viewpoints can be carried out from various perspectives including the appropriateness for stakeholders, the consistency among viewpoints, and the fitness of the language. We have evaluated the architecture framework according the approach that we have described in our earlier study [9]. Fig. 4 shows the process as a UML activity for adopting the five different views. The process starts initially with the definition of algorithm to decomposition view and the physical configuration view, which can be carried out in parallel. After the physical configuration view is defined the logical configuration view can be defined, followed by the modeling of the algorithm to logical view, and finally the algorithm to code view. Among the different steps several iterations can be required which is shown by the arrows.

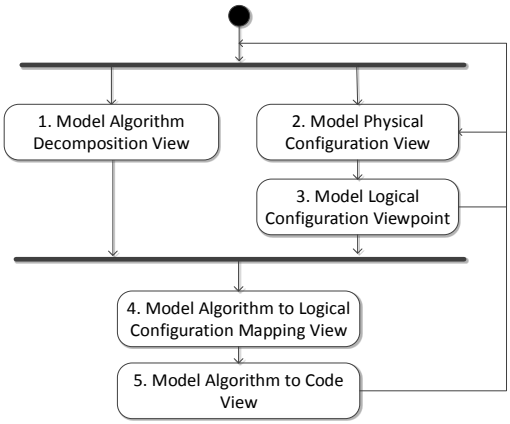


Fig. 4. Approach for Generating/Developing and Deployment of Parallel Algorithm Code

5 Related Work

In the literature of parallel computing the particular focus seems to have been on parallel programming models such as MPI, OpenMP, CILK etc. [8] but the design and the modeling got less attention. Several papers have focused in particular on higher level design abstractions in parallel computing and the adoption of model-driven development.

Several approaches have been provided to apply model-driven development to high performance computing. Similar to our approach Palyart et. al. [7] propose an approach for using model-driven engineering in high performance computing. They focus on automated support for the design of a high performance computing application based on abstract platform independent model. The approach includes the steps for successive model transformations that enrich progressively the model with platform information. The approach is supported by a tool called Archi-MDE. Gamatie et al. [3] represent the Graphical Array Specification for Parallel and Distributed Computing (GASPARD) framework for massively parallel embedded systems to support the optimization of the usage of hardware resources. GASPARD uses MARTE standard profile for modeling embedded systems at a high abstraction level. MARTE models are then refined and used to automatically generate code. Our approach can be considered an alternative approach to both GASPARD and Archi-MDE. The difference of our approach is the particular focus on optimization at the design level using architecture viewpoints.

Several hardware/software codesign approaches for embedded systems start from high level designs from which the system implementation is produced after some automatic or manual refinements. However, to the best of our knowledge no architecture viewpoints have been provided before for supporting the parallel computing engineer in mapping the parallel algorithm to parallel computing platform.

6 Conclusion

In this paper we have provided an architecture framework for supporting the mapping of parallel algorithms to parallel computing platforms. For this we have first defined the metamodel that includes the underlying concepts for defining the viewpoint. We have evaluated the viewpoints from various perspectives and illustrated it for the array increment algorithm. We were able to apply the viewpoint for the incrementing array algorithm. We have adopted the approach also for other parallel algorithms without any problem. Adopting the viewpoints enable the communication among the parallel computing stakeholders, the analysis of the design decisions and the implementation of the parallel computing algorithm. In our future work we will define the tool support for implementing the viewpoints and we will focus on depicting the design space of configuration alternatives and the selection of feasible alternatives with respect to the relevant high performance computing metrics.

References

1. Arkin, E., Tekinerdogan, B., Imre, K. 2013. Model-Driven Approach for Supporting the Mapping of Parallel Algorithms to Parallel Computing Platforms. *Proc. of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems.*
2. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford. Documenting Software Architectures: Views and Beyond. Second Edition. Addison-Wesley, 2010.
3. Gamatié, A., et. al. 2011. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Transactions on Embedded Computing Systems*, *10*(4), 1–36, 2011.
4. ISO/IEC 42010:2007] Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010), 2011.
5. Kogge, P. et al., *Exascale Computing Study: Technology Challenges in Achieving Exascale Systems*. DARPA. (2008)
6. Object Management Group (OMG). http://omg.org, accessed: 2013.
7. M. Palyart, D.Lugato, I.Ober, and J.M. Bruel. MDE4HPC: an approach for using model-driven engineering in high-performance computing. In Proc. of the 15th Int. Conf. on Integrating System and Software Modeling (SDL'11), Iulian Ober and Ileana Ober (Eds.). Springer-Verlag, 2011.
8. D. Talia. 2001. Models and Trends in Parallel Programming. *Parallel Algorithms and Applications* 16, no. 2: 145-180.
9. B. Tekinerdogan, E. Demirli. Evaluation Framework for Software Architecture Viewpoint Languages. in Proc. of Ninth International ACM Sigsoft Conference on the Quality of Software Architectures Conference (QoSA 2013), Vancouver, Canada, pp. 89-98, June 17-21, 2013.

Model-Driven Transformations for Mapping Parallel Algorithms on Parallel Computing Platforms

Ethem Arkin[1], Bedir Tekinerdogan[2]

[1]Aselsan MGEO, Ankara, Turkey
earkin@aselsan.com.tr
[2]Bilkent University, Dept. of Computer Engineering, Ankara, Turkey
bedir@cs.bilkent.edu.tr

Abstract. One of the important problems in parallel computing is the mapping of the parallel algorithm to the parallel computing platform. Hereby, for each parallel node the corresponding code for the parallel nodes must be implemented. For platforms with a limited number of processing nodes this can be done manually. However, in case the parallel computing platform consists of hundreds of thousands of processing nodes then the manual coding of the parallel algorithms becomes intractable and error-prone. Moreover, a change of the parallel computing platform requires considerable effort and time of coding. In this paper we present a model-driven approach for generating the code of selected parallel algorithms to be mapped on parallel computing platforms. We describe the required platform independent metamodel, and the model-to-model and the model-to-text transformation patterns. We illustrate our approach for the parallel matrix multiplication algorithm.

Keywords: Model Driven Software Development, Parallel Computing, High Performance Computing, Domain Specific Language, Tool Support.

1 Introduction

The famous Moore's law which states that the performance of the processing power doubles every eighteen months is coming to an end due to the physical limitations of a single processor [11]. To keep increasing the performance of the processing power the current trend is towards applying parallel computing on multiple nodes. Unlike serial computing in which instructions are executed serially, multiple processing elements are used to execute the program instructions in parallel. An important challenge in parallel computing is the mapping of the parallel algorithm to the parallel computing platform. The mapping of the algorithm requires the analysis of the algorithm, writing the code for the algorithm and deploying it on the nodes of the parallel computing parallel computing platform. This mapping can be done manually in case we are dealing with a limited number of processing nodes. However, the current trend shows the dramatic increase of the number of processing nodes for parallel computing platforms with now about hundreds of thousands of nodes providing petascale to exascale level processing power [8]. As a consequence mapping the parallel algorithm to computing platforms has become intractable for the human parallel computing engineer.

Once the mapping has been realized in due time the parallel computing platform might need to evolve or change completely. In that case the overall mapping process must be redone from the beginning requiring lots of time and effort.

In this paper we provide a model-driven approach for both the mapping of parallel algorithms to parallel computing platform, and the evolution of the parallel computing platform. In

essence our approach is based on the model-driven architecture design paradigm that makes a distinction between platform independent models and platforms specific models or code. We provide a platform independent metamodel for parallel computing platform and define the model-to-model transformation patterns for realizing the platform specific parallel computing platforms. Further we provide the model-to-text transformation patterns for realizing the code from the platform specific models.

The remainder of the paper is organized as follows. In section 2, we describe the problem statement. Section 3 presents the implementation approach for mapping the parallel algorithm to parallel computing platform by the help of model transformations. Section 4 presents the related work and finally we conclude the paper in section 5.

2 Problem Statement

To define a feasible mapping the parallel algorithm needs to be analyzed and a proper configuration of the given parallel computing platform is required to meet the corresponding quality requirements for power consumption, efficiency and memory usage. To illustrate the problem we will use the parallel matrix multiplication algorithm [10]. The pseudo code of the algorithm is shown inFig.1a. The matrix multiplication algorithm recursively decomposes the matrix into subdivisions and multiplies the smaller matrices to be summed up to find the resulting matrix. The algorithm is actually composed of three different sections. The first serial section is the multiplication of subdivision matrix elements (line 3), which is followed by a recursive multiplication call for each subdivision (line 5-15). The final part of the algorithm defines the summation of the multiplication results for each subdivision (line 13-16).

Given a physical parallel computing platform consisting of a set of nodes, we need to define the mapping of the different sections to the nodes. In this context, the logical configuration is a view of the physical configuration that defines the logical communication structure among the physical nodes. Typically, for the same physical configuration we can have many different logical configurations [2]. An example of a logical configuration is shown inFig.1b. In this paper we assume that a feasible logical configuration is selected and the mapping of the code need to be realized.

```
1.  Procedure Matrix-Multiply(A, B, s):
2.  if s=1 then
3.  C = A * B
4.  endif
5.  P0 = Matrix-Multiply(A00, B00, s-1)
6.  P1 = Matrix-Multiply(A01, B10, s-1)
7.  P2 = Matrix-Multiply(A00, B01, s-1)
8.  P3 = Matrix-Multiply(A01, B11, s-1)
9.  P4 = Matrix-Multiply(A10, B11, s-1)
10. P5 = Matrix-Multiply(A11, B10, s-1)
11. P6 = Matrix-Multiply(A10, B01, s-1)
12. P7 = Matrix-Multiply(A11, B11, s-1)
13. C00 = P0 + P1
14. C01 = P2 + P3
15. C10 = P4 + P5
16. C11 = P6 + P7
```

a) b)

Fig.1.Matrix Multiplication Algorithm (a) to be mapped on (b) logical configuration platform

Fig.2 shows an example of a manually written C code for the matrix multiplication algorithm. The code is implemented using the MPI [12], a widely used parallel programming library. For simplicity, we assume that a 2x2 physical configuration is selected. Hence, the example code is defined for a four node logical configuration. Before starting the code it is required to initialize the MPI configuration and related variables (line 3). For succinctness we have omitted the code in the figure. The algorithm will run in parallel on four nodes. To distinguish among the nodes the variable rank defines four different ids including 0, 1, 2, and 3. From line 4 to 8

the code for node 0 is defined which sends the sub matrices to the other nodes (1,2,3). Lines 9 to 14 define the code for receiving the matrices in node 1. A similar code is implemented for the nodes 2 and 3 (not shown in the figure). Line 16 defines a so-called barrier to let the process wait until all the sub-matrices have been distributed and received by all the nodes. After the distribution of the sub-matrices to the nodes, each node runs the code as defined in line 17-18 and, as such, multiplies, the received sub-matrices. Once the multiplication is finalized the results are submitted to node 0, which is shown in line 19-22 for node 1 (code for node 2 and 3 is not shown). Line 23 to 25 defines again the collection of the results in node 0. Line 27 defines again a barrier to complete this process. Finally in line 28 to 33 the results are summed in node 0 to compute the resulting matrix C.

```
1.  #include "mpi.h"
2.  int main
3.  {//MPI initializations
4.  if(rank == 0) {
5.  MPI_Isend(A_0_0, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
6.  MPI_Isend(B_0_0, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
7.  MPI_Isend(A_0_1, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);
8.  MPI_Isend(B_1_0, 4, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &request);}
9.  if(rank == 1) {
10. MPI_Irecv(A_0, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
11. MPI_Irecv(B_0, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
12. MPI_Irecv(A_1, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
13. MPI_Irecv(B_1, 4, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &request);}
14. ...
15. MPI_Barrier(MPI_COMM_WORLD);
16. //SERIAL SECTION PART (RUN ON ALL NODES)
17. C_0 = A_0 * B_0;
18. C_1 = A_1 * B_1;
19. if(rank == 1) {
20. MPI_Isend(C_0, 4, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, &request);
21. MPI_Isend(C_1, 4, MPI_DOUBLE, 0, 2, MPI_COMM_WORLD, &request);
22. }
23. if(rank == 0) {
24. MPI_Irecv(P_0, 4, MPI_DOUBLE, 2, MPI_ANY_TAG, MPI_COMM_WORLD, &request);
25. MPI_Irecv(P_1, 4, MPI_DOUBLE, 2, MPI_ANY_TAG, MPI_COMM_WORLD, &request);}
26. ...
27. MPI_Barrier(MPI_COMM_WORLD);
28. // SERIAL SECTION PART (RUN ON FIRST NODE)
29. if(rank == 0) {
30. C00 = P 0 + P 1
31. C01 = P_2 + P_3
32. C10 = P_4 + P_5
33. C11 = P_6 + P_7 }
34. MPI_Finalize();}
```

Fig.2.Example parallel code of the matrix multiplication algorithm code

After the code implementation, we can allocate and deploy the developed code to the nodes of the parallel computing platform. In our example we have assumed a simple configuration consisting of four nodes. Here we could easily decide on the strategy for sending, receiving and collecting the data over the nodes. However, one can imagine easily that the code for the larger configurations such as in petascale and exascale becomes dramatically larger, the strategy for the data distribution will be much more difficult [4] and likewise the effort to implement the code will be much higher. Because of the size and complexity implementing the code is not trivial and can become easily error-prone. In case of platform evolution or change the whole code needs to be substantially adapted or even rewritten from scratch.

3 Implementation Approach

To support the implementation and deployment of the code for the parallel computing algorithm on the parallel computing platform we propose a model-driven development approach. The approach integrates the conventional analysis of parallel computing algorithms with the

model-driven development approaches. The overall approach is shown inFig.3. In the first step of the approach the parallel computing algorithm is analyzed to define and characterize the sections that need to be allocated to the nodes of the parallel computing platform. In the second step, the plan is defined for allocating the algorithm sections to the corresponding nodes of the logical computing platform. In the third step the code for each serial section is manually implemented. The fourth step includes the implementation or reuse of predefined model transformations to generate the code for parallel sections. The final step includes the deployment of the code on the physical configuration platform. The details of the steps are described in the following sub-sections.

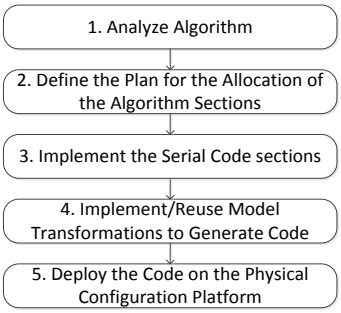


Fig.3.Approach for Generating/Developing and Deployment of Parallel Algorithm Code

3.1 Analyze Algorithm

The analysis of the parallel algorithm identifies the separate sections of the algorithm and characterizes these as serial or parallel sections. Here, a section is defined as a coherent set of instructions in the algorithm. A serial section defines the part of the algorithm that needs to run serially on nodes without interacting with other nodes. A parallel section defines the part of the algorithm that runs on each node and interacts with other nodes. For example the matrix multiplication algorithm (Fig.1a) has four main sections as shown in Table 1.

Table 1.Analysis of algorithm sections

NO	Algorithm Section	Section Type
1	Distribute the sub-matrices	PAR
2	C = A * B	SER
3	Collect matrix multiply results	PAR
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER

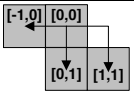
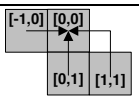
The first section defines the distribution of the sub-matrices to the different nodes. This section is characterized as a parallel section (PAR). The second section is characterized as serial (SER) and defines the set of instructions for the multiplication of the sub-matrices. The third section is a parallel section and defines the collection of the results of the matrix multiplications. Finally, the fourth section is characterized as serial and defines the summation of the result to derive the final matrix.

3.2 Define the Plan for the Allocation of the Algorithm Sections

The next step of the implementation approach is to define the plan for mapping the algorithm sections to logical configurations. Usually many different logical configurations can be derived for a given parallel algorithm and parallel computing platform. We refer to our earlier paper [2] in which we define the overall approach for deriving feasible logical configuration alternatives with respect to speed-up and efficiency metrics. In this paper we assume that a

feasible logical configuration has been selected and elaborate on the generation of the implementation of the algorithm sections.

Table 2.Plan for allocating sections to nodes

NO	Algorithm Section	Section Type	Plan
1	Distribute the sub-matrices	PAR	
2	C = A * B	SER	Run on each node
3	Collect matrix multiply results	PAR	
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	SER	Run on each node

The allocation of the sections to the nodes depends on the type of the sections. The plan for the matrix multiplication algorithm is shown in the fourth column of Table 2. Here we assume that each serial section runs on each node (section 2 and 4). The plan for allocating the parallel sections is defined as a pattern of nodes. The rectangles represent the nodes; the arrows represent the interactions (distribution or collection) among the nodes. Further, each node is assigned an id defining the coordinate of the node in the logical configuration. For section 1 the distribution of the data is represented as a pattern of four nodes in which the dominating node is the node with coordinate (0, 0). The arrows in the pattern show the distribution of the sub-matrices from the dominating node to the other nodes. For section 3 the pattern represents the collection of the results of the multiplications to provide the final matrix.

In the given example we have assumed a logical configuration consisting of four nodes. Of course for larger configurations defining the allocation plan becomes more difficult. Hereby, the required plan is not drawn completely but defined as a set of patterns that can be used to generate the actual logical configuration. For example, scaling the patterns of Table 2can be used to generate the logical configuration ofFig.1b. For more details about the generation of larger logical configurations from predefined patterns we refer to our earlier paper [2].

3.3 Implement the Serial Code Sections

Once the plan for allocating the algorithm sections to the logical configuration is defined we can start the implementation of the algorithm sections. Hereby, the code for the serial sections is implemented manually.

Table 3.Implementation of the serial sections

NO	Algorithm Section	Implementation
1	Distribute the sub-matrices	Will be generated
2	C = A * B	C0 = A_0 * B_0 C1 = A_1 * B_1
3	Collect matrix multiply results	Will be generated
4	C00 = P0 + P1 C01 = P2 + P3 C10 = P4 + P5 C11 = P6 + P7	C00 = P_0 + P_1 C01 = P_2 + P_3 C10 = P_4 + P_5 C11 = P_6 + P_7

The code for the parallel sections are generated using the model-transformation patterns as defined in the next sub-section. The third column of Table 3 shows the implementation of the serial sections of the matrix multiplication algorithm. Note that the implementation is alignment with the complete implementation of the algorithm as shown in Fig.2.

3.4 Model Transformations

After analyzing the algorithm, implementing the code for serial algorithm sections and defining the plan for mapping these sections to the logical configuration, the code for the parallel sections will be generated. To support platform independence this code generation process is realized in two steps using model-to-model transformation and model-to-text transformation. These transformation steps are described below.

Model-to-Model Transformation.

For different parallel computing platforms, there are several parallel programming languages such as, MPI, OpenMP, MPL, CILK [15]. According to the characteristic of the parallel computing platforms, different programming languages can be selected. Later on in case of changing requirements a different platform might need to be selected. To cope with the platform independence and the platform evolution problem we apply the concepts as defined in the Model-Driven Architecture (MDA) paradigm [13]. Accordingly, we make a distinction between platform independent models (PIM), platform specific models (PSM) and the source code. The generic model-to-model transformation process is shown in Fig.4.

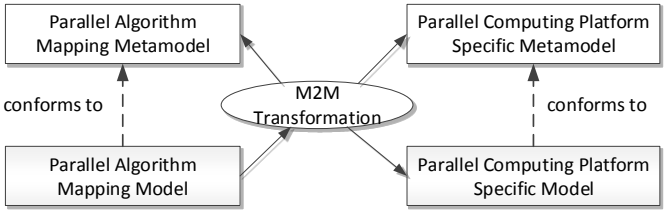


Fig.4.Model-to-model transformation.

Here the transformation process takes as input a platform independent model called, parallel algorithm mapping model. This model defines the mapping of the algorithm sections to the logical configuration. The model conforms to the parallel algorithm mapping metamodel which we will explain later in the section. The output of the transformation process is a platform specific model, called parallel computing platform specific model. Similarly this model conforms to its own metamodel, which typically represents the model of the language of the platform (e.g. MPI metamodel). The platform specific model will be later used to generate the code using model-to-text transformation patterns.

```
Algorithm: 'entity' name = ID '{'(sections+=Section*) '}';
Section:'abstract entity' name = ID '{'(sections+=Section*) '}';
SerialSection:
  'entity' name = ID ('extends' superType = [Section])? '{'code = STRING '}';
ParallelSection:
  'entity' name = ID ('extends' superType = [Section])? '{'tiles += Pattern*'}';
LogicalConfiguration:'entity' name = ID '{'(tiles+=Tile*)'}';
Tile:'abstract entity' name = ID '{''}';
Core:'entity' name = ID ('extends' superType = [Tile])?'{'i = INTj = INT '}';
Pattern:'entity' name = ID ('extends' superType = [Tile])?
'{' tiles += Tile*dominationgs += Tilecomms += Communication*
xsize = INTysize = INT'}';
Communication:'entity' name = ID '{'
    from = Coreto = Corelegsize = INTfromData = DatatoData = Data '}';
```

Fig.5.Concrete Syntax of the Parallel Algorithm Mapping Metamodel (PAMM)

The grammar for the parallel algorithm mapping metamodel is defined in *XText* in the Eclipse IDE and shown in Fig.5. Here, Algorithm consists of Sections, which can be either a *ParallelSection* or *SerialSection*. Each section can itself have other sections. In the grammar the serial sections are related to code implementations in the code block. The parallel sections include the data about the mapping plan that is determined with the logical configuration. *LogicalConfiguration* consists of *Tile* entity which can be either a single *Core* (processing

unit) or a *Pattern* with tiles and communications between these tiles. The assets related with the logical configuration with cores and patterns compose the plan for mapping algorithm to logical configuration.

Fig.6 shows, for example, the parallel algorithm mapping model for the matrix multiplication algorithm. In the figure two serial sections *MultiplyBlock* and *SumBlock* are defined. In the *MultiplyBlock* section the matrices are divided into sub-matrices and scattered by using the B2S pattern. The B2S pattern is a predefined pattern in the toolset indicating the pattern for section 1 as defined in the fourth column of Table 2. This multiply block also contains a Multiply serial section which contains the serial implementation of the multiply operation. In the *SumBlock* section, the resulting matrices are gathered by the pattern *B2G* which is predefined for section 3 as shown in the fourth column of Table 2. The *SumBlock* serial section contains the serial code for summation of the resulting sub-matrices.

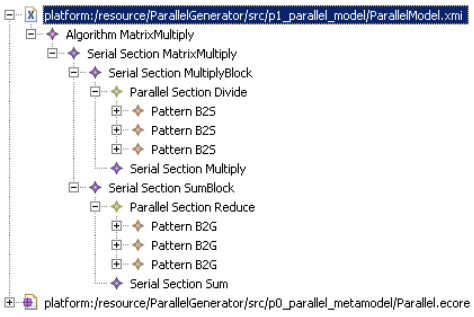


Fig.6.Parallel Algorithm Mapping Model for the Matrix Multiplication Algorithm

Once the platform independent parallel algorithm mapping model is defined we can transform it to the required platform specific model. We assume, for example, that the aim is to generate a MPI model. Fig.7shows the grammar of the MPI metamodel that is again defined using XText. In the metamodel each MPI model consists of a group of entities, which include MPISection, Process, Node, and Communication. Each section consists of processes and communication among these processes. Each Process allocates to a Node. Each communication defines the destination and target process.

```
MpiModel:'entity' name = ID '{'(groups+=MpiGroup*)'}';
MpiGroup:'entity' name = ID '{'(sections+=MpiSection*)(nodes+=Node)'}';
MpiSection:'entity' name = ID '{'
    (sections+=MpiSection*)(processes+=Process*)
    (communications+=Communication*)code = STRING'}';
Process:'entity' name = ID '{'rank = INTallocates=Node'}';
Node:'entity' name = ID '{''}';
Communication:'entity' name = ID '{'from = Processto = Process '}';
```

Fig.7.Grammar of the MPI Metamodel

The model-driven transformation rules refer to elements of both the PAMM and the parallel computing platform specific metamodel, in this case the MPI Metamodel. The M2M transformation rules are implemented using the ATL [1] transformation language. The transformation rules are shown in Fig.8. As shown in the figure we have implemented four different rules which define the transformations of mapping patterns to MPI sections, cores to processes and communications to MPI communications.

The rule *Algorithm2MpiModel*, is defined as the main rule of the transformation. The rule *Pattern2Section* transforms the algorithm pattern sections to *MpiSection* within the *MpiGroup*. The rule *Core2Process* transforms the cores as defined in the patterns to the processes in *MpiSection*. Each process is transformed from the core with the data of rank calculated from

the index values of the core. Similarly, *Comm2Comm* transforms the communications that are defined in the patterns, to the communications in *MPISection*.

```
1.  rule Algorithm2MpiModel {
2.    from algorithm: ParallelModel!Algorithmtompimodel: MpiModel!MpiModel (
3.  name<-algorithm.name, groups<-OrderedSet{mpiGroup}),
4.    mpiGroup: MpiModel!MpiGroup(name<-algorithm.name,
5.     sections<-algorithm.getPatterns())}
6.  rule Pattern2Section {
7.    from pattern: ParallelModel!Patternto section: MpiModel!MpiSection (
8.      name<-pattern.name,processes <- pattern.getCores(),
9.      communications<-pattern.getCommunications())}
10. rule Core2Process {from core: ParallelModel!Core to process: MpiModel!Process (
11.     rank<-core.i.mod(core.getGlobalSize())*core.getGlobalSize() +
12.           core.mod(core.getGlobalSize()),)}
13. rule Comm2Comm {from p_communication:ParallelModel!Communication
14. to communication : MpiModel!Communication (
15.     from<-p_communication.from, to<-p_communication.to,)}
```

Fig.8.Transformation rules from PAMM to MPI metamodel

The MPI model which is the result of the model-to-model transformation is shown in Fig.9. The MPI model includes the *MpiSection* with processes that will run on each node, communications from a destination process to target process and the serial code section implementation. This MPI model is now ready for model-to-text transformation to generate the final MPI source code.

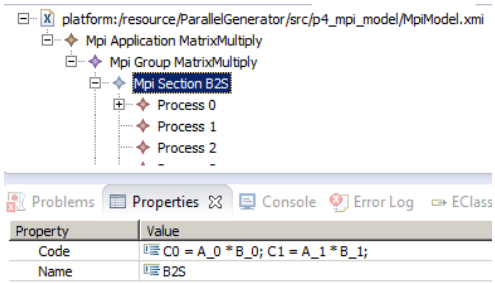


Fig.9.Part of the MPI model generated by model-to-model transformation

Model-to-Text Transformation

The generated PSM includes the mapping of the processes specific to the parallel computing platform. Subsequently, this PSM is used to generate the source code. The model-to-text transformation pattern for this is shown inFig.10.

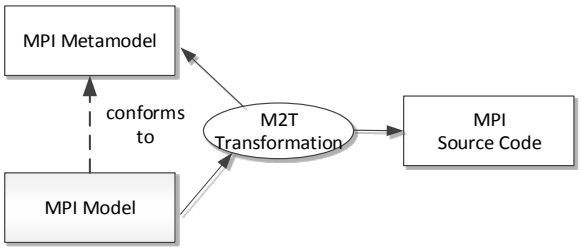


Fig.10.Example model transformation chain of MPI model

Fig.11 shows the implementation of the model-to-text transformation for which we used the XPand [18] transformation language. To map the sections to the parallel computing platform, for each section the communication operations for the data is generated for target and destination process ranks (line 6 to 11). Subsequently, the serial code implementation is imported to the source code in line 13. For each section, a barrier code is implemented to synchronize the section processes (line 14). The resulting code of the transformation is the code as defined in Fig.2.

```
1.  «IMPORT mpi»
2.  … // MPI initializations and type definitions
3.  «FOREACH groups AS group»
4.  «FOREACH group.sections AS section»
5.  «FOREACH section.communications AS comm»
6.  if(rank == «comm.from.rank») {
7.  MPI_Isend(«comm.fromData.name»,«comm.fromData.size»,MPI_«comm.fromData.type»,
8.            «comm.to.rank»,«comm.from.rank»,MPI_COMM_WORLD,&request);}
9.  if(rank == «comm.to.rank») {
10. MPI_Irecv(«comm.toData.name»,«comm.fromData.size»,MPI_«comm.toData.type»,
11.           «comm.from.rank»,MPI_ANY_TAG,MPI_COMM_WORLD,&request);}
12. «ENDFOREACH»
13. «section.code»
14. MPI_Barrier(MPI_COMM_WORLD);
15. «ENDFOREACH»«ENDFOREACH»
16. … // Final code
```

Fig.11. Transformation template from MPI metamodel to MPI source code

3.5 Deploy Code on Physical Configuration

The resulting code of the previous steps needs to be deployed on the physical configuration. The deployment can be done manually or using tool support in case of large configurations. In the literature various tools can be found which concern the automatic deployment of the code to the nodes of a parallel computing platform. We refer to, for example, [8][15][4] for further details.

4 Related Work

Several papers have been published in the domain of model-transformations for parallel computing. Palyart et. al. [14] propose an approach for using model-driven engineering in high performance computing. They focus on automated support for the design of a high performance computing application based on the distinction of different domain expertise like physical configuration, numerical computing, application architecture etc.

Bigot and Perez [3] adopt HLCM a hierarchical and generic component model with connectors originally designed for high performance applications. The authors represent on their experience with metamodeling and model transformation to implement HLCM. Gamatié et al. [7] introduced the GASPARD design framework systems that use model transformations for massively parallel embedded systems. They refined the MARTE models based on Model Driven Engineering paradigm. They provide tool support to automatically generate code with high-level specifications. Taillard et.al [16] implemented a graphical framework for integrating new metamodels to GASPARD framework. They used MDE paradigm to generate OpenMP, Fortran or C code.

Similar to our approach the above studies generate source code for high performance computing. The main difference of our approach is focus on the mapping of algorithm sections to parallel computing platforms.

5 Conclusion

In this paper we have described the model transformations needed to implement the mapping of a parallel algorithm to a parallel computing platform. In alignment with the MDA paradigm the approach is based on separating the platform independent parallel computing model from the platform specific parallel computing model and the source code. The model transformations do not only helps the parallel programming engineer to generate code but it also provides support for easier portability in case of platform evolution. We have illustrated the approach for the MPI platform but the approach is generic. In our future work we will elaborate on the application of model-driven approaches to parallel computing platform and focus on optimizing the values for metrics which are important for mapping parallel algorithms to parallel computing platforms.

References

1. ATL: ATL Transformation Language. http://www.eclipse.org/atl/
2. Arkin, E., Tekinerdogan, B., Imre, K. Model-Driven Approach for Supporting the Mapping of Parallel Algorithms to Parallel Computing Platforms. *Proc. of the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems*. (2013)
3. Bigot, J., Perez, C. On Model-Driven Engineering to implement a Component Assembly Compiler for High Performance Computing. JournéessurlIngénierieDirigée par les Modèles, IDM 2011. (2011)
4. Cumberland, D., Herban, R., Irvine, R., Shuey, M., and Luisier, M. Rapid parallel systems deployment: techniques for overnight clustering. In *Proceedings of the 22nd conference on Large installation system administration conference* (LISA'08). USENIX Association, Berkeley, CA, USA, 49-57. (2008)
5. Foster, I. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. *Addison-Wesley Longman Publishing Co.*, Inc., Boston, MA, USA. (1995)
6. Frank, M.P. The physical limits of computing. *Computing in Science &Engineering* , vol.4, no.3, pp.16,26, May-June 2002. (2002)
7. Gamatié, A., Le Beux, S., Piel, E., Ben Atitallah, R., Etien, A., Marquet, P., and Dekeyser, J. A Model-Driven Design Framework for Massively Parallel Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 10, 4, Article 39. (2011)
8. Hoffmann, A., Neubauer, B. Deployment and configuration of distributed systems. In *Proceedings of the 4th international SDL and MSC conference on System Analysis and Modeling* (SAM'04), Daniel Amyot and Alan W. Williams (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16. (2004)
9. Kogge, P., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R.S., Yelick, K., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Williams, R.S., and Yelick, K., *Exascale Computing Study: Technology Challenges in Achieving Exascale Systems*. DARPA. (2008)
10. Li, K.Scalable parallel matrix multiplication on distributed memory parallel computers. *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International* , vol., no., pp.307,314. (2000)
11. Moore, G.E.Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE* , vol.86, no.1, pp.82,85. (1998)
12. MPI: A Message-Passing Interface Standart, version 1.1. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.
13. Object Management Group (OMG). Model Driven Architecture (MDA), ormsc/2001-07-01.
14. Palyart, M., Lugato, D., Ober, I., and Bruel, J. MDE4HPC: an approach for using model-driven engineering in high-performance computing. In *Proceedings of the 15th international conference on Integrating System and Software Modeling* (SDL'11), Iulian Ober and Ileana Ober (Eds.). Springer-Verlag, Berlin, Heidelberg, 247-261. (2011)
15. Stawinska, M., Kurzyniec, D., Stawinski, J., Sunderam, V., Automated Deployment Support for Parallel Distributed Computing, *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on* , vol., no., pp.139,146. (2007)
16. Taillard, J., Guyomarc'h, F.,Dekeyser, J. A Graphical Framework for High Performance Computing Using An MDE Approach. In *Proc. of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing* (PDP '08). IEEE Computer Society, Washington, DC, USA, 165-173. (2008)
17. Talia, D. Models and Trends in Parallel Programming. *Parallel Algorithms and Applications* 16, no. 2: 145-180. (2001)
18. Xpand, Open Architectureware. http://wiki.eclipse.org/Xpand.
19. Zheng, G., Kakulapati, G., Kale, L.V. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. *Parallel and Distributed Processing Symposium, 2004. Proc.. 18th International* , vol., no., pp.78,, 26-30. (2004)