Domain Specific Language for Deployment of Parallel Applications on Parallel Computing Platforms

Ethem Arkin
Aselsan
MGEO Division
Etlik, 06011 Ankara Turkey
earkin@aselsan.com.tr

Bedir Tekinerdogan
Bilkent University
Computer Engineering Department
Bilkent, 06800 Ankara Turkey
bedir@cs.bilkent.edu.tr

ABSTRACT
To increase the computing performance the current trend is towards applying parallel computing in which parallel tasks are executed on multiple nodes. The deployment of tasks on the computing platform usually impacts the overall performance and as such needs to be modelled carefully. In the architecture design community the deployment viewpoint is an important viewpoint to support this mapping process. In general the derived deployment views are visual notations that are not amenable for run-time processing, and do not scale well for deployment of large scale parallel applications. In this paper we propose a domain specific language (DSL) for modeling the deployment of parallel applications and for providing automated support for the deployment process. The DSL is based on a metamodel that is derived after a domain analysis on parallel computing. We illustrate the application of the DSL for a traffic simulation system and provide a set of important scenarios for using the DSL.

Categories and Subject Descriptors
D.2.13 [**Software Engineering**].

General Terms
Design.

Keywords
Parallel Computing, Architecture Viewpoint, Domain Specific Language, Deployment, Software Language Engineering.

1. INTRODUCTION
The famous Moore's law states that the number of transistors on integrated circuits and likewise the performance of processors doubles approximately every eighteen months [25]. Since the introduction of the law in 1965, the law seems to have quite accurately described and predicted the developments of the processing power of components in the semiconductor industry [1]. Although Moore's law is still in effect, currently it is recognized that increasing the processing power of a single processor has reached the physical limitations [14]. Hence, to increase the performance the current trend is towards applying parallel computing on multiple nodes. Here, unlike serial computing in which instructions are executed serially, multiple processing elements are used to execute the program instructions simultaneously.

The deployment of tasks on the computing platform usually impacts the overall performance. Different deployment alternatives might, for example, impact the speedup and the efficiency of the parallel application [3]. To support the communication among stakeholders, to reason about the design decision, and support the analysis, the deployment needs to be modelled carefully. In fact, in the architecture design community the deployment viewpoint is an important viewpoint to support the mapping of applications on computing platforms. Here we can identify two important concerns. First of all the derived deployment views are usually visual notations that are basically targeted for human designers and as such the deployment needs to be done manually. Secondly, visual models are suitable for small to medium applications but soon they do not scale well for deployment of large scale parallel applications. This is important since 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 [21]. As a consequence the manual deployment of the parallel applications to computing platforms has become intractable.

In this paper we propose a domain specific language (DSL) for modeling the deployment of parallel applications and for providing automated support for the deployment process. The DSL is based on a metamodel that is derived after a domain analysis on parallel computing [12][22][15]. We illustrate the application of the DSL for a traffic simulation system and provide a set of important scenarios for using the DSL.

The remainder of the paper is organized as follows. In section 2, we describe the background. Section 3 presents the deployment viewpoint on which the DSL is based. Section 4 presents the deployment DSL. Section 5 presents the traffic simulation case study for illustrating the DSL. Section 6 presents a set of important scenarios for the automated support of the deployment process. Section 7 presents the related work, and finally section 8 presents the conclusions.

ECSAW, August 25 – 29, 2014, Vienna, Austria.

http://dx.doi.org/10.1145/2642803.2642819

2. PRELIMINARIES

In this section we discuss the background related to architecture modeling and deployment viewpoints, and present the requirements for the DSL for deployment in parallel computing.

2.1 Architecture Modeling

Historically, models have had a long tradition in software engineering and have been widely used in software projects. The primary reason for modeling is usually defined as a means for communication, analysis or guiding the production process. Models are different in nature and quality and different classifications of models have been provided in the literature. Mellor et al. [24] make a distinction between three kinds of models, depending on their level of precision. A model can be characterized as *Sketch*, *Blueprint*, or *Executable*. According to [24] an executable model is a model that has everything required to produce the desired functionality of a single domain. Executable models are more precise than sketches or blueprints, and can be interpreted by model compilers. A similar classification of models is defined by Fowler [13] who suggests a distinction based on three levels of models, namely *Conceptual Models*, *Specification Models* and *Implementation Models*. In model-driven software development the concept of models can be considered as executable models as defined by the above characterization of Mellor et al. [24]. In model-driven software development [5][21][24] models are not mere documentation but become "code" that is executable and that can be used to generate even more refined models or code. This is in contrast to model-based software development in which models are used as blueprints at the most. The language in which models are expressed can be defined by domain specific languages (DSLs).The application of a systematic, disciplined, quantifiable approach to the development, use, and maintenance of these languages is usually called software language engineering [21].

2.2 Deployment View

A common practice is to model and document different architectural views for describing the architecture according to the stakeholders' concerns. An architectural view is a representation of a set of system elements and relations associated with them to support a particular concern. Having multiple views helps to separate the concerns and as such support the modeling, understanding, communication and analysis of the software architecture for different stakeholders. 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. Different architectural frameworks have been proposed in the literature. Examples of the popular architectural frameworks include the Kruchten's 4+1 approach, the UML [13] 4+1 approach, the Siemens Four View Model [18], the Views and Beyond approach (V&B) [7] and the Architecture Perspectives approach of Rozanski and Woods [26]. All of these architecture frameworks include viewpoints for mapping software elements to nodes. In Kruchten's 4+1 the *physical view* is concerned with the topology of software components on the physical layer, as well as the physical connections between these components. In UML's 4+1 approach *deployment diagrams* depict a static view of the run-time configuration of processing nodes and the components that run on those nodes. The Siemens Four View Model describes the *execution architecture* which describes the mapping of functionality to physical resources and the runtime characteristics of the system. Rozanski and Woods define the *deployment viewpoint* and the *concurrency viewpoint* [26]. The deployment viewpoint addresses how to describe the environment into which the system will be deployed including the dependencies the system has with its runtime environment. The concurrency viewpoint describes the concurrency structure of the system, mapping functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently.

The purpose of deployment diagrams include in general the visualization of hardware topology of a system, describing the hardware components used to deploy software components, and describing the runtime processing nodes. Typically, deployment diagrams are used by the system engineers. The notation for deployment diagrams are also largely similar. In UML, for example, the nodes appear as boxes, and the software elements artifacts allocated to each node appear as rectangles within the boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers.

3. DEPLOYMENT VIEWPOINT FOR PARALLEL COMPUTING

To represent the different concerns of deployments in parallel computing we could consider adopting the existing general-purpose viewpoints. Unfortunately, these general-purpose frameworks fall short for expressing the particular concerns for parallel computing since these provide merely visual notations that are not executable. Accordingly, we have decided to define a viewpoint, and based on this a DSL, which can express parallel computing concerns explicitly.

Each DSL addresses specific concerns of a particular domain. For deriving the important concerns for mapping parallel tasks to parallel computing platforms we have carried out a domain analysis process. Domain analysis can be defined as the process of identifying, capturing and organizing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems [2]. Conventional domain analysis methods consist generally of the activities *Domain Scoping* and *Domain Modeling*: *Domain Scoping* identifies the domains of interest, the stakeholders, and their goals, and defines the scope of the domain. *Domain Modeling* is the activity for representing the domain, or the *domain model*. The domain model can be represented in different forms such as object-oriented language, algebraic specifications, rules, conceptual models or a DSL. Typically a domain model is formed through a commonality and variability analysis to concepts in the domain. The domain scope for our purposes included the literature on parallel computing in general, such as for example [12][22]. Based on commonality analysis of the selected studies we derived the following important concerns for mapping parallel tasks to parallel computing platforms:

- *Modeling of the physical computing platform*

The application will run on a selected or to be selected physical configuration platform that consists of multiple nodes. It is important to model the physical computing platform for smaller but also for very large computing platforms (e.g. exascale computing).

- *Modeling parallel and serial modules in the application*

Depending on the application semantics, while some modules can run in parallel others can only run in serial. Typically serial modules will be mapped to a single node, while parallel modules need to be mapped to multiple nodes. For the architect it is important to describe these explicitly and as such help to identify the proper selection of parallel module.

- *Modeling the mapping of parallel modules to physical nodes*

The allocation of the application to the computing platform can be carried out in different ways resulting in different performance. To reason about the mapping this should be explicitly represented.

- *Defining the interaction patterns among parallel modules*

Parallel modules typically exchange information to perform the requested tasks. In general it is important to define the proper interaction patterns not only for functional reasons but also to optimize the parallelization overhead and as such increase efficiency.

A further result of the domain analysis process is the metamodel for the deployment of the application to a physical configuration, which is shown in Figure 1. The metamodel has been defined based on the above identified concerns, and by analyzing existing deployment meta models such as in UML. The UML [32] includes a deployment metamodel which specifies a set of constructs that can be used to define the execution architecture of systems that represent the assignment of software artifacts to nodes. In the UML metamodel an artifact represents a concrete element in the physical world that is the result of a development process. Examples of artifacts are model files, source files, binary executable files etc. Artifacts can have different properties and can be deployed to various Node instances. Nodes can be devices or execution environments. Nodes are connected through communication paths to create network systems of arbitrary complexity. Nodes are typically defined in a nested manner, and represent either hardware devices or software execution environments.

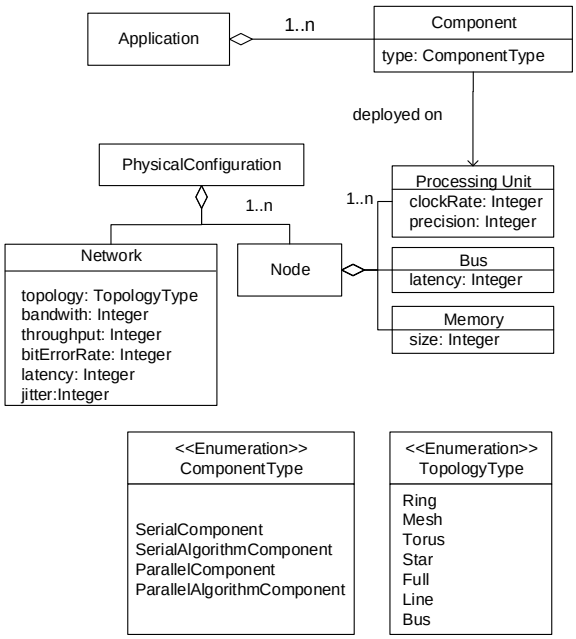
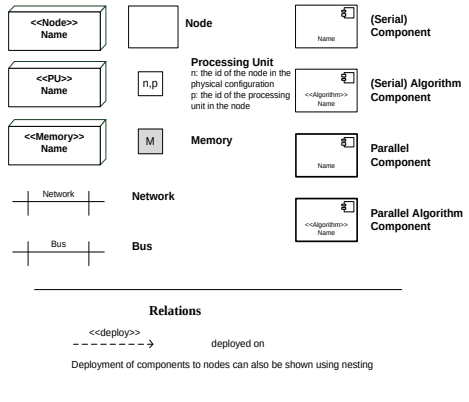


Figure 1. Deployment Metamodel

The metamodel of Figure 1 has been in the Epsilon environment [11] and aims to explicitly support the four concerns related to the deployment of parallel applications to parallel computing platforms. As defined in the metamodel Application consists of components which include an attribute type that defines whether the component is a serial component, serial algorithm component, parallel component, or parallel algorithm component. A serial component implements a serial activity performed on a single core that does not need parallelism. A parallel component implements a parallel activity that can be executed on multiple processing units. Parallel algorithm consists of sections that can be further decomposed into serial and parallel sections. Physical Configuration includes the Nodes and the Network among nodes of the computing platform. Network has attribute topology that represents the physical architecture topology and performance attributes like bandwidth, throughput, and latency. Node includes Processing Units, Memory and Bus. Components are deployed on the processing units of physical configuration. Processing Unit has clock rate and precision attributes. Bus has attribute latency for delivering data from memory to processing units. Memory has attribute size for storage size.

Based on the metamodel of Figure 1, we have derived the viewpoint as shown in Table 1. The first column of the table defines the template of the viewpoint, which is based on the template to document viewpoints as described by the ISO/IEC 42010:2007 recommended standard for architectural description [19]. The <<viewpoint name>> describes the selected name of the viewpoint. The field <<overview>> describes a brief description of the viewpoint and its key features. The field <<concerns>> describes a listing of the architecture related concerns framed by the viewpoint. The field <<Typical stakeholders>> describes the stakeholders for the viewpoint which are in this case system engineers. The field <<Constraints>> defines constraints for composing elements in the view. Finally, the field <<Model types and notations>> describes the adopted notations for the architectural elements.

Table 1. Deployment Viewpoint

Section	**Description**
«Viewpoint Name»	Deployment Viewpoint
«Overview»	The deployment for the components of the parallel application
«Concerns»	Which component runs on which processing unit?
«Typical Stakeholders»	System Engineer
«Constraints »	Parallel component can be deployed on different processing units. Serial component can be deployed on a single processing unit.
«Model types and notation»	

4. DSL FOR DEPLOYMENT VIEWPOINT

Based on the metamodel and the viewpoint we can now define the DSL. In principle we could describe one DSL that implements all the required concerns for deployment. However, we have decided to define two separate DSLs including a DSL for describing the physical configuration and a DSL for describing the components and the deployment of these components on the nodes. In this way

both concerns can be described separately. On the other hand the relation between both is ensured by import relations. That is, once the physical configuration is described, this can be imported in the description for the components and the deployment. A further advantage of this approach is that we can have the same physical configuration with different deployment descriptions. In this way different alternatives can be described and analyzed to meet the required quality requirements of the deployment. The DSLs has been designed to cover different kinds of application and platforms. Different applications can be modeled by adopting different types of components. The DSL for the platform can address small to large scale (e.g. exascale), and both homogenous and heterogeneous platforms.

```
1. grammar PhysicalConfiguration
2. Model:
3.  types+=TypeDef*
4.  physicalConfigurations+=PhysicalConfiguration*;
5. PhysicalConfiguration:
6.  'physicalconfiguration' name=ID '{'
7.    'nodes' ':' nodes+=[NodeType] '['size=INT']'
    (',' nodes+=[NodeType] '['size=INT']')* ';'
8.    'network' ':' network=[NetworkType] ';'
9.  '}';
10. TypeDef:
11.  NodeType | MemoryType | BusType |
12.  ProcessingUnitType | NetworkType;
13. NetworkType:
14. 'network' name=ID '{'
15.   'topology'':' topology=TopologyType ';'
16.   'bandwith'':' bandwith=INT bunit=BwithUnit';'
17.   'throughput'':' throughput=INT ';'
18.   'bitErrorRate'':' berate=INT '%' ';'
19.   'latency'':' latency=INT 'usec' ';'
20.   'jitter'':' jitter=INT ';'
21. '}';
22. enum TopologyType:
23.  Ring='Ring'|Mesh='Mesh'|Torus='Torus'|
    Star='Star'|
24.  Full='Full'|Line='Line'| Bus='Bus';
25. NodeType:
26. 'node' name=ID '{'
27.   'processingunits'':' pus+=[ProcessingUnitType] '['size=INT']' (',' pus+=[ProcessingUnitType] '['size=INT']')* ';'
28.   'memory' ':' memory=[MemoryType] ';'
29.   'bus' ':' bus=[BusType] ';'
30. '}';
31. ProcessingUnitType:
32. 'processingunit' name=ID '{'
33.   'clockRate'':' rate=DOUBLE unit=ClockRateType
    ';'
34.   'precision'':' precision=PrecisionType ';'
35. '}';
36. MemoryType:
37. 'memory' name=ID '{'
38.   'size' ':' size+=MemorySize ';'
39. '}';
40. MemorySize: size=INT unit=MemorySizeUnit;
41. DOUBLE returns EDouble: '-'? INT? '.' INT;
```

Figure 2. Physical Configuration DSL Grammar

The grammar of the Physical Configuration DSL is shown in Figure 2. The DSL is described using xText [34]. The DSL model includes a type definition section and physical configuration definition section. The type definitions are *NodeType*, *MemoryType*, *BusType*, *ProcessingUnitType* and *NetworkType*. A NodeType definition consists of Processing Units, Bus and Memory which are using ProcessingUnitType, BusType and MemoryType definitions in order. These type definitions include the attribute definitions that are defined in the metamodel as shown in Figure 1. ProcessingUnitType has clock rate and precision, BusType has latency, and MemoryType has size attributes. The NetworkType is also defined based on the metamodel which has topology, bandwidth, throughput, bit error rate, latency and jitter attributes.

The grammar for the second part of the DSL, the Deployment DSL, is shown in Figure 3. The grammar actually builds further on the grammar for the physical configuration. This is realized by importing the previous grammar in line 2. The DSL further includes the application definition and components which can be either a SerialComponent, a SerialAlgorithmComponent, a ParallelComponent or a ParallelAlgorithmComponent. The Component has a 'deployed on' attribute for a list of processing units which will be deployed on. The deployment can be either described by enumerating the components and nodes, or it can be succinctly described by using cardinality.

```
1. grammar Deployment
2. import "/dsl/PhysicalConfiguration" as pc
3. Model: applications+=Application*;
4. Application:
5.  'application' name=ID '{'
6.    components+=Component*
7.  '}';
8. Component:
9.  'component' name=ID '{'
10.   'type'':' type=ComponentType ';'
11.   'deployed on'':' pus+=DeploymentUnit
12.                   (',' pus+=DeploymentUnit)*';'
13. '}';
14. DeploymentUnit:
15. physicalconfiguration+=
    [pc::PhysicalConfiguration]('.' 'nodes'
    '['(index=INT | fromIndex=INT '..'
    toIndex=INT)(','(index=INT | fromIndex=INT '..'
    toIndex=INT))*']');
16. enum ComponentType:
17.  serial='SerialComponent'|
18.  parallel='ParallelComponent' |
19.  serialalg='SerialAlgorithmComponent' |
20.  parallelalg='ParallelAlgorithmComponent';
```

Figure 3. Deployment DSL Grammar

5. CASE STUDY

To illustrate the DSL we use case study for the development of a traffic simulation. The goal of this simulation is to support the analysis and optimization of various traffic flow parameters for efficient movement of traffic and minimal traffic congestion problems. Typically, a traffic simulation consists of a large set of components that need to be deployed on a parallel computing platform. The main components of the simulation environment are cars, trucks, drivers, speed cameras, traffic lights, lane closes and a traffic analyzer. Other components such as crossings, pedestrians, fixed/mobile radars, on-ramps and weather conditions that affect the traffic flow are not included in the case study for the sake of simplicity. The defined simulation system case study includes cars and trucks as vehicles. A vehicle component includes properties such as model year, motor power, current driver id, etc. Drivers have different physical and behavioral properties that affect the traffic flow. A driver component shall include properties such as driver id, sociodemographic factors (age, gender, driving experience in years, etc), driving style (dissociative, anxious, risky, angry, high-velocity, distress-reduction, patient, and careful), and accident experience that indicates how many accidents the driver has been involved in. Speed cameras, traffic lights, and lane closes are participants that generally slow down the traffic flow. A speed camera component shall define position and speed limit value parameters. A traffic light component shall define position and light state (red, yellow, or green). A lane close component defines a start position, an end

position, the time slice that the lane is closed and a lane index that indicates closed lane (like 1st lane, 2nd lane). Traffic analyzer is a passive participant that collects simulation data from other participants such as vehicles and drivers to perform analysis.

For the traffic simulation case study, first we need to define the physical computing platform. The computing platform conforms to the number of simulations on the simulation scenario. Here, we define a sample Traffic Simulation Scenario shown in Table 2. The 'Simulation Component' column of the table indicates the simulation participants that together form the simulation of the system. The column 'Number' defines the number of simulation participants of the simulation module type in the given scenario. For example, in the scenario defined in Table 2, there are 600 cars and 80 trucks. As it can be observed for the given scenario, the total number of required simulation modules might be quite large. For the given scenario in Table 2 the total number of simulation modules is 1385.

Table 2. Traffic Simulation Scenario

Simulation Component	Number
Car Simulation	600
Truck Simulation	80
Driver Simulation	680
Speed Camera Simulation	5
Traffic Light Simulation	15
Lane Close Simulation	4
Traffic Analyzer Simulation	1

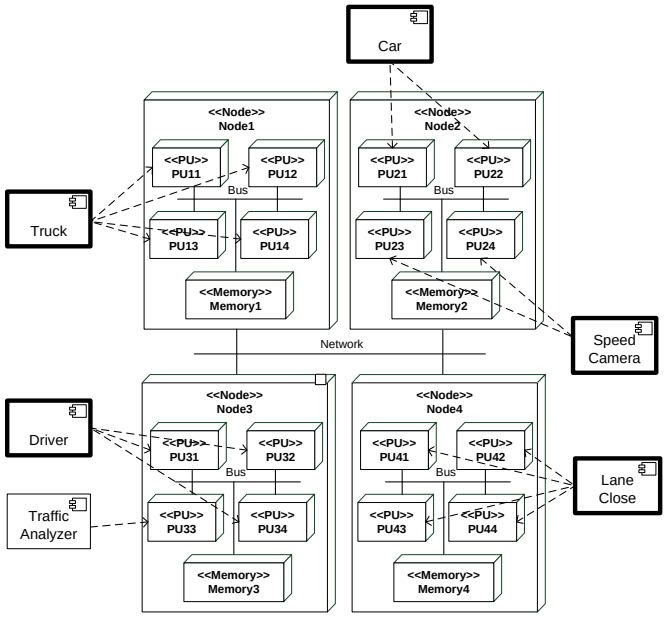


Figure 4. Part of the Deployment View for Traffic Simulation

```
1.  memory DDR2Memory {
2.   size : 4 GByte;
3.  }
4.  processingunit PowerPC450 {
5.   clockRate : 850.00 MHz;
6.   precision : 32Bit;
7.  }
8.  bus PowerPcBus {
9.   latency : 300 usec;
10. }
11. node TrafficSimulatorHost {
12.  processingunits : PowerPC450[4];
13.  memory : DDR2Memory;
14.  bus : PowerPcBus;
15. }
16. network TrafficSimulatorNet {
17.  topology : Torus;
18.  bandwith : 1 GBit;
19.  throughput : 10;
20.  bitErrorRate : 20%;
21.  latency : 500 usec;
22.  jitter : 10;
23. }
24. physicalconfiguration TrafficSimulatorComputer {
25.  nodes : TrafficSimulatorHost[350];
26.  network : TrafficSimulatorNet;
27. }
```

Figure 5. Traffic Simulator Computer Physical Configuration

Part of the deployment view is shown in Figure 4. Here, the Physical Configuration of the Simulator Computer is constructed using 4 nodes. Each node includes 4 processing units and a memory. Nodes are connected using a network. On this physical configuration, the components of the Traffic Simulator application are deployed on processing units using deployment relations. The illustration of the example is a small part of the deployment diagram, for larger applications such as traffic simulation example, a much larger view will be needed. Sooner or later the visual representation becomes less feasible.

For the scenario given in Table 2 and assuming that all the components need to be deployed on a distinct processing unit, then at least 1385 processing units will be needed. Figure 5 shows an example Traffic Simulator Computer physical configuration. The Traffic Simulator Computer consists of 350 Traffic Simulator Host nodes and a Traffic Simulator Network. Each node includes 4 PowerPC 450 processing units, so the physical configuration includes 1400 PowerPC 450 processing units. The PowerPC 450 processing unit is a 850 MHz core with 32 Bit precision. Each node has a 4 GB DDR2 memory and a bus with 300 usec latency. The network is constructed as Torus topology with 1 GBit bandwith, 20% bit error rate and etc.

To define the deployment of the Traffic Simulator application on Traffic Simulator Computer, the components of the application are defined including 'deployed on' relation. The deployment relation conforms to the processing units of the physical configuration. Figure 6 shows the deployment of the traffic simulator application for the given scenario. Each component is defined with the deployment relation that are either a number of nodes or processing units. For example Car components are deployed on node 1 to 150, which means they are deployed on 600 (4 x 150) processing units.

```
1. application TrafficSimulator {
2.  component Car {
3.    type : ParallelComponent;
4.    deployed on :
    TrafficSimulatorComputer.nodes[1..150];
5.  }
6.  component Truck {
7.    type : ParallelComponent;
8.    deployed on :
    TrafficSimulatorComputer.nodes[151..170];
9.  }
10. component Driver {
11.   type : ParallelComponent;
12.   deployed on :
    TrafficSimulatorComputer.nodes[171..340];
13. }
14. component SpeedCameraSim {
15.   type : ParallelComponent;
16.   deployed on :
    TrafficSimulatorComputer.nodes[341],
    TrafficSimulatorComputer.nodes[342].processingu
    nits[1];
17. }
18. component TraficLightSim {
19.   type : ParallelComponent;
20.   deployed on :
    TrafficSimulatorComputer.nodes[342].processingu
    nits[2..4],
    TrafficSimulatorComputer.nodes[343..345];
21. }
22. component LaneCloseSim {
23.   type : ParallelComponent;
24.   deployed on :
    TrafficSimulatorComputer.nodes[346];
25. }
26. component TrafficAnalyzer {
27.   type : SerialComponent;
28.   deployed on :
    TrafficSimulatorComputer.nodes[347].processingu
    nits[1];
29. }
30.}
```

Figure 6. Traffic Simulator Computer Deployment

6. IMPORTANT SCENARIOS

The DSL for deployment view provides an expressive approach for modeling large scale parallel applications. In addition to the declarative specification the DSL is useful for supporting several important scenarios. In the following, we list several of the most important scenarios, which can be addressed using the described DSL:

- *Analysis of Deployment Alternatives*

Using the DSL the deployment alternatives can be specified for a given physical configuration. The physical configuration components have attributes that affect the performance evaluation of the parallel application. For instance, the latency of the network increases the messaging overhead of the parallel application which will increase the total elapsed time of the application processes. Thus, each specification could be analyzed with respect to the defined functional and non-functional requirements. Using the DSL, different specifications can be defined that describe different alternatives.

Figure 7 shows the typical process, represented as a data flow diagram, for the analysis of deployment alternatives to find the feasible deployment solution. Based on the functional and non-functional requirements and using the DSL, different deployment specifications will be described. These specifications can then be automatically analyzed for selecting the feasible alternative. The analysis of the deployment alternative can be done using performance prediction methods like analytical, statistical or heuristic methods. The feasible deployment can be selected based on the performance prediction of the analysis using the calculated metrics for the deployments.

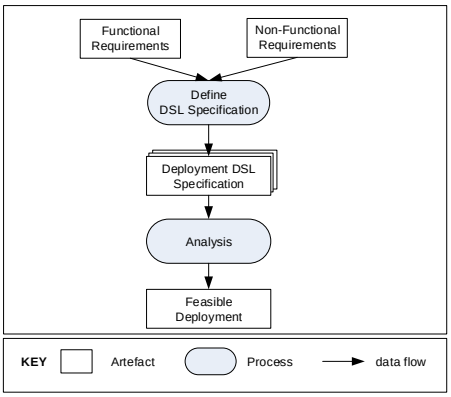


Figure 7. General process for analysis of deployment alternatives using DSL specifications

- *Automated Deployment of Application Code to Computing Platform*

The application code needs to be deployed on the computing platform to be run in parallel. This can be done manually, but for very large scale parallel applications this will be time consuming and cumbersome. 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][17][30] for further details. These tools mainly use installation specifications or configurations that define how the deployment framework will distribute and run the tasks among the processing units. To support reuse and integrate the design and analysis activities with the eventual installation of the code, the DSL specification of the feasible deployment alternative can be transformed to the format required by the existing tools. Figure 8 depicts the general process for this purpose. Based on the feasible deployment DSL specification the installation specification will be generated. Given the installation case, which we assume is automatically generated as well, the application can then be deployed automatically. In particular for large scale parallel computing platform this overall process will pay off and increase productivity and quality.

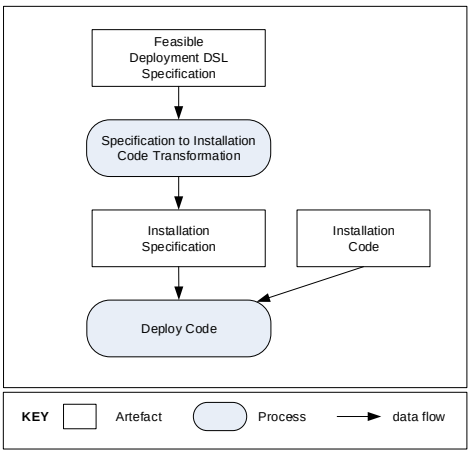


Figure 8. General process for analysis of deployment alternatives using DSL specifications

- *Documentation of Deployment Architecture*

Every architecture needs a documentation to guide architecture stakeholders about how to benefit from the architecture and clarify

ambiguous points. Architecture documentation is a communication artifact for all stakeholders and it is used during the whole lifecycle of the architecture. It contains both natural language descriptions about system and formal architecture models. We utilize our DSLs in order to automatically generate the architecture view related part of the architecture documentation. The generation is done via model-to-text transformation.

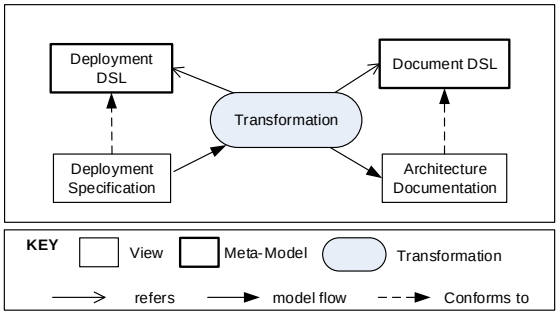


Figure 9. General process for generation of documentation of the deployment specification

7. RELATED WORK

In our earlier work we have provided a software language engineering approach to define viewpoints as domain specific languages [10]. This enhances the formal precision of architectural viewpoints and leads to executable views that can be interpreted and analyzed by tools. We have illustrated the approach for defining and evaluating domain specific languages for the viewpoints of the Views and Beyond approach [10][31]. In this paper we have used the same approach to define the DSL for mapping applications to parallel computing platforms.

To model of the deployment of an application has been addressed in several studies in the literature and specific languages are introduced for modeling architectures [23]. Architecture description languages (ADLs) have been proposed to model the components of architectures. The parts of the components are: interface, implementation and deployment. Different attempts have been made to provide clear guidelines for characterizing and distinguishing ADLs, by providing comparison and evaluation frameworks. None of the ADLs that we have studied has explicitly focused on the problem of modelling parallel applications on parallel computing platforms.

xADL has been introduced to support modularity and extensibility of architectural modeling [9][20]. Despite earlier ADLs xADL is not a single fixed ADL but encapsulates various ADL features in modules that can be composed to form new ADLs. This is achieved by using the extension mechanisms provided by XML and XML schemas. In our approach the presented DSL is actually an ADL but that is focused on parallel computing for large scale computation. It would be interesting to use and enhance xADL to model the mapping of parallel applications to parallel computing platforms. We consider this complementary alternative to the approach that we have presented.

Architecture Analysis and Design Language (AADL) defines a language for describing both the software architecture and the execution platform architectures of performance-critical, embedded, real-time systems. Similar to our DSL, AADL represents the physical hardware and the application software and can model the deployment of the application. Furthermore, OSATE toolkit can analyze the deployment alternatives and find feasible architecture designs which is an important scenario that we mentioned.

Several toolsets have been developed for supporting the deployment of parallel, distributed or cloud applications. Brandtzaeg et. al. [6] propose to model cloud applications using component-based approach. They define a high-level DSL including deployment descriptors. Sledziewski et. al. [28] also propose a DSL-based approach for deploying cloud applications. They aim to create a seamless environment which hides deployment details and enable application designers to focus mainly on the problem domain. Van Nieuwpoort et. al. [33] introduce Grid Application Toolkit (GAT) for deploying parallel applications on a large-scale grid system. They defined an API for grid applications to support the automatic deployment on grid systems. Hoefler and Snir [16] present a library named *libtopomap* to map parallel applications to large-scale parallel architectures using generic topology mapping strategies. They also propose an API to include deployment issues within parallel applications. Boujbel et. al. [4] present a DSL for multi-scale distributed applications for autonomic deployment. They focus on autonomic deployment strategies rather than building and running a deployment plan. For this problem domain, they express the importance of a DSL to define the scalability of the applications. Sabharwal [27] proposes a solution for grid infrastructure deployment across multiple heterogeneous distributed machines in parallel using SmartFrog (Smart Framework for Object Groups) [29]. The framework helps in abstracting the configuration for grid enabling process and its runtime environment automatically triggers installations across distributed machines. The baove studies also show that deployment is an important concern for addressing both functional and nonfunctional concerns. We consider these studies complementary to our approaches. Our distinguishing characteristic with respect to these studies is that we apply DSL to the deployment of parallel applications to parallel computing platforms. To the best of our knowledge, this has not yet been addressed.

8. CONCLUSION

In this paper we have proposed a domain specific language (DSL) for modeling the deployment of applications on parallel computing platforms. The DSL has been derived after having studied existing deployment viewpoints as well as the literature on parallel computing platforms. The DSL as such is expressive and provides a declarative and scalable approach for representing different deployment views in parallel computing. Using the DSL, the deployment of applications to even very large computing platforms such as exascale computing can be conveniently modeled. In addition, the executable DSL specifications can be used to support the automated analysis, automated generation of deployment specification, and automated generation of deployment architecture documentation. In our future work we aim to elaborate on these different important scenarios for supporting generative parallel computing.

REFERENCES

[1] Aizcorbe, A.M., and Kortum, S.S. 2005. Moore's Law and the Semiconductor Industry: A Vintage Model. Scandinavian Journal of Economics, Vol. 107, No. 4, pp. 603-630.

[2] Arrango, G. 1994. Domain Analysis Methods. In Software Reusability, Schäfer, R. Prieto-Díaz, and M. Matsumoto (Eds.), Ellis Horwood, New York, New York, pp. 17-49.

[3] Arkin, E., Tekinerdogan, B., and 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.*

[4] Boujbel,R. Leriche,S., Arcangeli, JP. 2013. A DSL for Multi-Scale and Autonomic Software Deployment". In: Int. Conf. on Software Engineering Advances, pp. 291–296

[5] Bézivin, J. 2005. On the Unification Power of Models. Software and System Modeling (SoSym) 4(2):171-188.

[6] Brandtzæg, E., Parastoo, M., Mosser, S. Towards a Domain-Specific Language to Deploy Applications in the Clouds. 2012. In Cloud computing 2012: 3rd int. conf. on cloud computing, grids, and virtualization, pages 213-218. IARIA.

[7] Clements, P. Bachmann, F., Bass, L., Garlan, D. Ivers, J. Little, R., Merson, P., Nord, R., Stafford, J. 2010. Documenting Software Architectures: Views and Beyond. Second Edition. Addison-Wesley.

[8] Cumberland, D. et. al. 2008. 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.

[9] Dashofy, E. M., van der Hoek, A. Taylor, R.N. 2005. A comprehensive approach for the development of modular software architecture description languages. ACM Transactions on Software Engineering and Methodology (TOSEM), v.14 n.2, p.199-245, April 2005

[10] Demirli, E., Tekinerdogan, B. 2011. Software Language Engineering of Architectural Viewpoints. in Proc. of the 5th European Conference on Software Architecture (ECSA 2011), LNCS 6903, pp. 336–343, 2011.

[11] Epsilon, http://www.eclipse.org/epsilon.

[12] Foster, I. 1995. Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering. *Addison-Wesley Longman Pub. Co.*, Inc., Boston, MA, USA.

[13] Fowler, M. Scott, S. Booch, G. 1999. UML distilled, Object Oriented series, 179 p. Addison-Wesley, Reading.

[14] Frank, M.P., 2002. The physical limits of computing. *Computing in Science & Engineering*, vol.4, no.3, pp.16,26.

[15] Grama, A., Gupta, A., Karypis, A., and Kumar, P. 2003. *Introduction to Parallel Computing, Second Edition*: Addison Wesley.

[16] Hoefler, T., & Snir, M. 2011. Generic topology mapping strategies for large-scale parallel architectures. *Proceedings of the International Conf. on Supercomputing - ICS '11*, 75.

[17] Hoffmann, A., Neubauer, B. 2004. Deployment and configuration of distributed systems. In Proceedings of the 4th international SDL and MSC Conf. on System Analysis and Modeling (SAM'04), Daniel Amyot and Alan W. Williams (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16.

[18] Hofmeister, C., Nord, R., Soni, D. Applied Software Architecture. Addison-Wesley, NJ, USA.

[19] ISO/IEC 42010:2007] Recommended practice for architectural description of software-intensive systems (ISO/IEC 42010), 2011.

[20] ISR, ArchStudio 4.0 tool set for the xADL language, http://www.isr.uci.edu/projects/archstudio/

[21] Kleppe, A. 2009. Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Longman Publishing Co., Inc., Boston.

[22] 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., 2008. *Exascale Computing Study: Technology Challenges in Achieving Exascale Systems*. DARPA.

[23] Medvidovic, N., Taylor, R. N. 2000. A classification and comparison framework for software architecture description languages, IEEE Trans. Software Eng., vol. 26, no. 1, pp. 70–93.

[24] Mellor, S.J. Scott, K. Uhl, A., Weise, D. 2004. MDA Distilled: Principle of Model Driven Architecture, Addison Wesley, Reading.

[25] Moore, G.E., 1998. Cramming More Components Onto Integrated Circuits. Proc. of the IEEE , vol.86, no.1, pp.82,85.

[26] Rozanski, N., Woods, E., 2005. Software Architecture Systems Working with Stakeholders Using Viewpoints and Perspectives, First Edition, Addison-Wesley.

[27] Sabharwal, R. 2006. Grid infrastructure deployment using SmartFrog technology. In *International Conference on Networking and Services 2006, ICNS'06*.

[28] Sledziewski, K., Bordbar, B., & Anane, R. 2010. A DSL-Based Approach to Software Development and Deployment on Cloud. 24th IEEE International Conference on Advanced Information Networking and Applications (AINA).

[29] SmartFrog. http://www.smartfrog.org.

[30] Stawinska, M., Kurzyniec, D., Stawinski, J., and Sunderam, V. 2007. Automated Deployment Support for Parallel Distributed Computing, *15th EUROMICRO Int. Conf. on Parallel, Distributed and Network-Based Processing. PDP '07.* vol., no., pp.139,146.

[31] Tekinerdogan, B., Demirli. E. 2013. Evaluation Framework for Software Architecture Viewpoint Languages. in Proc. of Ninth International ACM Sigsoft Conference on the Quality of Software Architectures Conference (QoSA 2013), pp. 89-98, Vancouver, Canada, June 17-21.

[32] UML Superstructure Spec. Unified Modeling Language (UML), V2.4.1, http://www.omg.org/spec/UML/2.4.1/.

[33] Nieuwpoort, R. V. van., Maassen, J.,Agapi, A. Oprescu, A.M., and Kielmann, T. 2006. Experiences Deploying Parallel Applications on a Large-scale Grid. In Proc. of EXPGRID – Experimental Grid Testbeds for the Assessment of Large-scale Distributed Applications and Tools. Workshop in conjunction with (HPDC-15).

[34] xText, http://www.eclipse.org/xText