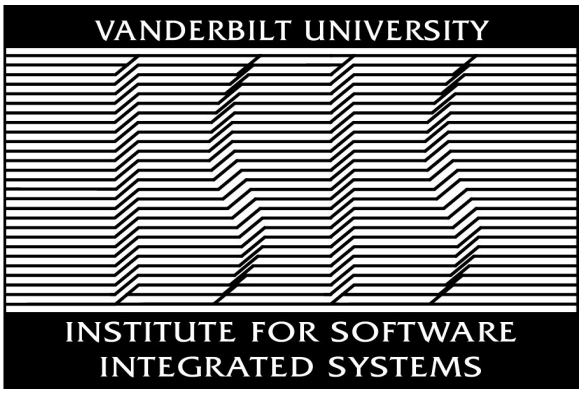

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, 37203

Towards a Product Line of Heterogeneous Distributed Applications

Subhav Pradhan, Abhishek Dubey, William R. Otte, Gabor Karsai, Aniruddha Gokhale

TECHNICAL REPORT

ISIS-15-117

4-20-2015

Abstract

Next generation large-scale distributed systems – such as smart cities – are dynamic, heterogeneous and multi-domain in nature. The same is true for applications hosted on these systems. Application heterogeneity stems from their Unit of Composition (UoC); some applications might be coarse-grained and composed from processes or actors, whereas others might be fine-grained and composed from software components. Software components can further amplify heterogeneity since there exists different component models for different domains. Lifecycle management of such distributed, heterogeneous applications is a considerable challenge.

In this paper, we solve this problem by reasoning about these systems as a Software Product Line (SPL) where individual dimensions of heterogeneity can be considered as product variants. To enable such reasoning, first, we present UMRELA (Universal feature-Model for distRibutEd appLicAtions), a conceptual feature model that identifies commonalities and variability points for capturing and representing distributed applications and their target system. This results in a product line of a family of distributed applications. UMRELA facilitates representation of initial configuration point, and the configuration space of the system. The latter represents all possible states the system can reach and is used as an implicit encoding to calculate new configuration points at runtime. Second, we present a prototype Application Management Framework (AMF) as a proof of concept configuration management tool that uses UMRELA to manage heterogeneous distributed applications.

Towards a Product Line of Heterogeneous Distributed Applications

Subhav Pradhan Abhishek Dubey William R. Otte
Gabor Karsai Aniruddha Gokhale

Institute for Software Integrated Systems, Vanderbilt University, Nashville, TN 37203, USA

***Abstract*— Next generation large-scale distributed systems – such as smart cities – are dynamic, heterogeneous and multi-domain in nature. The same is true for applications hosted on these systems. Application heterogeneity stems from their Unit of Composition (UoC); some applications might be coarse-grained and composed from processes or actors, whereas others might be fine-grained and composed from software components. Software components can further amplify heterogeneity since there exists different component models for different domains. Lifecycle management of such distributed, heterogeneous applications is a considerable challenge.**

In this paper, we solve this problem by reasoning about these systems as a Software Product Line (SPL) where individual dimensions of heterogeneity can be considered as product variants. To enable such reasoning, first, we present UMRELA (Universal feature-Model for distRibutEd appLicAtions), a conceptual feature model that identifies commonalities and variability points for capturing and representing distributed applications and their target system. This results in a product line of a family of distributed applications. UMRELA facilitates representation of initial configuration point, and the configuration space of the system. The latter represents all possible states the system can reach and is used as an implicit encoding to calculate new configuration points at runtime. Second, we present a prototype Application Management Framework (AMF) as a proof of concept configuration management tool that uses UMRELA to manage heterogeneous distributed applications.

I. Introduction

Next generation large-scale distributed systems, such as smart cities [1], [2], [3], are dynamic, heterogeneous and multi-domain in nature. They are dynamic because the participating nodes of these systems can join or leave a cluster at any given time; heterogeneous because the participating nodes can have varying resources since some can be as simple as RFIDs and simple sensors running on batteries while others can be something more resourceful; multi-domain because these systems can comprise multiple subsystems pertaining to various domains as we show in our motivating scenario presented in Section III.

These systems can be used as a platform for hosting a variety of distributed applications, where each application can provide and/or require one or more services. Like the system itself, these applications are also dynamic, heterogeneous and cross multiple domains. These applications are dynamic because they can be added, removed, reconfigured, and updated at any time; similar to "apps" in smartphones and tablets. These dynamic applications are different when compared to traditional distributed applications that are static, one-time deployment applications. Managing such dynamic applications is a considerable challenge particularly in the case of large-scale distributed systems that are remotely deployed and therefore have limited opportunity for human interventions.

The challenge of managing dynamic applications in large-scale distributed systems is further amplified if we consider the heterogeneous nature of these applications. Heterogeneity in applications stems from their varying Unit of Composition (UoC). Some applications can be *coarse-grained,* i.e., composed from processes or actors, while others can be *fine-grained,* where they follow the Component-Based Software Engineering (CBSE) [4] approach by using software components as the UoC.

The variability in the UoC of applications gives rise to a corresponding variability in the middleware technologies used to develop the applications. For example, middleware such as Alljoyn [5], MQTT [6], ROS [7], and DDS [8] can be used for coarse-grained applications; whereas, fine-grained applications that use software components can rely on different underlying component models. Component models are at the very core of the CBSE approach. However, due to varying demands and domain-specific requirements, there exist many different component models [9]. For example, COM [10] and .NET [11] from Microsoft, , Enterprise Java Beans (EJB) [12] from Sun/Oracle, and CORBA Component Model (CCM) [13] from Object Management Group (OMG) are some of the most popular component models. CCM is a standard defined by OMG and there exists multiple implementations of this standard. This heterogeneity in application UoC and associated middleware further complicates application management.

In this paper we present a novel approach to addressing the challenges of managing distributed applications by treating the different heterogeneous instances of application design as variants of a software product line (SPL) [14], [15], [16]. The key idea behind SPLs is to support the development of a family of similar software products, rather than individual systems, through a systematic and planned reuse of common assets (commonalities) in tandem with product specific variabilities. Figure 1 presents a high level variability model of distributed applications that captures application heterogeneity.

This paper makes the following contributions: first, we show how the various dimensions of heterogeneity described above form the basis of our feature model called UMRELA (Universal feature-Model for distRibutEd appLicA-

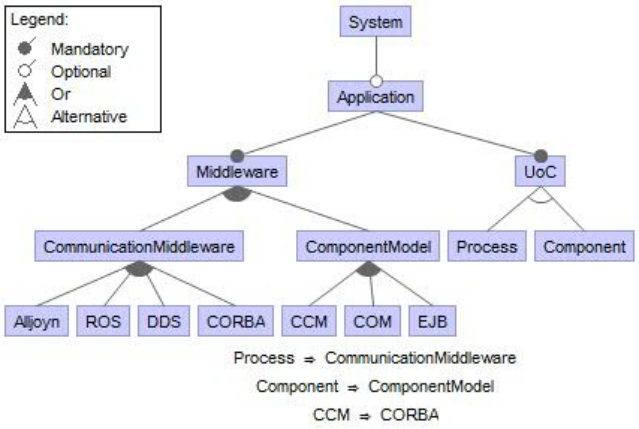

Fig. 1. High level application variability model.

tions). UMRELA is a conceptual feature model that identifies commonalities and variability points for capturing and representing distributed applications, with varying UoC, and their target system. It allows us to create a product line of a family of heterogeneous, distributed applications. UMRELA facilitates representation of both (a) the initial configuration point of the system, which is required for initial deployment of related applications, and (b) the configuration space of the system. The configuration space is an implicit encoding that can be used to calculate new configuration points at runtime if the system needs to reconfigure to mitigate failures or anomalies.

Second, we present the architecture of a prototype Application Management Framework (AMF), which is a configuration management tool that leverages UMRELA to manage heterogeneous applications. The architecture of AMF comprises platform-independent and platform-specific modules. The former uses UMRELA to monitor the configuration space and determine generic application management actions. The latter uses specific plugins to handle platform specific management actions depending on the UoC.

The rest of the paper is organized as follows: Section II presents related research work and compares them with UMRELA; Section III describes a concrete motivating scenario for our work presented in this paper; Section IV describes UMRELA in detail; Section V presents a use case scenario by describing the prototype AMF; and finally Section VI provides concluding remarks and alludes to future work.

II. Related Work

In this section we present existing related research and compares them to our work presented in this paper. We roughly classify existing work into three groups - (a) Model Driven Engineering (MDE) for large scale distributed applications, (b) standardization activities, and (c) management infrastructure for large scale distributed applications.

A. MDE approach for large scale distributed applications

In [17], authors present K@RT, an aspect-oriented and model-driven framework for supervising component-based systems. The purpose of this framework is to facilitate development, test and validation of Dynamic software Product Lines (DSPL) [18]. DSPLs are useful for modeling adaptive systems as they can reconfigure themselves at runtime by modifying corresponding feature model. K@RT is based on a generic and extensible metamodel to represent component-based systems.

This work was extended in [18] by presenting a framework, called DiVA, for managing dynamic variability in adaptive systems. DiVA can be used to model a dynamic component-based system. The different metamodels resulted in different domain specific modeling languages that can be used to model (a) feature model that describes the system's variability, (b) context model that specifies the system's environment, (c) reasoning model that describes selection of features based on context, and (d) architecture model that describes the component-based architecture. In general, the idea here is to observe system changes. Any change in the system can possibly result in changes in the context model which then results in the update of the feature model (i.e. DSPL). The updated features of the feature model is converted to the aspect model in order to obtain corresponding global configuration which is first checked for correctness and then used to derive appropriate configuration actions to adapt the system from the current configuration to the one obtained from the abstract model representation of updated feature model.

DiVA is a very interesting approach to achieving dynamic systems by using DSPLs. However, this work is strictly related to fine-grained applications composed of software components. One of the key challenges we are trying to address in our work presented in this paper is to design a model that can be used for heterogeneous applications. Another key difference is how DiVA performs reconfiguration. Unlike DiVA, we do not consider UMRELA to be a DSPL. As such, modifying the feature model itself does not initiate reconfiguration. In our approach, configuration points and space can be represented using UMRELA and these are what drive our reconfiguration logic.

B. Standardization activities

An advantage of technology standardization is the possibility of interoperability. Having a common source is beneficial in adding mechanisms for interoperability between implementations that are based on the common source. In case of coarse-grained applications, processes or actors can use different kinds of middleware and frameworks, as shown in Figure 1, in order to communicate with others in a large scale distributed scenario. Standardization at this level of application granularity would mean standardizing the various middleware that processes or actors can use. To the best of our knowledge, there does not exists any previous work related to this. Since these are vendor specific products, standardization might not be feasible at all. Solutions towards interoperability between these middleware is something that we are interested in and is part of our future work but is out of scope for this paper.

However, for fine-grained applications composed from software components, there exists some component model standards but even these standards are domain-specific. For example, the Object Management Group's (OMG) CORBA Component Model (CCM) [13] is one of the most common component model standard that has different variations. Light-weight CCM (Lw-CCM) [19] is one such variation of CCM that is targeted towards resource-constrained Distributed Real-time Embedded (DRE) systems. Other examples of component model standardization are AUTOSAR [20] and EJB; AUTOSAR is targeted towards automotive domain, whereas, EJB is targeted towards server-side components. This shows that, even though there exists standard component models, they are specific to certain domains and still necessitate a common abstract representation if they were to be used in a multi-domain scenario.

Currently, OMG is leading the effort towards achieving a Unified Component Model (UCM) [21]. UCM is OMG's approach to evolve Lw-CCM, by overcoming its shortcomings, and eventually replace Lw-CCM. One primary shortcoming of Lw-CCM is that it is tightly coupled to the CORBA standard. Therefore, one of the key requirements of UCM is to achieve a component model which is completely independent of any distribution middleware; the middleware should be completely pluggable via connectors. Another UCM requirement is better MDE support. This requirement exists to facilitate standard meta-models for UCM concepts, which will eventually result in tools and vendor independent models. One possible approach of doing this, as proposed by CEA [22], is to use UML to model UCM concepts. This approach is very close to our work presented in our paper. However, the obvious difference is that UCM is targeted only towards fine-grained, component-based applications. Having said that, we will be following UCM's progress closely.

C. Management tools for distributed applications

Large scale distributed systems such as smart cities are dynamic and remotely deployed. This necessitates application management tools that are capable of managing remotely deployed distributed applications of varying granularity. For coarse-grained applications, various Configuration Management (CM) tools such as Chef[1], Puppet[2] could be used to perform remote deployment, configuration and updates. However, these CM tools by themselves cannot be used as runtime management tools. Furthermore, these tools are not intended to support management of component-based applications. Deployment specific tools such as Capistrano[3] and Fabric[4] have similar issues. Docker[5] with compatible deployment tool such as Centurion[6] can be used to deploy docker containers but these are not suitable for component models such as CCM which require explicit wiring between components that use point-to-point interactions.

[1]https://www.chef.io/chef/
[2]puppetlabs.com
[3]capistranorb.com
[4]www.fabfile.org
[5]https://www.docker.com
[6]https://github.com/newrelic/centurion

For fine-grained applications composed from software components pertaining to different component models, there exists multiple management tools. However, these tools are specific to their component models; there does not exist any generic management tool or framework. DeployWare [23] can be used to deploy Fractal [24] components in grid environment. GoDIET [25] can be used to deploy DIET components in grid environment. LE-DAnCE [26] can be used to deploy Lw-CCM components. In this paper, we present a application management framework that, given appropriate plugins, can be used as a generic management tool for heterogeneous applications.

III. Motivational Scenario

In this section we present Emergency Response System (ERS) as a smart city application that spans multiple domain to highlight the challenges. As shown in Figure 2, ERS consists of 5 different domains where each domain can be treated as a different sub-system. Following are brief descriptions of each domain:

Domain A represents smart buildings equipped with smart devices – such as smart smoke detectors, thermostats, elevators – that use micro-controllers. Since devices in this domain use micro-controllers we can assume fine-grained applications that are composed from components with μ-Kevoree [27] as an example of the underlying component model. These applications would detect emergency situations and report to appropriate remote application servers in *Domain B*.

Domain B represents servers for smart building applications. These application servers are remotely hosted on some cloud or datacenter and are responsible for receiving incident reports from smart buildings, processing the report and forwarding meaningful information to application in *Domain C*. Timing and reliability is crucial for these applications. Furthermore, these server applications will need to handle a large amount of data. Therefore, applications in this domain can be coarse-grained and implemented on top of data oriented middleware such as DDS [8], [7], which also supports flexible Quality of Service (QoS) properties.

Domain C represents a sub-system consisting of a cluster of small satellites. In our scenario, this sub-system is used to provide GPS location dissemination application, which takes input from an application in Domain B, calculates precise GPS location of the incident, and sends it to smart Road Side Units (RSUs) in *Domain D*. Applications in this domain can be built using F6COM [28], which is designed for dynamic, resource-constrained, embedded systems such as a cluster of satellites.

Domain D represents sub-system consisting of smart RSUs. In our scenario, this sub-system is used to provide GPS notification (location forwarding) application. This application takes the GPS location as input from the corresponding application in *Domain C* and forwards that information to nearby emergency response vehicles. Since RSUs

[7]Data Distribution Service (DDS) is an OMG standard that has been implemented by different vendors.

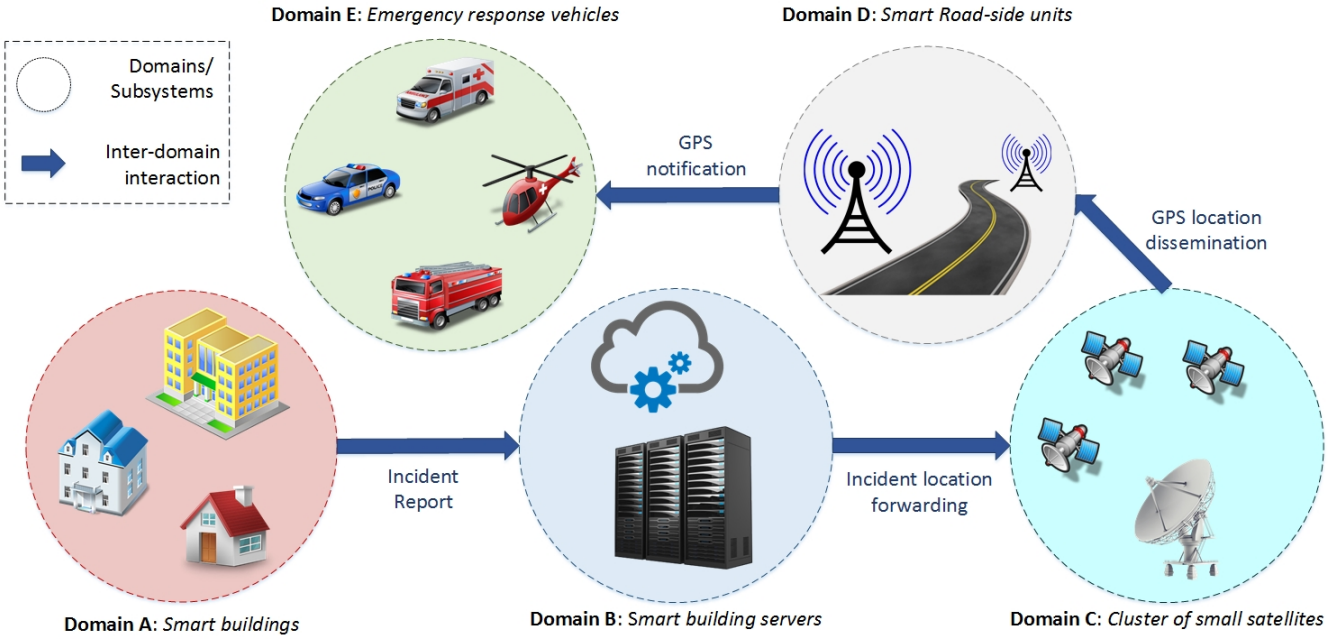

Fig. 2. Emergency Response System consisting of five domains/sub-systems.

are embedded systems that require real-time properties, Light weight CORBA Component Model (LwCCM) [19] can be used as the underlying component model to build fine-grained, component-based applications. Connectors are required for this domain as well.

Domain E represents a sub-system consisting of emergency response vehicles such as police vehicles, fire trucks, and ambulances. In our scenario, this sub-system is used to receive GPS notification from *Domain D* so that the aforementioned emergency response vehicles can attend to the initial incident site and provide the required services. AUTOSAR [20] is one component model that can possibly be used for applications in this domain.

The above description of the ERS system shows the need for a of a common application representation model that can be used to manage such heterogeneous applications. Furthermore, requirements for inter-domain interaction and therefore communication interoperability across different middleware is also evident. Interoperability for large-scale, dynamic, and heterogeneous distributed system is of keen interest to researchers working in the field of the Internet of Things (IoT) [29]. However, communication interoperability is out of scope for our work presented in this paper and is part of our future work.

IV. UMRELA Feature Model

This section describes the UMRELA feature model in detail. We first present an overview of a target system architecture for UMRELA. Second, we present the core feature model and describe the different categories and entities of UMRELA to show how it can be used to represent heterogeneous applications with varying UoC. Third, we show how configuration points and space represented using UMRELA can be used to re-configure applications.

A. System overview

The target system consists of cluster of remote deployed devices. As shown in Figure 3, each device hosts an Operating System (OS). It also hosts one or more communication middleware that are compatible with the underlying OS. A communication middleware essentially provides well defined patterns for both local and remote connections. Some example of such communication middleware that are relevant for large-scale distributed systems are different implementations of OMG's DDS [8], Alljoyn [5], MQTT [6].

Each device can also host one or more component models to facilitate fine-grained applications. Component models depend on specific underlying middleware to allow components to interact with each other. For example, Lw-CCM, which is an implementation of OMG's CCM standard, requires underlying middleware that is an implementation of OMG's CORBA standard.

Furthermore, each node can host multiple applications and each application can use different middleware or can be composed using components pertaining to different component model. Each application can have multiple implementations. However, regardless of the implementation, applications can provide and/or require one or more services. Since the target system is composed of remotely deployed devices, we require a distributed infrastructure that is capable of handling lifecycle of aforementioned heterogeneous applications. As a solution, Section V presents a prototype Application Management Framework (AMF).

B. Feature model description

Figure 4 presents a version of the simplified UMRELA feature diagram. For better understanding, UMRELA can

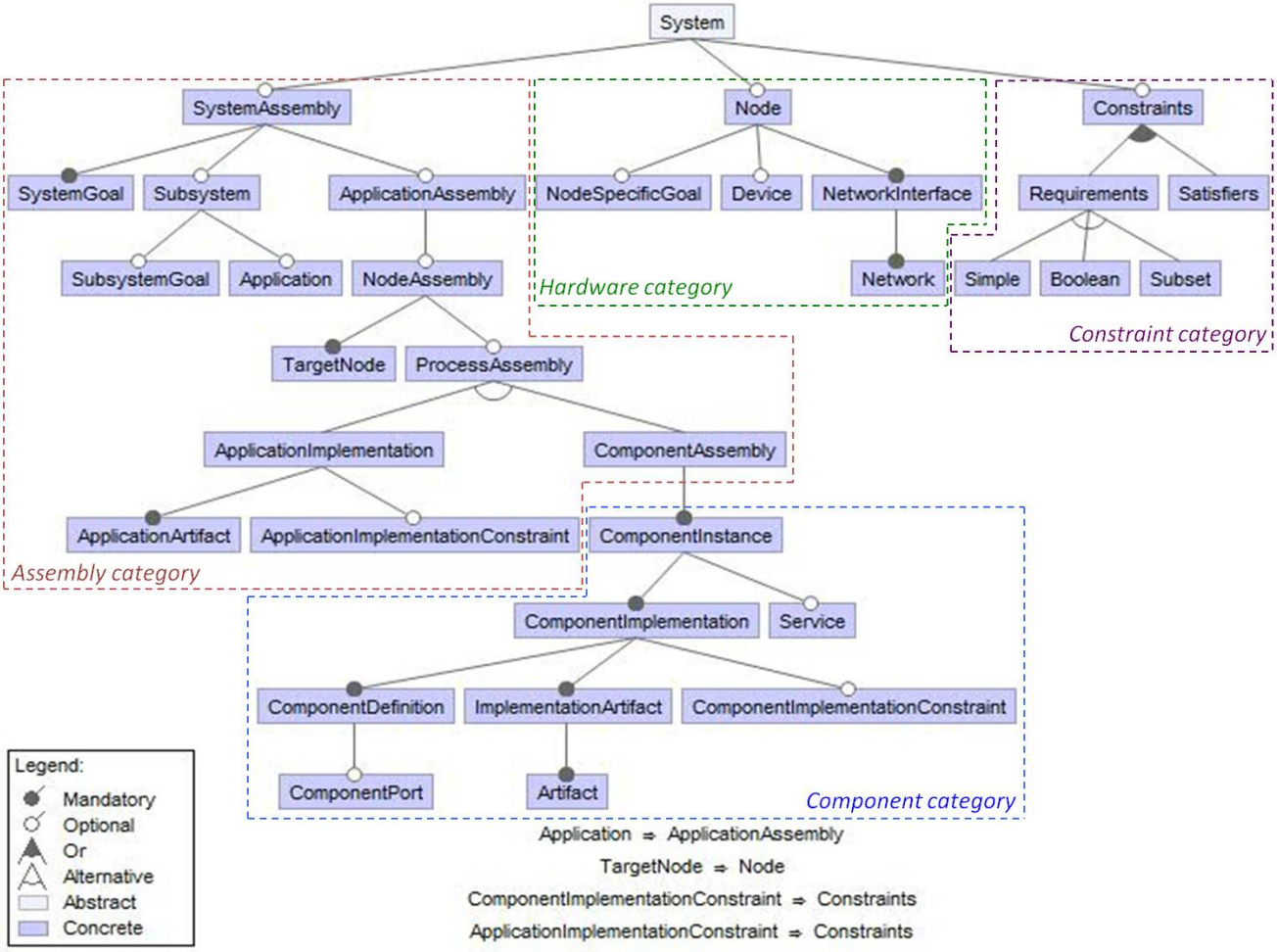

Fig. 4. Overview feature diagram of UMRELA.

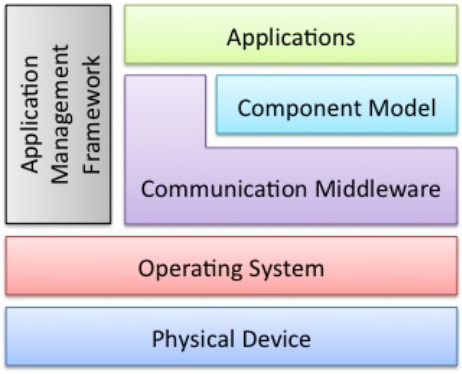

Fig. 3. Overview of target system architecture.

be classified into five different categories - (a) assemblies category, (b) components category, (c) connections category, (d) hardware category, and (e) constraints category. The feature diagram shown in Figure 4 consists of features belonging to assemblies, components, hardware, and constraints categories. We use UML class diagrams to better explain commonalities and variability points of some categories which otherwise might not be clear from the feature model in Figure 4. Following is a detailed description of each category of UMRELA:

1) Assembly Category: Figure 5 presents a UML class diagram consisting of entities belonging to the assembly category. This category consists of entities that allows hierarchical structuring of applications allowing various entities related to an application to be represented in a hierarchy that corresponds with the system architecture. Namely, as shown in Figure 4 and Figure 5, there exists a top level *SystemAssembly*, which represents a system level assembly that either contains subsystems or application assemblies. Each *Subsystem* represents part of the system, and is specially useful in cases where a system is multi-domain, like the ERS example presented in Section III, and each domain of the system forms a separate part of the overall system.

Application assemblies are either contained within *SystemAssembly* or *Subsystem* depending on whether the system is divided into multiple subsystems. Each *ApplicationAssembly* represents an application and it consists of node assemblies. Since we are considering distributed applications that are deployed on distributed system consisting of multiple participating nodes, a *NodeAssembly* entity is used to represent parts

of an application that belongs to a particular node. Therefore, an application assembly can consist of multiple node assemblies. Each *NodeAssembly* consists of process assemblies, where each *ProcessAssembly* represents parts of an application that belongs to a particular process. Each *ProcessAssembly*, in turn, consists of multiple component assemblies, which is useful only for fine-grained applications composed of one or more *ComponentInstance*. For coarse-grained applications, *ApplicationImplementation* and *ApplicationArtifact* can be used to represent appropriate application implementation and artifact.

ApplicationAssembly, NodeAssembly, ProcessAssembly and ComponentAssembly entities are deployable entities and therefore have associated states (INACTIVE, ACTIVE, FAULTY), and commands (DEPLOY, TEARDOWN, NOACTION). Furthermore, each of these entities can also contain *Constraints*, *Connection* and *AssemblyPort*. Constraints are described with further detail in Section IV-B.5. Connection and assembly port entities are used to represent connections between different parts of an application. They are present at every layer of the assembly hierarchy because connections can be established at different levels. For example, there can be connections between two component instances present in different component assemblies of a process assembly; in this scenario the process assembly entity will store appropriate connection since the connection is between elements of component assemblies that are part of the process assembly. Furthermore, in order to represent this connection, the actual communication ports of the component instances needs to be propagated to their respective component assemblies.

System assemblies also contain connections, which are used to represent interaction between subsystems and their applications. Also, system assemblies contain *Goal*, which is an entity used to represent the goal of the system. A system's goal is fulfilled via services provided by different applications that belongs to that system. For example, the goal of the ERS system presented in Section III is to be able to provide automatic emergency response for various incidents that can happen in smart buildings. Different domains of this system have their own goals and combination of these domain goals yields in fulfillment of the overall system goal.

In general, entities of assembly category are used to represent an application's layout with respect to the system. As such, the variabilities arise from how the same application or application with similar purpose can have varying layouts depending on the system they will be deployed on. As a simple example, we can consider a scenario where we have an application that can be distributed across varying number of nodes depending on how many nodes are available. In this scenario, we can have multiple variants (products) of the application where each variant has different number of node assemblies. Another example is when similar applications are implemented using processes and components.

2) Connection category: Due to space constraint, we do not present a UML class diagram for connection category. Furthermore, it is also true that features in this category

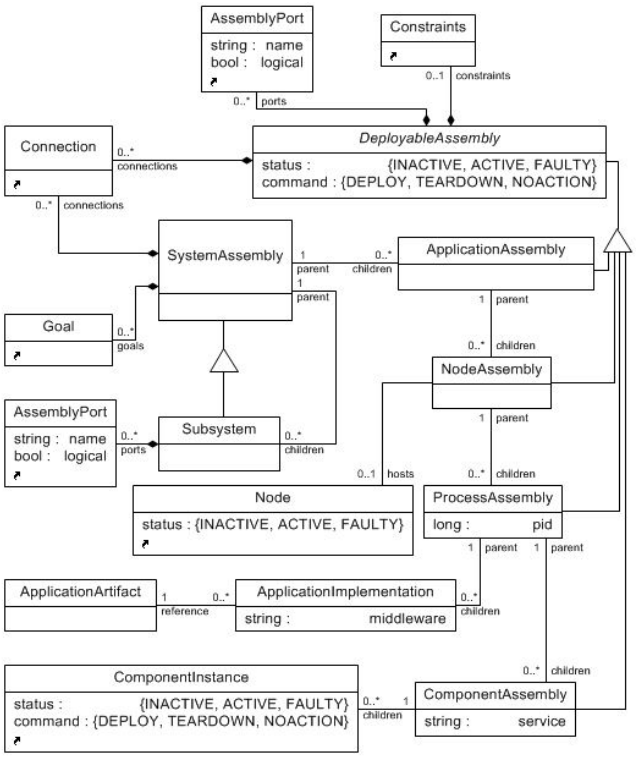

Fig. 5. UML class diagram for Assembly category.

are less likely to vary as this category only consists of features required to represent interactions at different levels of assembly hierarchy, as previously explained in Section IV-B.1. The main purpose of this category is to ease connection management, which can be very complicated for large-scale distributed applications if all connections at different level of hierarchy is represented using a flat model.

Basically, this category consists of features that allows communication ports to be propagated from one level of assembly hierarchy to another. For each such port abstraction, there exists another feature that can be used to identify the source of the abstraction.

3) Component catogery: Figure 6 presents a UML class diagram consisting of entities belonging to component category. Entities of this category are only used for fine-grained applications as it allows representation of various related entities. The *ComponentInstance* entity represents the actual component that will be deployed in the system. Each component instance uses a *ComponentImplementation*, while a component implementation implements a *ComponentDefinition* and contains a logical grouping of different artifacts that are required to instantiate a component instance. A component implementation contains multiple *ImplementationArtifact* each of which refers to an *Artifact* in the filesystem. A component definition, via ports (interfaces), defines interactions of a component. A component definition can be implemented by multiple different component implementation and among those various implementations, a component instance uses one.

Furthermore, since some component models (e.g. Frac-

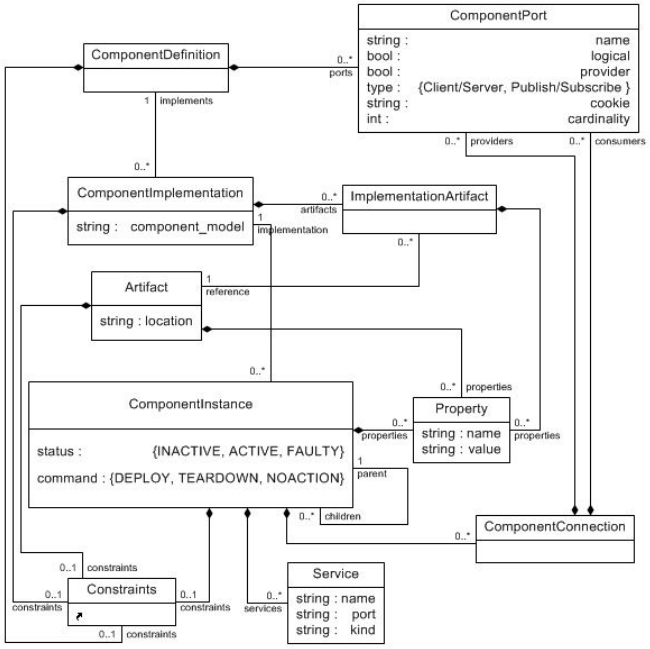

Fig. 6. UML class diagram for Component category.

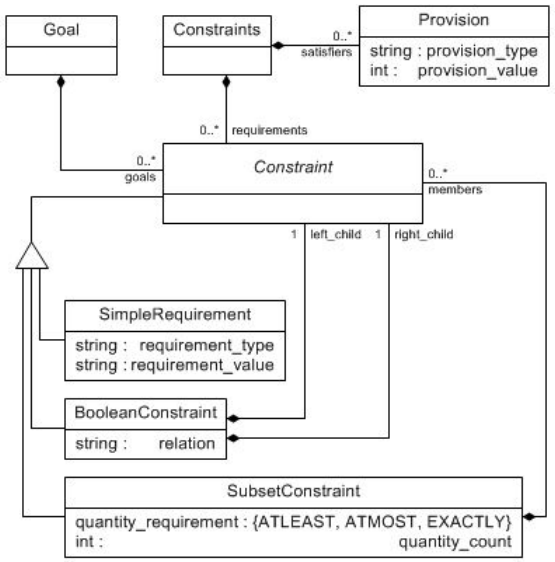

Fig. 7. UML class diagram for Constraint category.

tal [24]) support hierarchical components, component instances are recursive. Each component can have associated *Constraints*, which allows representation of any requirements that a component might have or any requirement that a component could satisfy. Also, each component can provide or require services; this is represented using *Service* entity.

In this category, variabilities can arise due to different component instance, implementation, definition and artifacts. A simple and most common scenario is when an application has multiple variants, each with a different implementation and related constraints. Another example could be a scenario where we a variant in which a component instance does not expose its service, and another variant which does by using the service feature.

4) Hardware category: As shown in Figure 4, the hardware category of UMRELA represents different nodes that are part of the distributed system. Each *Node* could possibly contain multiple *Device*, such as camera, thermal sensors, or GPS. Each node also consists of *NetworkInterface*, which contains information about associated IP addresses. Also, each network interface is part of a *Network*. Therefore, a node with multiple network interfaces can be part of multiple networked cluster. A node could be associated with node-specific goals as well.

This category of UMRELA, essentially, models the system on which different applications are hosted. As such, any variation with respect to features of this category would represent a different configuration of the system and not the applications. However, any modification to features of this category might render some existing application products to be invalid.

5) Constraints category: Figure 7 presents a UML class diagram consisting of entities belonging to constraints category. Constraints are associated with various features in different categories of our feature model, as shown in Figure 5 and Figure 6. The *Constraints* feature essentially represents requirements and satisfiers. Requirements are represented as *Constraint* and satisfiers are represented as *Provision* in the UML class diagram. Satisfiers are represented as a type/value pair. Requirements can be of three types - *SimpleRequirement*, *BooleanConstraint*, and *SubsetConstraint*.

Simple requirement, as the name suggests, can be used to represent very simple requirements as type/value pair. For example, memory requirement for a particular component implementation can be represented as *type: memory, value: 64 MB*. Boolean constraints can be used to represent more complex constraints as boolean tree that can be n-level deep and each node in the boolean tree can be any of the three requirements type. However, the leaf nodes of the tree needs to be anything but a boolean constraint.

Constraint category also includes *Goal* feature which is related to system assemblies, subsystems, and nodes. A goal is essentially used to represent purpose to entities it is related to. For example, the entire ERS presented in Section III, has a collective goal of providing emergency response to smart buildings. However, each of its subsystems could also have their own goals, for example, the smart buildings subsystem has the goal of sensing dangerous situations and relaying related information promptly. A goal when related to a node becomes a node-specific goal. A node-specific goal is only used when certain system or subsystem goals are explicitly tied to a node due to certain classification of resources available in that node (or for other similar reasons). A goal is represented as a collection of different types of constraints.

C. Application reconfiguration

Using features of different categories, UMRELA can be used to represent the initial configuration point as well as the configuration space of a system. As shown in Figure 8, the

configuration space of a system consists of multiple configuration points among which one is the initial configuration point. A configuration point is essentially a mapping of one or more applications present in the system to the hardware nodes of the system. Adding, updating, or removing applications, hardware nodes and related constraints results in the associated configuration growing or shrinking in size.

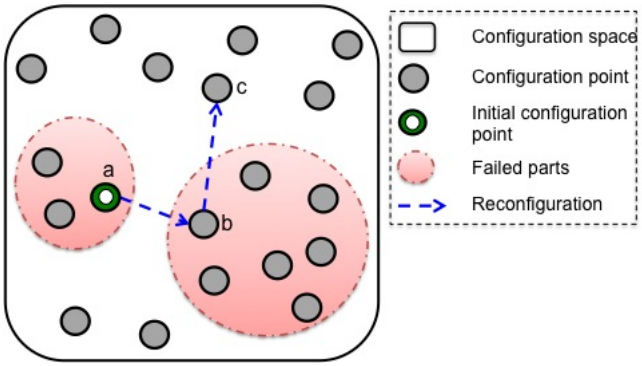

Fig. 8. Configuration space and re-configuration.

When failures or anomalies occur due to faults or unexpected changes in the system, the current configuration point, which represents the current state of the system is rendered faulty. Since we have access to the configuration space, we can calculate a new configuration point at runtime by pruning the configuration space. Once a new configuration point is calculated we can reconfigure the system by migrating the system from the current faulty configuration point to a newly calculated configuration point.

Detailed description of the runtime configuration calculation process is out of scope for this paper. In general, a constraint solver can be used to check the configuration space for various requirements, provisions, system goals, application dependencies and deployment constraints. All of this information can be encoded as a Satisfiability Modulo Theories (SMT) [30] problem, for which the constraint solver will find a solution, i.e. a valid configuration point, if there is one in the configuration space.

In Figure 8, *configuration point a* denotes the initial configuration point. Three configuration points including the initial configuration point are affected by a fault, after which *configuration point b* is computed as the new configuration point and the system migrates to the new configuration point via reconfiguration. A second fault occurs in the system rendering *configuration point b* to be faulty as well. Again, a new configuration point, *configuration point c*, is calculated and the system migrates to this configuration point in the space. While a constraint solver can be used to calculate new configuration points in the system, a management infrastructure is required to actually reconfigure the system from one point to another. For this very purpose, we present an Application Management Framework (AMF) in Section V.

V. Use Case Scenario

In this section we present a prototype Application Management Framework (AMF) as a proof of concept configuration management system that uses UMRELA. The AMF presented in this section is capable of managing applications modeled using UMRELA; it uses UMRELA to access, maintain and evolve system configuration by deploying, configuring, reconfiguring, and removing applications from the system. To better describe the prototype AMF and how it uses UMRELA, we present an architectural overview of AMF.

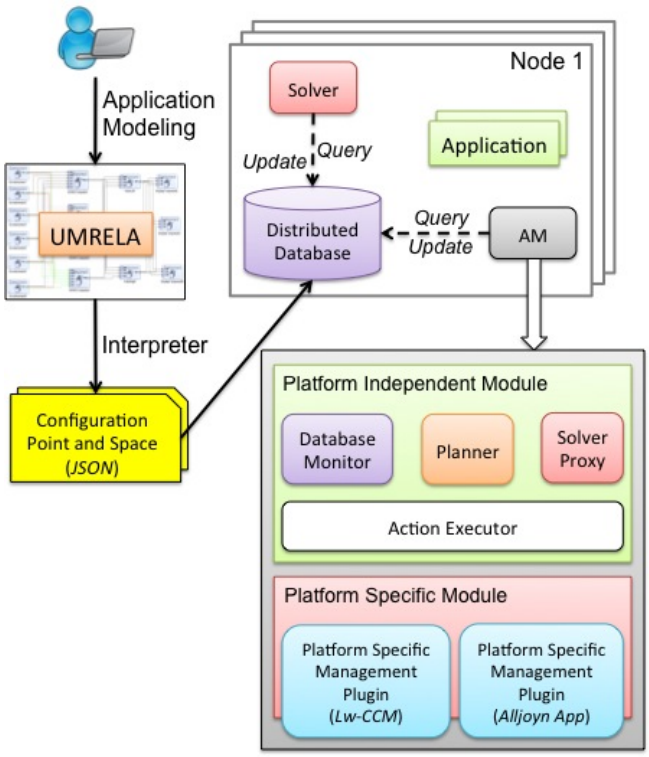

Fig. 9. AMF architecture overview.

Figure 9 presents the architecture of the AMF and shows how it fits in the overall system consisting of a modeling front-end based on UMRELA, a generative interpreter, distributed database, and solver. As shown in the figure, each node in the system hosts (a) a local instance of the distributed database, (b) a solver, which is used to calculate new configuration points at runtime, (c) multiple user applications, and (d) an Application Manager (AM). Different AMs present in a system constitutes the AMF.

Applications are designed using a Domain Specific Model Language (DSML) based on UMRELA. The interpreter associated with the aforementioned DSML is capable of interpreting an application model and generating corresponding configuration points and space as a set of JSON files. These generated JSON files are then stored in the distributed database for which we are using MongoDB [31]. System configuration information stored in this distributed database is accessed and updated by solvers or AMs. A solver updates the database when it calculates a new configuration point. Whereas, an AM updates the database to reflect consequences of various management actions. Since we are using document/aggregate oriented databases, there does not

exist an explicit schema that is enforced by the database. Therefore, solvers and AMs use UMRELA as an implicit schema to interact with the database.

Figure 9 also presents a detailed architecture of an AM. Each AM has a platform-independent module and a platform-specific module. The platform-independent module consists of the following:

Database Monitor: It is responsible for monitoring local instance of the distributed database. This functionality is achieved by monitoring the log file, which is updated by the database. For example, let's consider a scenario where a new application needs to be deployed. Once the configuration related JSON files are generated and injected into the database, its log file gets updated accordingly. This update in the log file is observed by the Database Monitor and information about each update is provided to the Planner.

Planner: It is responsible for analyzing changes observed by the Database Monitor. For each update, or a group of related updates, the planner figures out if any application management action needs to be taken. In order to do this, the Planner first checks to see if the changes require any local action. For example, when deploying a new application, Planners in each AM checks to see if any part of the new application needs to be deployed in their node. Once a planner determines that local actions are required, it calculates a set of actions, i.e. a plan, that needs to be executed. This plan is what transitions the system from current configuration point to the one stored in the database, which includes the new application that needs to be deployed.

Solver Proxy: It is responsible for communicating with the collocated Solver. This communication is required by the AM to be informed about any configuration changes made by the Solver in response to failures or anomalies.

Action Executor: The Action Executor is responsible for executing management actions computed by the Planner. In order to do so, the Action Executor analyzes each action and passes them on to the platform-specific module such that management actions are forwarded to appropriate plugins.

The platform-specific module, as shown in Figure 9, is a collection of Platform Specific Management Plugins (PSMP). Each PSMP is responsible for handling management actions for a specific component model or middleware. A PSMP receives generic commands from the *Action Executor* part of platform-independent module, after which, it converts that generic command to a set of platform-specific commands and carrying out those command.

VI. Conclusions and Future Work

In this paper we first presented UMRELA, which is an abstract feature model that identified commonalities and variability points for capturing and representing heterogeneous distributed applications that can have varying Unit of Compositions (UoC) and use different middleware. We motivated the need for UMRELA by presenting the Emergency Response System (ERS), as a large-scale multi-domain distributed application. Using UMRELA, different heterogeneous distributed applications can be represented as a product line family where each application representation becomes a product of the family. An application can have multiple implementations and therefore multiple representation; in this case, the same application will have multiple products with varying degree of common features.

This paper also presents a prototype Application Management Framework (AMF) as a proof of concept configuration management system that uses UMRELA. AMF is a distributed framework that uses UMRELA to manage lifecycle of heterogeneous applications. Current version of AMF includes a platform specific management plugin for the Lightweight CORBA Component Model (Lw-CCM). As such, the AMF is capable of managing distributed applications composed of Lw-CCM components.

Our future work includes plans to (a) extend the AMF prototype presented by implementing additional platform-specific management plugins to handle management actions for different middleware and component models, (b) develop mechanisms to facilitate communication interoperability amongst heterogeneous applications, and (c) deploy and manage large-scale heterogeneous applications simultaneously using a fully functional AMF. All of these problems are challenging, interesting and necessary to make progress towards next generation of large-scale distributed systems.

References

[1] K. Su, J. Li, and H. Fu, "Smart city and the applications," in *Electronics, Communications and Control (ICECC), 2011 International Conference on*. IEEE, 2011, pp. 1028–1031.

[2] P. Liu and Z. Peng, "China's smart city pilots: A progress report," *Computer*, no. 10, pp. 72–81, 2014.

[3] A. Dubey, M. Sturm, M. Lehofer, and J. Sztipanovits, "Smart city hubs: Opportunities for integrating and studying human cps at scale," *Workshop on Big Data Analytics in CPS: Enabling the Move from IoT to Real-Time Control*, 2015.

[4] G. T. Heineman and B. T. Councill, *Component-Based Software Engineering: Putting the Pieces Together*. Reading, Massachusetts: Addison-Wesley, 2001.

[5] A. Alliance, "Alljoyn," https://allseenalliance.org/.

[6] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s?a publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008, pp. 791–798.

[7] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[8] *Data Distribution Service for Real-time Systems Specification*, 1.2 ed., Object Management Group, Jan. 2007.

[9] K.-K. Lau and Z. Wang, "A survey of software component models," in *in Software Engineering and Advanced Applications. 2005. 31 st EUROMICRO Conference: IEEE Computer Socity*. Citeseer, 2005.

[10] D. Box, *Essential Com*. Addison-Wesley Professional, 1998.

[11] A. Wigley, M. Sutton, S. Wheelwright, R. Burbidge, and R. Mcloud, *Microsoft. net compact framework: Core reference*. Microsoft Press, 2002.

[12] L. G. DeMichiel, "Enterprise javabeans specification, version 2.1," 2002.

[13] *The Common Object Request Broker: Architecture and Specification Version 3.1, Part 3: CORBA Component Model*, OMG Document formal/2008-01-08 ed., Object Management Group, Jan. 2008.

[14] P. Clements and L. Northrop, "Software product lines: practices and patterns," 2002.

[15] G. Böckle, F. J. van der Linden, and K. Pohl, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[16] D. M. Weiss *et al.*, "Software product-line engineering: a family-based software development process," 1999.
[17] B. Morin, O. Barais, and J.-M. Jézéquel, "K@ rt: An aspect-oriented and model-oriented framework for dynamic software product lines," in *Proceedings of the 3rd International Workshop on Models@ Runtime, at MoDELS'08*, 2008.
[18] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, 2008.
[19] *Light Weight CORBA Component Model Revised Submission*, OMG Document realtime/03-05-05 ed., Object Management Group, May 2003.
[20] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange, "Autosar–a worldwide standard is on the road," in *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, 2009.
[21] "Unified Component Model for Distributed, Real-Time and Embedded Systems RFP," http://www.omg.org/cgi-bin/doc?mars/2013-09-10.
[22] "Specifying a Unified Component Model with UML and its Extension Mechanism," http://www.omg.org/news/meetings/tc/berlin-13/special-events/component-pdfs/S3-2-Radermacher.pdf.
[23] A. Flissi, J. Dubus, N. Dolet, and P. Merle, "Deploying on the grid with deployware," in *Cluster Computing and the Grid, 2008. CCGRID'08. 8th IEEE International Symposium on*. IEEE, 2008, pp. 177–184.
[24] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.
[25] E. Caron, P. K. Chouhan, H. Dail *et al.*, "Godiet: a deployment tool for distributed middleware on grid'5000," 2006.
[26] W. Otte, A. Gokhale, and D. Schmidt, "Predictable deployment in component-based enterprise distributed real-time and embedded systems," in *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. ACM, 2011, pp. 21–30.
[27] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel, "A dynamic component model for cyber physical systems," in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*. ACM, 2012, pp. 135–144.
[28] W. R. Otte, A. Dubey, S. Pradhan, P. Patil, A. Gokhale, G. Karsai, and J. Willemsen, "F6COM: A Component Model for Resource-Constrained and Dynamic Space-Based Computing Environment," in *Proceedings of the 16th IEEE International Symposium on Object-oriented Real-time Distributed Computing (ISORC '13)*, Paderborn, Germany, Jun. 2013.
[29] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A Survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.
[30] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories." *Handbook of satisfiability*, vol. 185, pp. 825–885, 2009.
[31] MongoDB Incorporated, "MongoDB," http://www.mongodb.org, 2009.