CHARIOT: A Domain Specific Language for Extensible Cyber-Physical Systems

Subhav M. Pradhan, Abhishek Dubey, Aniruddha Gokhale
Institute of Software Integrated Systems
Department of EECS, Vanderblit University
Nashville, TN, USA
<firstname.lastname>@vanderbilt.edu

Martin Lehofer
Siemens Corporate Technology
Princeton, NJ, USA
<firstname.lastname>@siemens.com

Abstract

Wider adoption, availability and ubiquity of wireless networking technologies, integrated sensors, actuators, and edge computing devices is facilitating a paradigm shift by allowing us to transition from traditional statically configured vertical silos of Cyber-Physical Systems (CPS) to next generation CPS that are more open, dynamic and extensible. Fractionated spacecraft, smart cities computing architectures, Unmanned Aerial Vehicle (UAV) clusters, platoon of vehicles on highways are all examples of extensible CPS wherein extensibility is implied by the dynamic aggregation of physical resources, affect of physical dynamics on availability of computing resources, and various multi-domain applications hosted on these systems. However, realization of extensible CPS requires resolving design-time and run-time challenges emanating from properties specific to these systems. In this paper, we first describe different properties of extensible CPS - dynamism, extensibility, remote deployment, security, heterogeneity and resilience. Then we identify different design-time challenges stemming from heterogeneity and resilience requirements. We particularly focus on software heterogeneity arising from availability of various communication middleware. We then present appropriate solutions in the context of a novel domain specific language, which can be used to design resilient systems while remaining agnostic to middleware heterogeneities. We also describe how this language and its features have evolved from our past work. We use a platform of fractionated spacecraft to describe our solution.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques

Keywords System description language, Model-driven development, Extensible Cyber-Physical Systems

1. Introduction

Cyber-Physical Systems (CPS) consists of sensors, actuators, network resources and computation resources that form cyber components used to monitor and control the surrounding physical environment by working closely with human operators at times. Traditionally, CPS have been designed as a single point solution with a focus on a specific domain. These systems are designed as composition of sensors, actuators, computation resource and networking technologies designed for specific purposes with self-contained resources and mostly closed architectures. This approach results in vertical silos of capabilities that do not support next generation CPS [13] – such as Smart Cities, fractionated spacecraft [5] – that require open, dynamic, extensible and interoperable solutions.

However, wider adoption, availability and ubiquity of wireless networking technologies, integrated sensors, actuators, and edge computing devices such as wearables, smart phones, tablets provides us with a great opportunity to move away from traditional CPS towards next generation, extensible CPS. The key idea behind extensible CPS is the notion of open and extensible CPS platforms. These platforms are built not as a single function system but rather as potentially loosely connected networked platforms that "virtualize" and share their resources to host multi-domain cyber-physical applications that provide a variety of objectives to satisfy system goals. Extensible CPS takes inspiration from existing fields, such as cloud computing that provide elastic and multi-tenant computing resources. However, interaction with physical devices is rarely an issue in cloud computing where everything is virtualized without consideration for management of resources that are not part of the computation platform.

Dynamism, extensibility, remote deployment, security, heterogeneity and resilience are some of the key properties of extensible CPS. These systems are dynamic because physical entities that form different platforms can join or leave a group at any time. Similarly, these systems are extensible as physical or software resource can be added to existing platform. Security is also an important property as these platforms are open and therefore host applications belonging to different organizations with varying security requirements. Extensible CPS are heterogeneous since physical nodes that form a platform, as well as devices hosted on these nodes can be of different kinds. In addition to these hardware related heterogeneities, there can also be software related heterogeneities such as communication middleware hosted on each node. Finally, resilience is required because anything can go wrong at any time. As such, the platform must be able to handle both internal faults as well as environmental changes while ensuring that system properties and requirements are met.

The aforementioned properties of extensible CPS result in design-time and run-time challenges that need to be resolved in order to realize these systems. In this paper, we present our initial work, which focuses on design-time challenges arising specifically

DSM '15, October 27, 2015, Pittsburgh, PA, USA.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

due to heterogeneity and resilience requirements. Below we describe each challenge addressed in this paper:

Challenge 1: The first challenge we address in this paper emanates from heterogeneous property of extensible CPS. Even though heterogeneity can be related to physical resources, we focus on software heterogeneity. To be more precise, we particularly focus on communication middleware because they serve as existing solutions to overcome hardware and operating system heterogeneity since they provide required abstractions to facilitate platform independent interaction between applications hosted on heterogeneous physical resources. However, there currently exists many different middleware solutions, such as RTI DDS [12], AllJoyn [2], MQTT [11], AMQP [18], etc. Some of these middleware, such as RTI DDS are implementation of a standard, while others are vendor-specific technologies. They all have their own advantages and disadvantages, therefore, arriving at a common solution is almost impossible and maybe undesirable. More importantly, these middleware do not provide a clean separation between the computation and communication aspects. As such, applications written using one middleware are tightly coupled with that middleware.

Challenge 2: The second challenge we address in this paper emanates from resilience requirements of extensible CPS. An extensible CPS can host applications providing different objectives to satisfy different system goals. Some of these objectives are critical and it is of utmost importance to make sure that these objectives are not affected by failures and anomalies such that requirements for associated system goals are always met. This requirement necessitates a resilient system that supports self-reconfiguration mechanisms to facilitate autonomous fault tolerance. Autonomy in the resilience mechanism is importance since extensible CPS can be remotely deployed systems with limited opportunity for human interference. To support any run-time solution that performs self-reconfiguration, we require the design-time solution to allow modeling of design-time related resilience logic – such as configuration space of a system – that can be used at run-time.

In order to resolve the above-mentioned challenges, in this paper we present a Domain Specific Language (DSL) that is part of our application architecture CHARIOT (Cyber-pHysical Application aRchItecture with Objective-based reconfiguraTion). CHARIOT DSL is a textual DSL developed using Xtext [6]. It incorporates lessons learned from previous experiences with DSLs for CPS. As such, we show how our approach to implementing DSLs has evolved and how our current solution can be used to design resilient systems while remaining agnostic to middleware heterogeneities. Following are the key contributions of this paper:

- *Contribution 1:* We address *Challenge 1* by presenting a solution that enforces clean separation-of-concerns between computation and communication aspects and therefore allows users to design cyber physical applications while remaining completely agnostic to the underlying middleware that can be used by these applications to interact with each other. We also briefly describe how our design manifests in the run-time system.
- *Contribution 2:* We address *Challenge 2* by presenting a solution that allows application designers to explicitly model systems goals, objectives and associated functionalities. These become part of a configuration space that will be used at run-time to support self-reconfiguration mechanisms.

The rest of this paper is organized as follows: Section 2 presents related work and compares them to our work presented in this paper; Section 3 uses fractionated spacecraft as a motivating scenario; Section 4 presents the evolution of our DSL and describes in detail how it facilitates design-time heterogeneity and resilience requirements; finally, Section 5 provides concluding remarks, and describes our on-going and future work.

2. Related Work

Extensible CPS are Distributed Real-time and Embedded (DRE) systems and these classes of systems are commonly constructed using component-based approach. Architecture Analysis and Design Language (AADL) [7] is one such standardized architecture description language, which has notion of software components and hardware components that can be used to architect different systems. An AADL model is essentially a tree of components that can interact with each other via connections that are modeled as features. Our work is similar to AADL since we are also trying to achieve a generic system description language. However, our solution includes a well-defined (software) component model that captures interactions as part of components but with clear distinction from the computation aspect. In addition, AADL does not provide support for modeling resilience specific entities.

Currently there exist numerous component models targeted towards specific embedded system domains. For example, the Pervasive Component Systems (PECOS) [9]project describes a component model for embedded systems that is specifically tailored to field devices. Field devices are reactive, embedded devices fitted with sensors and actuators that are developed using the most inexpensive of hardware. AUTomotive Open System ARchitecture (AUTOSAR) [8] is an open and standardized automotive software architecture that supports a component model and is specifically targeted towards supporting vehicular design. These component models are specific to certain domain and they are, more often than not, tied to a specific middleware.

3. Motivating Scenario

Consider a platform of fractionated spacecraft, which is a cluster of independent satellite modules flying in formation and communicating with each other via ad-hoc wireless networks. Each independent satellite that is part of a fractionated spacecraft cluster, can come from different organization. Together these independent satellite modules provide an extensible CPS platform that facilitates sharing sensors and other computing and communication resources across multiple applications. This architecture can realize the functions of monolithic satellites at a reduced cost and with improved adaptability and robustness [4]. Several existing and future missions use this type of architecture including NASA's Edison Demonstration of SmallSat Networks, TANDEM-X, PROBA-3, and PRISMA from Europe. In each of these missions, the cooperating fractionated satellites are expected to provide the foundations for applications, used by many, possibly concurrent missions.

Individual satellite modules of a fractionated spacecraft cluster are present in the Low Earth Orbit (LEO), where one of the basic requirements is to be able to maintain orbital flight so that they can overcome the atmospheric drag and orbit the Earth while remaining in the LEO. Each individual satellite achieves this functionality by periodically using their thrusters to adjust their position. In addition to this flight control objective, which is required to be satisfied all the time by each individual satellite, the platform of fractionated spacecraft can be used to host different applications and depending on which application is hosted, more objectives are added and new system goals are created.

Figure 1, presents a schematic overview of an application that can be hosted on a platform of fractionated spacecraft to achieve a system whose goal is *ImageSatelliteCluster* i.e. to capture images of various resolutions and process them. As mentioned before, regardless of what applications are hosted on a fractionated spacecraft platform, it always needs to satisfy objectives required to maintain orbital flight of individual satellites. These are represented by objectives *ClusterFlightPlanning* and *SatelliteFlight* in Figure 1. The *ClusterFlightPlanning* objective is associated with

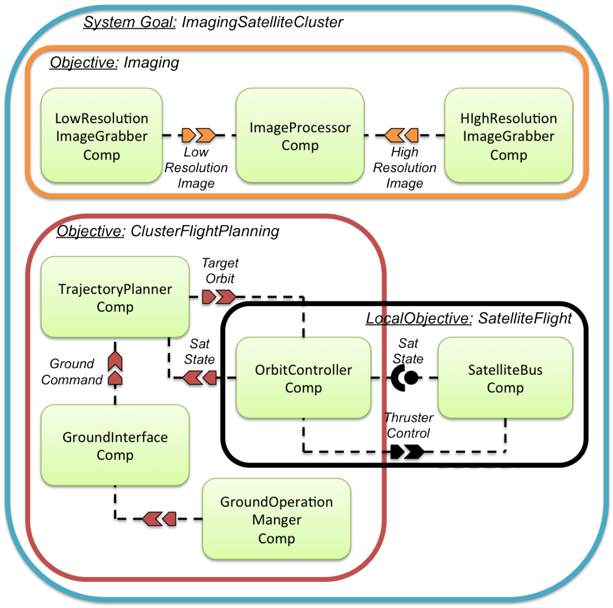


Figure 1. Imaging satellite cluster.

the tasks of receiving ground commands, processing each command and calculating flight plan (new target location) for each satellite that is part of the cluster. The *SatelliteFlight* objective is associated with the tasks of receiving the aforementioned target locations and controlling satellite thrusters to move towards that position. The *SatelliteFlight* objective is a local objective. A local objective is a kind of objective that is always associated with a category of node. A local objective implies that at least one of its functionality should be present in each node belonging to its associated node category. Imaging specific requirements are fulfilled by the *Imaging* objective.

Each objective is a collection of one or more functionalities provided by different components. For example, as shown in Figure 1, the *Imaging* objective is a collection of three functionalities provided by three different components (a) *LowResolutionImageCapture* to capture low resolution images, (b) *HighResolutionImageCapture* to capture high resolution images, and (c) *ImageProcessor* to process images with different resolutions. The components that capture low and high resolution images, publishes those images for the image processing component to consume and process.

Generally, the components are based on a component model, which determines how components are designed, composed, managed as well as how they interact using their ports. Traditional component models are tightly coupled with specific middleware solutions, for example, Component Integrated ACE ORB (CIAO) [19] uses The ACE ORB (TAO) [17]. Furthermore, these component models do not support clean separation-of-concerns between their communication and computation aspects. This becomes problematic when we consider the above described fractionated spacecraft scenario where individual or a group of satellite modules can come from different organization and therefore support different middleware. This results in communication heterogeneity and thus necessitates resolution of *Challenge 1* described in Section 1.

All three objectives of the imaging satellite cluster presented above are critical to achieving the associated system goal. As such, when there are failures or anomalies in the system, it is of utmost importance that the system adapts itself to recover and maintain all objectives for as long as possible. This necessitates resolution of *Challenge 2* described in Section 1.

4. CHARIOT DSL

This section presents detailed description of our solution and show how it resolves challenges listed in Section 1. We first present a brief description explaining how our work has evolved to meet the requirements of next generation CPS. Then we present detailed description of relevant aspects of CHARIOT DSL that addresses the different challenges.

4.1 Evolution of our DSL from prior efforts

Our initial approach [3] towards achieving a DSL for next generation CPS focused on platforms of fractionated spacecraft, which had static communication channels declared at design-time in order to guarantee secure interaction between applications at run-time. Furthermore, we had static deployment and configuration plans generated from systems modeled at design-time. These plans contained information about artifacts, parameters, and communication flows of different application components. They did not capture information about other aspects of the system such as resource availabilities, constraints, system objectives and goals; these are critical configuration information required to achieve a resilient system. Finally, since our initial approach was designed for a specific CPS domain – fractionated spacecraft – capturing and supporting heterogeneity was not a consideration at that time.

Second phase of our effort focused on advancing our initial approach by supporting resilience. We supported modeling of configuration space instead of a specific deployment plan [15]. A configuration space represents the state of an entire platform. It includes information about different resources available, well known faults, system goals, objectives, application components, where these components are deployed and how they are configured. A configuration space can contain multiple configuration points, where each configuration point represents the state of the entire platform at any given time. Resilience in the run-time was facilitated by self-reconfiguration mechanisms that used configuration space to compute and transition to a new configuration point.

Third phase of our effort is our current solution presented in this paper. In this phase, similar to the last, we allow users to model the configuration space for different systems. In addition, we also provide mechanisms to model communication heterogeneity by enforcing a strict separation-of-concerns between communication and computation logic thereby allowing users to model their applications using generic interaction patterns that can be implemented on top of any communication middleware.

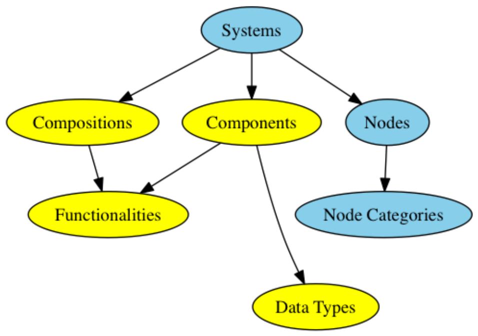


Figure 2. Modeling concepts in CHARIOT DSL and their interdependencies.

Figure 2 presents different first class modeling concepts and their interdependencies in CHARIOT DSL. The responsibility of modeling these concepts are assigned to three different roles (a) application developers, who are responsible for modeling *Data Types*, *Functionalities*, *Compositions*, and *Components* belonging to their applications, (b) SDK developers, who are responsible for modeling *Platform Interactions* corresponding to platform interaction libraries they develop, and (c) systems architects, who are responsible for modeling *Node Categories*, *Nodes*, and *Systems*. Table 1 presents a brief summary of these modeling concepts.

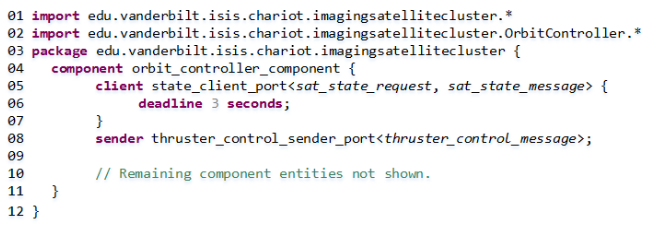
Table 1. Summary of modeling concepts in CHARIOT DSML.

Modeling concept	Description
Data types	Most basic modeling construct. It facilitates modeling of data types used for interaction as well as computation.
Functionalities	Logical concepts used to compose objectives. Functionalities are provided by components.
Compositions	Logical groups of functionalities, where each functionality can be part of multiple compositions and functionalities of same composition can have inter-dependencies. Objectives are instantiations of compositions.
Components	Applications in CHARIOT are composed of components that communicate with each other. Components have well defined ports for interaction and use workflows and tasklets to describe computational behavior.
Platform interactions	Artifacts that can be used to interact with platform specific resources.
Node categories	Categories to which different nodes belong to.
Nodes	Different nodes that are part of a platform.
Systems	A system consists of a goal that is satisfied by one or more objectives. An objective depends on functionalities provided by components.

4.2 Addressing Challenge 1: Supporting communication heterogeneity

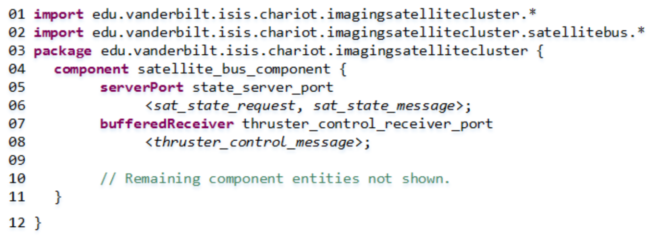
In order to facilitate communication heterogeneity, we enforce strict separation-of-concerns between the communication and computation logic of application components. Having the computation logic clearly separated from the communication logic results in more predictability, which is an important real-time property. Similarly, having a communication logic clearly separated from computation logic results in highly configurable communication and therefore support for heterogeneous communication middleware. Using software components to design distributed application is not a new concept as significant amount of prior work has been done in the field of Component-based Software Engineering (CBSE) [10]. However, as mentioned in Section 3, existing component models are generally tightly coupled to a specific middleware solution. This is why our approach is different; in essence, we are providing a universal component model, which does not depend on any specific communication middleware. This aspect of our work aligns well with current efforts of the Object Management Group (OMG) to achieve a Unified Component Model (UCM) [1].

CHARIOT supports different kinds of ports that can be used by application developers to model common interaction patterns – such as point-to-point (client/server) interaction and group publish/subscribe interaction – that are supported by most middleware solution. Therefore, this allows application developers to focus on modeling application interaction using different ports without having to worry about what middleware will be used to support those interactions at run-time. Different kinds of ports supported by CHARIOT are - (a) client port, which is used to send request to one or more server ports, (b) server port, which is responsible for receiving requests from one or more client ports, (c) buffered receiver port, which receives messages sent from one or more sender ports and stores them in a buffer of predefined size, (d) sampling receiver port, which also receives messages sent from sender ports but unlike buffered receiver port, it does not use a buffer to store multiple messages, and (e) sender port, which is used to send messages to one or more buffered or sampling receiver ports.



```
01 import edu.vanderbilt.isis.chariot.imagingsatellitecluster.*
02 import edu.vanderbilt.isis.chariot.imagingsatellitecluster.OrbitController.*
03 package edu.vanderbilt.isis.chariot.imagingsatellitecluster {
04   component orbit_controller_component {
05         client state_client_port<sat_state_request, sat_state_message> {
06               deadline 3 seconds;
07         }
08         sender thruster_control_sender_port<thruster_control_message>;
09
10         // Remaining component entities not shown.
11   }
12 }
```

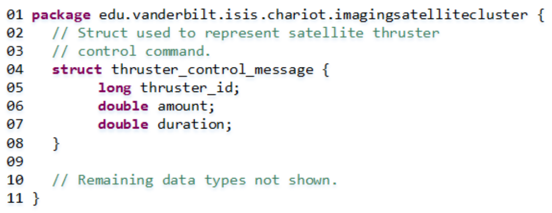
Figure 3. Snippet of OrbitController component declaration.



```
01 import edu.vanderbilt.isis.chariot.imagingsatellitecluster.*
02 import edu.vanderbilt.isis.chariot.imagingsatellitecluster.satellitebus.*
03 package edu.vanderbilt.isis.chariot.imagingsatellitecluster {
04   component satellite_bus_component {
05         serverPort state_server_port
06               <sat_state_request, sat_state_message>;
07         bufferedReceiver thruster_control_receiver_port
08               <thruster_control_message>;
09
10         // Remaining component entities not shown.
11   }
12 }
```

Figure 4. Snippet of SatelliteBus component declaration.

In order to show how interactions are modeled at design-time, Figures 3 and 4 present snippets of declaration of *OrbitController* and *SatelliteBus* components that are part of the imaging satellite cluster, previously presented in Section 3. As shown in Figure 3, the *OrbitController* component uses *thruster_control_sender_port* (line 8) as sender port to send *thruster_control_message*, and *state_client_port* (line 5-7) as client port to send *sat_state_request* and receive *sat_state_message*. Similarly, the *SatelliteBus* component, shown in Figure 4, uses *state_server_port* (line 5-6) as server port to receive *sat_state_request* and send *sat_state_message*, and *thruster_control_receiver_port* (line 7-8) as buffered receiver port in order to receive *thruster_control_message*.



```
package edu.vanderbilt.isis.chariot.imagingsatellitecluster {
  // Struct used to represent satellite thruster
  // control command.
  struct thruster_control_message {
        long thruster_id;
        double amount;
        double duration;
  }

  // Remaining data types not shown.
}
```

Figure 5. Snippet of data types declaration.

At the very core of supporting communication middleware heterogeneity is the fact that we support generic data types that are supported, in one form or other, by most existing middleware. This is what ensures interoperability between different middleware solutions, as component ports are associated with these generic data types. For example, the *thruster_control_sender_port* in Figure 3 send data to type *thruster_control_message*, whose declaration using CHARIOT DSL is shown in Figure 5. Table 2 shows how our data types map to that of Java, as well as data types supported by different middleware. Table 2 does not include MQTT [11] since it is totally data-agnostic and virtually every data can be sent in its binary format; serialization and deserialization must be handled by application developers.

Table 2. Table showing how CHARIOT DSL data types map to that of Java and different existing middleware.

CHARIOT-ML	**Java**	**RTI Connext** [12]	**AllJoyn** [2]	**ROS** [16]	**LCM** [14]	**AMQP** [18]
float	float	DDS_Float	N/A	float32	float	float
double	double	DDS_Double	DOUBLE	float64	double	float
short	short	DDS_Short	INT16	int16	int16_t	short
long	int	DDS_Long	INT32	int32	int32_t	int
long long	long	DDS_LongLong	INT64	int64	int64_t	long
char	char	DDS_Char	N/A	N/A	N/A	char
wchar	char	DDS_WChar	N/A	N/A	N/A	N/A
boolean	boolean	DDS_Boolean	BOOLEAN	bool	boolean	boolean
octet	byte	DDS_Octet	BYTE	N/A	byte	byte
string	String	String	STRING	string	string	string
struct	class	struct	STRUCT	structure	struct	composite type
sequence	array	sequence	ARRAY	array	array	array
enum	enum	DDS_Enum	BYTE, INT16, INT32, INT64	constants	constants	restricted type

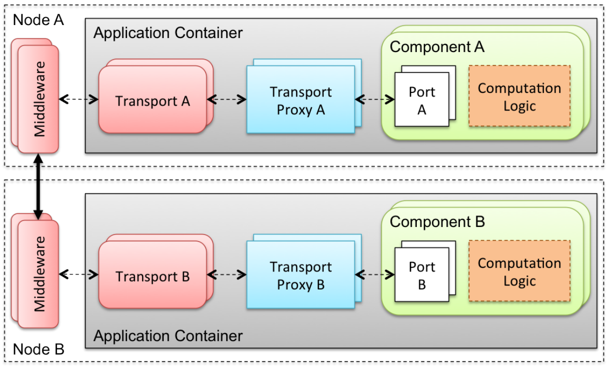


Figure 6. Run-time mapping of component communication logic.

Figure 6 shows how the CHARIOT component communication logic maps to the run-time. As shown in the figure, an application container hosts transports, transport proxies, and components. Transports allow interaction with specific middleware, whereas transport proxies facilitate interaction between transports and component ports. These entities constitute the communication logic and are strictly separated from the computation logic. Depending upon availability of appropriate transports and transport proxies, a component can use any middleware.

4.3 Addressing Challenge 2: Modeling resilient systems

Maintaining system goals is of utmost importance for extensible CPS. Therefore, resilience is a very important property of extensible CPS. In order to be resilient, a system must support run-time mechanisms to monitor, detect, diagnose, and mitigate failures and anomalies. In addition, resilience requirements of extensible CPS also necessitate design-time solutions that allow modeling of appropriate information at design-time such that they can be used to facilitate run-time resilience mechanisms.

Since our ongoing work on run-time resilience mechanism is based on self-reconfiguration capabilities, our design-time tool allows an application developer to model (a) a complete configuration space of a system, and (b) an initial configuration point, as a complete or partial deployment specification. A configuration space represents the state of an entire platform. It includes information about different resources available, well known faults, system goals, objectives and corresponding functionalities, components that provide different functionalities, where these components are deployed, and how they are configured. A configuration space can contain multiple configuration points and a configuration point represents state of the associated platform at any given time; change in state of the platform is represented by transition from one configuration point to another.

```
01 import edu.vanderbilt.isis.chariot.imagingsatellitecluster.*
02 package edu.vanderbilt.isis.chariot.imagingsatellitecluster {
03   system ImagingSatelliteCluster {
04       SatelliteFlight as
05           localObjective SatteliteFlight_Objective {
06               appliesTo Satellites nodes;
07               keep OrbitController perNode;
08               keep SatelliteBus perNode;
09       }
10       ClusterFlightPlanning as objective ClusterFlight_Objective;
11       Imaging as objective Imaging_Objective;
12       deployment {
13           ground_interface_component on SatBeta as GI_B;
14           satellite_bus_component on SatAlpha as SB_A;
15           satellite_bus_component on SatBeta as SB_B;
16           satellite_bus_component on SatGamma as SB_G;
17           orbit_controller_component on SatAlpha as OC_A;
18           orbit_controller_component on SatBeta as OC_B;
19           orbit_controller_component on SatGamma as OC_G;
20           trajectory_planner_component on SatAlpha as TP_A;
21           trajectory_planner_component on SatBeta as TP_B;
22           low_resolution_image_component on SatAlpha as LRI_A;
23           high_resolution_image_component on SatGamma as HRI_G;
24           image_processor_component on SatAlpha as IP_A;
25       }
26   }
27 }
```

Figure 7. Snippet of imaging satellite cluster system declaration.

Figure 7 presents a declaration of the imaging satellite cluster system previously presented in Section 3. In order to model a system in CHARIOT DSL, we need to declare its goal (line 3), different objectives (line 4-11), deployment constraints (not shown in Figure 7), and initial deployment specification (line 12-25). As shown in the figure, the system comprises three objectives where the *SatelliteFlight* objective is a local objective that applies to all nodes of *Satellite* category. This local objective implies that *OrbitController* and *SatelliteBus* functionalities should be present in each node of *Satellite* category.

Above described system declaration is part of the overall configuration space of the imaging satellite cluster. In the run-time resilience infrastructure, which is part of our ongoing work, we store configuration space and points in a distributed database such as MongoDB[1]. As such, a configuration space becomes a list of collection, where each collection can contain multiple documents. Due to space restriction we do not show the entire configuration space, but Figure 8 shows part of the configuration space of imaging satellite cluster by presenting snippet of the system declaration presented in Figure 7. Each system declaration stored has - (a) an associated id, (b) constraints, (c) name of the system, which also represents system goal, (d) a list of objectives, which themselves

[1] http://www.mongodb.org

```
{
        "_id": {
                "$oid": "55ba96a4acb3be94ff360f76"
        },
        "constraints": [],
        "name": "ImagingSatelliteCluster",
        "objectives": [
                {
                        "constraints": [],
                        "functionalities": [
                                {
                                        "dependsOn": [
                                                "SatelliteBus"
                                        ],
                                        "name": "OrbitController"
                                },
                                {
                                        "dependsOn": [
                                                "OrbitController"
                                        ],
                                        "name": "SatelliteBus"
                                }
                        ],
                        "name": "SatelliteFlight_Objective",
                        "nodeCategory": {
                                "$oid": "55b9c136acb3be94ff360f72"
                        }
                }

                // Remaining objectives not shown.
        ]
}
```

Figure 8. Snippet of system description that is part of the overall configuration space of imaging satellite cluster.

contain constraints, list of required functionalities and their dependencies, name, and a node category to represent objective locality.

5. Conclusions

Traditionally CPS have been designed as vertical silos of capabilities. With wider adoption, availability and ubiquity of wireless networking technologies, integrated sensors, actuators, and edge computing devices, we are moving towards the paradigm of extensible CPS wherein the sensors, actuators, and computing resources of one or more CPS form an open platform whose resources can be "virtualized" and shared among different applications. However, practical realization of extensible CPS requires us to resolve design-time and run-time challenges emanating from system specific properties such as heterogeneity and resilience.

This paper focuses on resolving design-time challenges related to communication heterogeneity and resilience. We present our solution for these challenges in the context of CHARIOT DSL, which is a novel domain specific language that supports - (a) a component model with clean separation-of-concerns between the communication and computation aspects to handle communication heterogeneity, and (b) configuration space modeling in order to capture information required to facilitate run-time self-reconfiguration mechanisms. Our ongoing work focuses on realization of a complete end-to-end system, which includes a model interpreter, a complete run-time solution that includes our universal component model, and a comprehensive resilience infrastructure. In future we plan to extend our work by using model checkers to perform formal verification of system models at design-time, and supporting redundancy patterns to enhance run-time resilience mechanism.

Acknowledgments

This work is sponsored in part by a research grant from Siemens Corporate Technology. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of Siemens Corporate Technology.

References

[1] Unified Component Model for Distributed, Real-Time and Embedded Systems RFP. http://www.omg.org/cgi-bin/doc?mars/2013-09-10.

[2] A. Alliance. Alljoyn. https://allseenalliance.org/.

[3] D. Balasubramanian, A. Dubey, W. Otte, T. Levendovszky, A. Gokhale, P. Kumar, W. Emfinger, and G. Karsai. Drems ml: A wide spectrum architecture design language for distributed computing platforms. *Science of Computer Programming*, 2015.

[4] O. Brown and P. Eremenko. The Value Proposition for Fractionated Space Architectures. AIAA Paper 2006-7506, 2006.

[5] A. Dubey, W. Emfinger, A. Gokhale, G. Karsai, W. Otte, J. Parsons, C. Szabo, A. Coglio, E. Smith, and P. Bose. A Software Platform for Fractionated Spacecraft. In *Proceedings of the IEEE Aerospace Conference, 2012*, pages 1–20, Big Sky, MT, USA, Mar. 2012. IEEE.

[6] M. Eysholdt and H. Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SPLASH*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. . URL http://doi.acm.org/10.1145/1869542.1869625.

[7] P. Feiler, B. A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In *Computer Aided Control System Design*, pages 1206–1211, 2006. .

[8] 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.

[9] T. Genßler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. Müller, and C. Stich. Components for Embedded Software: the PECOS Approach. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 19–26. ACM, 2002.

[10] G. T. Heineman and W. T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70485-4.

[11] 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*, pages 791–798. IEEE, 2008.

[12] R.-T. Innovations. RTI Data Distribution Service. http://www.rti.com/products/dds/index.html.

[13] G. Karsai, D. Balasubramanian, A. Dubey, and W. Otte. Distributed and managed: Research challenges and opportunities of the next generation cyber-physical systems. In *17th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2014, Reno, NV, USA, June 10-12, 2014*, pages 1–8, 2014. . URL http://dx.doi.org/10.1109/ISORC.2014.36.

[14] D. Moore, E. Olson, and A. Huang. Lightweight communications and marshalling for low-latency interprocess communication. 2009.

[15] S. Pradhan, A. Dubey, W. R. Otte, G. Karsai, and A. Gokhale. Towards a product line of heterogeneous distributed applications. *ISIS*, 15:117.

[16] 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*, volume 3, page 5, 2009.

[17] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, and C. Gill. TAO: A Pattern-Oriented Object Request Broker for Distributed Real-time and Embedded Systems. *IEEE Distributed Systems Online*, 3(2), Feb. 2002.

[18] S. Vinoski. Advanced message queuing protocol. *IEEE Internet Computing*, (6):87–89, 2006.

[19] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, and C. D. Gill. QoS-enabled Middleware. In Q. Mahmoud, editor, *Middleware for Communications*, pages 131–162. Wiley and Sons, New York, 2004.