The Journal of Systems and Software 122 (2016) 344–363

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Achieving resilience in distributed software systems via self-reconfiguration

Subhav Pradhan, Abhishek Dubey[*], Tihamer Levendovszky, Pranav Srinivas Kumar, William A. Emfinger, Daniel Balasubramanian, William Otte, Gabor Karsai

Institute for Software Integrated Systems, Department of EECS Vanderbilt University, Nashville, Tennessee, USA

A R T I C L E   I N F O

Article history:
Received 29 October 2014
Revised 14 May 2016
Accepted 24 May 2016
Available online 27 May 2016

Keywords:
Distributed systems
Wireless computing clusters
Resilient systems
Algorithms
Performance evaluation

A B S T R A C T

Improvements in mobile networking combined with the ubiquitous availability and adoption of low-cost development boards have enabled the vision of mobile platforms of Cyber-Physical Systems (CPS), such as fractionated spacecraft and UAV swarms. Computation and communication resources, sensors, and actuators that are shared among different applications characterize these systems. The cyber-physical nature of these systems means that physical environments can affect both the resource availability and software applications that depend on resource availability. While many application development and management challenges associated with such systems have been described in existing literature, resilient operation and execution have received less attention. This paper describes our work on improving runtime support for resilience in mobile CPS, with a special focus on our runtime infrastructure that provides autonomous resilience via self-reconfiguration. We also describe the interplay between this runtime infrastructure and our design-time tools, as the later is used to statically determine the resilience properties of the former. Finally, we present a use case study to demonstrate and evaluate our design-time resilience analysis and runtime self-reconfiguration infrastructure.

1. Introduction

Improvements in mobile networking, combined with the ubiquitous availability and adoption of low-cost embedded platforms, have enabled the vision of mobile cyber-physical platforms, such as a swarm of Unmanned Aerial Vehicles (UAV) and fractionated spacecraft, which is an ad-hoc cluster comprising individual satellite modules. Often these platforms are comprised of multiple systems spanning across physical domains, where each domain is represented by a separate subsystem. Sensors, actuators, computing resources, and communication resources shared among different applications, characterize these systems. These systems are called “cyber-physical” because (a) their mobility and physical environment affect the resources available during their operation, and (b) the software applications often interact with the sensors and actuators to monitor and control physical environment of the system. An example for the latter, in case of a swarm of UAVs, is an instance of a flight management application that uses sensors and actuators like compass, camera, GPS receiver, accelerometer, gyroscope, and propellors to maintain safe flight path for each UAV.

To design and develop applications for these platforms, there exist well-established software engineering techniques such as Component Based Software Engineering (CBSE) (Heineman and Councill, 2001) and Model Driven Engineering (MDE) (Sztipanovits and Karsai, 1997; Schmidt, 2006). CBSE promotes robust composition of complex distributed applications using pre-fabricated and pre-tested software components as building blocks. This increases reuse and reduces product time to market. The MDE approach facilitates modeling various aspects of these platforms using Domain-Specific Modeling Languages (DSMLs). Models created using these DSMLs can be used at design-time to perform tasks such as code generation, analysis and validation. In our previous work (Levendovszky et al., 2014; Balasubramanian et al., 2015), we described the underlying software information architecture and the supporting DSMLs developed for the domain of distributed, real-time cyber-physical platforms.

Broadly speaking, cyber-physical platforms fall into one of two categories: open or closed. Closed platforms do not allow dynamic application provisioning and they host software applications that are geared towards closed-loop, low-latency and real-time

[*] Corresponding author.
E-mail addresses: subhav.m.pradhan@vanderbilt.edu (S. Pradhan), abhishek.dubey@vanderbilt.edu (A. Dubey), tihamer.levendovszky@vanderbilt.edu (T. Levendovszky), pranav.s.kumar@vanderbilt.edu (P.S. Kumar), william.a.emfinger@vanderbilt.edu (W.A. Emfinger), daniel.balasubramanian@vanderbilt.edu (D. Balasubramanian), william.otte@vanderbilt.edu (W. Otte), gabor.karsai@vanderbilt.edu (G. Karsai).

http://dx.doi.org/10.1016/j.jss.2016.05.038
0164-1212/© 2016 Elsevier Inc. All rights reserved.

interconnections. Open platforms, however, allow dynamic application provisioning and require a set of applications that can provide services such as monitoring, tracking, preventive maintenance and logging data for off-line analysis; this paper considers open platforms.

In these cyber-physical platforms, the tight integration between the physical and cyber elements, both within and across different entities, can lead to failure cascades, which in turn can affect the delivery of essential services. Mobile nature of cyber-physical platforms also require mechanisms to handle temporary, intermittent, as well as permanent network connection issues due to fluctuating communication bandwidth. Therefore, *resilience*, which includes efficient techniques for managing the system and ensuring its correct operation within the specified parameters, even in the presence of faults and failures, is crucial. Furthermore, these platforms are remotely deployed; for example, a cluster of UAVs might allow a new UAV to join the existing cluster or an existing UAV to leave the cluster, at anytime. As such, the resilience mechanism should be autonomous due to lack of human interaction opportunity.

In order to support autonomous resilience, we require both design-time as well as runtime support. Appropriate design-time tools should be used to perform static, design-time analysis as admittance test. This is important because open cyber-physical platforms can host multiple applications running simultaneously and it is critical to make sure that addition of new applications does not cause existing applications to fail. Some examples of design-time tools are timing analysis tool, network Quality-of-Service (QoS) analysis tool, and reliability analysis tool. In this paper, we present brief description of our prior work related to timing and network QoS analysis tools. A reliability analysis tool, however, is described in detail.

Runtime support for resilience involves failure monitoring, detection, diagnosis, and mitigation. For autonomous resilience, these actions should be performed as a closed-loop without any external intervention. Significant amount of work related to monitoring, detection and diagnosis is already present in existing literature; as such, this paper mainly focuses on failure mitigation. In order to solve the problem of autonomous failure mitigation, a variety of design-time explicit encoding approaches have been presented in existing literature (Andrade and de Araújo Macêdo, 2009; Asmare et al., 2012; Valls et al., 2013; Schaeffer-Filho et al., 2014; García-Valls et al., 2014). However, they suffer from two main drawbacks: it is time consuming, and explicitly enumerating all possible failure scenarios at design-time is impossible for a mobile, dynamic system.

In comparison, our approach relies on implicit encoding of all possible states a system can reach. We refer to this encoding as a *configuration space* and it consists of relevant information about different system goals, functionalities, services, resources, and constraints. At any given time, there is exactly one *configuration point* that represents the current state of a platform. At runtime, when a configuration point is deemed faulty (due to failure or anomaly), we rely on our runtime self-reconfiguration infrastructure to first compute a valid new configuration point that belongs to the same configuration space, and then transition/migrate/reconfigure to the newly computed configuration point such that failures or anomalies are mitigated.

Our work, presented in this paper, can be roughly divided into two parts (a) design-time analysis and validation tools, and (b) runtime self-reconfiguration infrastructure for autonomous resilience. Key contributions of this paper are listed below.

- A novel design-time reliability analysis tool that provides feedback about the resilience of a system's architecture.
- A novel runtime self-reconfiguration infrastructure that facilitates autonomous resilience via transitions between configuration points computed at runtime using implicitly encoded configuration space.
- A case study to demonstrate and evaluate the design-time resilience analysis and runtime self-reconfiguration mechanism.

It is important to note that our current implementation of the runtime self-reconfiguration infrastructure does not use any of the analysis tools, that we use at design-time, to analyze and validate new configuration points computed at runtime. We believe that, ultimately, design-time analysis tools should also be used at runtime by the self-reconfiguration infrastructure, however, this is part of our future work.

The rest of the paper is organized as follows: Section 2 presents related research work and compares it to our work; Section 3 presents the system model; Section 4 presents a motivating scenario comprising a cluster of fractionated satellites; Section 5 presents the problem statement; Section 6 presents an overview of our solution approach; Section 7 presents our prior work, as well as, a novel contribution related to design-time analysis tools; Section 8 presents our runtime self-reconfiguration infrastructure and algorithms; Section 9 first presents a use case scenario, and then demonstrates and evaluates the design-time resilience analysis tool and runtime self-reconfiguration mechanism; finally, Section 10 provides concluding remarks and describes future work.

2. Related research

We classify the related works along two broad categories - (a) design-time analysis tools, and (b) runtime dynamic reconfiguration mechanisms.

2.1. Design-time analysis tools

Design-time timing analysis for real-time systems is a mature field. In Audsley et al. (1995); Sha et al. (2004), authors address challenges in uniprocessor and multiprocessor scheduling of unique task sets, triggering mechanisms and interactions. Simulation tools like Harbour et al. (2001); Singhoff et al. (2004); Amnell et al. (2004); Derler et al. (2008) are used for various kinds of timing analysis and verification, providing results that feed back into the design for refinement. Component-based design models are typically transformed into a formal analysis model such as Timed Automata (Alur and Dill, 1994; Macariu and Cretu, 2010) or high-level Petri nets (Masri et al., 2009) for which analysis tools exist. Architecture Analysis and Design Language (AADL) models have been translated into high-level Petri nets like Symmetric nets (Renault et al., 2009b) and Timed Petri nets (Renault et al., 2009a) to verify real-time properties like deadline violations. Our prior work, which has been briefly described in Section 7, uses a Colored Petri Net-based (CPN) (Jensen and Kristensen, 2009) analysis model to analyze the structural and behavioral properties. This work was originally presented in Kumar et al. (2014); Kumar and Karsai (2015).

Mobile CPS rely on wireless network communications to coordinate the distributed applications' services. This network communication over dynamic wireless links must provide design-time guarantees about application and system performance. Methods for determining these guarantees arise from simulation, mathematical analysis, or a combination of the two. OMNET++ (Varga and Hornig, 2008) and the INETMANET framework within OMNET++ can be used to simulate network traffic through different network layers and over dynamic wireless links. However, these tools are less useful for providing design-time application performance guarantees. Additionally, large, complex systems increase the complexity of the simulation. Network Calculus, (Le Boudec and Thiran, 2001), focuses on abstracting the application traffic and network links as arrival curves and traffic shapers. Resulting bounds provide design-time guarantees about *worst-case* application performance on the network. The network QoS analysis techniques described briefly in Section 7.1.1 are based on the Network Calculus results, but are designed to provide tighter guarantees on the QoS results at the cost of more precise specification of the system and its applications. This work was originally presented in Emfinger et al. (2014).

Apart from describing timing and network QoS analysis tools, this paper also presents a novel, design-time reliability analysis tool.

2.2. Runtime dynamic reconfiguration

Significant amount of prior work has been done in order to achieve dynamically reconfiguring systems. Asmare et al. (2012); Schaeffer-Filho et al. (2014); Andrade and de Araújo Macêdo (2009) presents different policy-based approaches to achieving dynamic reconfiguration. In Asmare et al. (2012), the authors present a policy-based framework that requires mission specification, which describes how specific roles are assigned to different nodes based on their credentials and capabilities, and how these roles should be re-assigned in response to changes or failures. As such, this mission specification explicitly encodes reconfiguration actions, i.e., role re-assignments, during design-time. Schaeffer-Filho et al. (2014) also follows similar approach where declarative policies are used to specify adaptation. In Andrade and de Araújo Macêdo (2009), the authors present a policy-based approach where each adaptation policy comprises rules, actions, and the rate at which each rule should be evaluated. These approaches are different from ours, as we do not explicitly encode reconfiguration actions at design-time; it is impossible to cover all possible combinations of failure scenarios at design-time.

Alternative approaches to achieving dynamic reconfiguration include use of system health management techniques (Srivastava and Schumann, 2011). Our prior work that follows this approach includes (Mahadevan et al., 2011a), which shows how system-wide mitigation can be performed based on reactive timed state machines specified at design-time, using the results of a two-level fault-diagnoser (Dubey et al., 2011a). Thereafter, we presented a boolean encoding for reconfiguring a system using a search based strategy in Mahadevan et al. (2013). These approaches have similar limitations to aforementioned policy-based approaches since the runtime reconfiguration mechanism depends on static design time specifications.

In Valls et al. (2013); García-Valls et al. (2014), the authors present a middleware that supports timely reconfiguration in distributed real-time systems. *Application Graph*, which contains information about what services are required and how they depend on each other, and *Expanded Graph*, which contains information about different service implementations, are studied *a priori* at design-time. As such, these solutions also have similar limitations to aforementioned solution since runtime reconfiguration mechanism relies on artifacts computed at design-time.

In Arshad et al. (2007), the authors present a tool called *Planit* for deployment and reconfiguration of component-based applications. Planit uses a temporal planner and is based on a *sense-plan-act* model for fault detection, diagnosis, and reconfiguration to recover from runtime application failures. As such, it is similar to our work presented in this paper. In order to facilitate the runtime planning, Planit allows modeling of both, implicit and explicit configurations at design-time. Although our reconfiguration mechanism is also based on implicitly encoded configuration (we call this configuration space), we capture this encoding in a very generic manner. To be precise, we use a goal-based system description approach to ensure loose coupling between requirements and actual software entities that fulfill those requirement. However, in Planit, implicit encoding is defined in terms of low-level software artifacts such as components and their connections. We believe that a solution for mobile systems needs to provide better flexibility to account for dynamism.

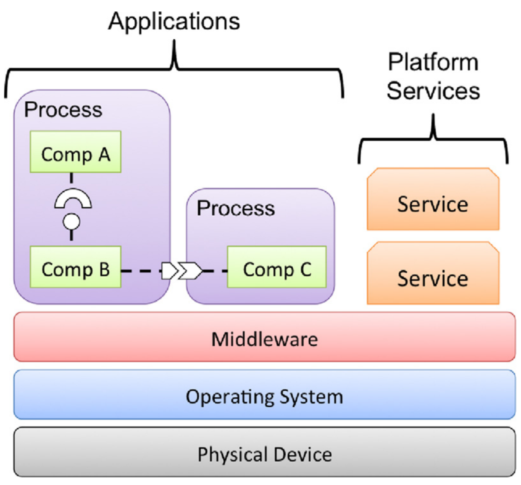


Fig. 1. System model.

3. System model

The target system consists of clusters of remotely deployed, heterogeneous nodes with computation and communication resources, as well as, a variety of sensors and actuators. As shown in Fig. 1, each device hosts a layered software architecture consisting of an operating system (OS), middleware, applications and platform services. Since we are considering heterogeneous systems, different nodes could consist of varying OS and middleware. However, solving challenges arising from these heterogeneities are out of scope of this paper, and is part of our ongoing research efforts.

Platform services are, in essence, long running services that serve as an extension of the OS by providing generic services for applications to use. Examples of platform services include monitoring services and distributed application management service. As shown in Fig. 1, applications consists of software components. These components are hosted inside processes. Each process can host one or more components. Processes are created, deployed, configured, and managed by a specific platform service that is responsible for application management. Readers are encouraged to refer to Karsai et al. (2014) to review the system architecture in detail.

3.1. Goal-based system description

Mobile CPS are dynamic in nature and therefore require a generic way to represent system goals expected to be satisfied by a system during a given time interval. A time interval sequence consisting of high-level system objectives that must be available during those intervals is called a mission *goal*. Different systems associated with a cyber-physical platform are mission oriented and therefore have specific mission goals. Since objectives are essentially functions, system *objectives* can be defined using the concept of functional decomposition, which is the process of decomposing high-level functions into a set of sub-functions, until a set of leaf-level functions is reached. Leaf-level functions are functions that cannot be decomposed, and they are mapped to components (Kurtoglu et al., 2010; Mahadevan et al., 2013) that

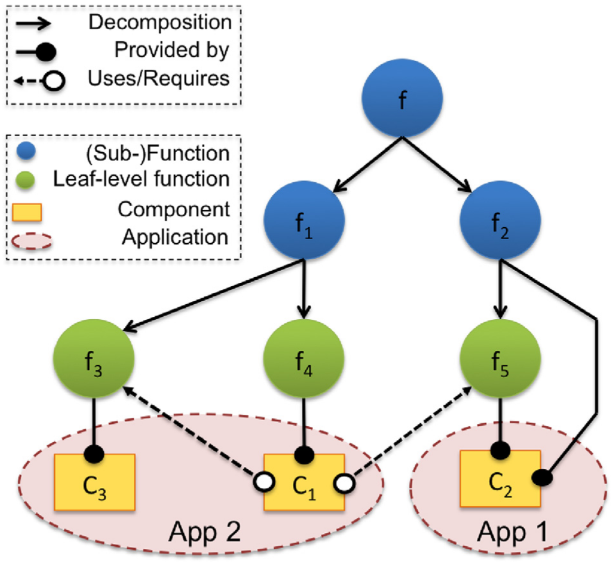


Fig. 2. Functional decomposition graph for a two-application system.

provide these functions as *services*. Services are provided or required by components through their ports.

Components are the basic unit of system composition. As shown in Fig. 2, a component can provide one or more functions (leaf-level or non leaf-level) via its ports. Furthermore, different components can provide the same functionality. If a functionality is provided by multiple components, then any of those component can be deployed; this allows more flexibility. Similarly, a component can also require one or more functions via its ports. This provided and required relations between components and functions allow us to establish dependencies between components. In addition to functions, a component can also require resources in order to be considered available. As such, we classify a component's requirement into *functional requirements* and *resource requirements*.

Fig. 2 presents the functional decomposition graph of a simple two-application system. As shown in the figure, function f is a high-level function that represents a system's objective. Function f can be decomposed into sub-functions f_1 and f_2. Sub-function f_1 can be further decomposed into leaf-level functions f_3 provided by component C_3 and f_4 provided by component C_1. Similarly, sub-function f_2 can be decomposed into leaf-level function f_5 provided by component C_2.

Definition 1. A functional decomposition is a directed acyclic graph (DAG), $FD = (F, DE)$, where F is the set of functions, and $DE = F \times F$ is the set of dependency edges. The functions with zero indegree are called functions while the others are called sub-functions.

Definition 2. A software component is a collection, $C = (P, S, R)$, where P is a set of ports associated with a component, S is a set of functions provided by a component, and R is a set of requirements.

Definition 3. An application is a graph of components, $G = (C, E)$, where $E \subseteq C \times C$ represent the control/data flow dependencies between components. These dependencies impose operational requirements on components. That is, unless specified otherwise, a component requires all other components to which it is connected to be available.

3.2. Resource model

The physical computing infrastructure provides *computation* and *communication* resources. Computation resources correspond to hardware facilities required to execute computation tasks at a given computation node. These include processing speed (number of instructions per second), memory size (amount of memory required), and specific hardware required for certain tasks such as sensing or signal processing. Communication resources on the other hand correspond to facilities required for interaction between tasks executing on different computation nodes. This includes communication bandwidth, available security measures such as encryption, etc.

The resource model represents the capabilities and evolution of resources that are used to carry out a mission, as a function of control inputs (actions) applied to the resources. For example, the resource specification for a network link would include capability specifications like the maximum data flow capacity of the link. It would also include the possible discrete states of the link, such as whether it is in service or broken.

3.3. Fault model

A fault is defined as a problem within a system entity that can manifest itself in observable discrepancies: deviations from expected behavior; or it can remain unobservable. A fault may cause a failure. The failure of a system or a component is the breakdown of its capability to provide required services or functions. The interconnections between system entities imply that a failure of one entity can also lead to a secondary failure in a connected entity. If the failure propagates to the global level, i.e. the top-level system, it is called a global failure. In a "system of systems", fault-tolerance algorithms are required to detect faults, mask fault effects, and mask lower-level component failures so that they do not lead to a global failure. To be considered fault tolerant, a system must be able to detect occurrences of discrepancies that signify faults, to diagnose and isolate the probable fault sources, to take actions to either contain the faults (and thus stop them from propagating outwards), and/or mitigate their effects on system functions.

We classify failures into two categories - (a) infrastructure, and (b) application. Infrastructure failures are failures that arise due to faults affecting a system's network, participating nodes, devices hosted on different nodes, or processes running on different nodes. There exist causality between these four different kinds of infrastructure failures. A network failure causes all nodes that are part of the network to fail since those nodes become unreachable after their network failure. A node failure causes all the devices and processes running on that node to fail. A device failure might cause processes using that device to fail, it might even cause the entire node that hosts the device to fail, or if the device is a networking device then it might cause network failure. However, a process can fail without its host node failing or one of the devices it uses failing. Similarly, a device can fail without its host node failing, and a node can fail due to reasons other than network separation or device failure. We consider infrastructure failures to be *primary failures* that can result in application failures, ultimately causing the system to lose existing functions.

Application failures are failures pertaining to the application components. We assume that application components have been thoroughly tested before deployment and therefore classify application failures as *secondary failures* that are caused by to infrastructure failures. However, there can be scenarios where an application component failure becomes a primary source of failure and results in its hosting process, i.e., infrastructure to fail. In this case, application failure becomes a primary failure. Some environmental changes could also lead to application failures, where the changes in the environment can cause an application to receive unexpected input or the environment might not react, as expected, to an application's output.

Failures can be temporary, intermittent or permanent. Temporary failures are failures that have a short duration, while intermittent failures are temporary failures that occur at irregular intervals. Currently, our work focuses on permanent failures but we intend to handle both temporary and intermittent failures in our future work. In case of temporary failures, if we assume a *fail-stop* model, we can treat them like permanent failures. For example, when a node fails temporarily due to network partition, all of its hosted entities are considered failed and appropriate reconfiguration actions will be taken. However, because the failure is temporary, the node comes back online after some time, at which point it can be treated as a brand new node joining an existing cluster. However, applications that are still running on the node that suffered temporary failure must be removed before using its resources. A similar approach can be taken for intermittent failures as well.

3.4. Deployment model

Deployment means instantiating a set of components and mapping them to available physical resources. Given a set of currently deployed and active applications and a set of components included in the application, we can deduce the set of system functions that can be supported.

Definition 4. A deployment $D = (d_V)$ is a function that maps software structure *SC*, which is a set of component instances and their inter-dependencies, to a hardware network $HC = (N, L)$, which is a DAG, where N is the set of nodes, and $L = N \rightarrow N$ is the set of links between these nodes. A communication link resource function is a function $N \times L \Rightarrow \mathbb{N}$ that represents the capacity of a specific communication link on a node. d_V: $C \rightarrow N$, where C is a set of components.

3.4.1. Alternate deployment configurations

Two deployment configurations are considered to be alternatives if they deploy the same set of applications on the same physical architecture within the same resource constraints while satisfying the same set of goals. Alternate deployment configurations could be compared against each other on the basis of resource cost and performance.

3.4.2. Configuration space and configuration point

The configuration space of a platform represents the state space of the platform. This includes (a) goal-based system description of different systems hosted on the platform, (b) resource requirements of different components that are part of the system description, (c) nodes that comprises the platform and their corresponding resources, such as memory, storage, and devices, and (d) deployment constraints that determine whether components should be collocated on the same node, distributed across different nodes, or always deployed on a specific node. A configuration space can expand or shrink depending upon addition or removal of related entities.

A configuration space can contain multiple configuration points. A configuration point represents valid configurations of all systems that are part of the associated configuration space. A valid configuration of a system represents component to node mappings (deployment) for all components that are required to satisfy the system's goal. We always begin with a valid configuration point, which we call the initial configuration point. Initial configuration point represents the initial deployment of different systems. Similarly, current configuration point represents the current deployment. A configuration point, since it is a component-to-node mapping, can be represented using a component-to-node matrix defined below.

Definition 5. The component to node mapping can be represented using a component-to-node (C2N) matrix, where rows represent component and columns represent available computing nodes. A separate table is used to map the index of a component or a node instance to its name. The size of this matrix is $\alpha \times \beta$, where α is number of component and β is the number of available computing nodes. Each element of this matrix is an integer variable that can either be 0 or 1, where 0 indicates that the component (row) is not deployed on the corresponding node (column), and 1 indicates otherwise.

$$C2N = \begin{bmatrix} c2n_{00} & c2n_{01} & c2n_{02} & \dots & c2n_{0\beta} \\ c2n_{10} & c2n_{11} & c2n_{12} & \dots & c2n_{1\beta} \\ c2n_{20} & c2n_{21} & c2n_{22} & \dots & c2n_{2\beta} \\ \dots & \dots & \dots & \dots & \dots \\ c2n_{\alpha 0} & c2n_{\alpha 1} & c2n_{\alpha 2} & \dots & c2n_{\alpha\beta} \end{bmatrix}$$

$$C2N = (c2n_{cn} : c \in \{0 \dots \alpha\}, n \in \{0 \dots \beta\}, (\alpha, \beta) \in \mathbb{Z}^+)$$

At any given point in time only one of the configuration point reflects the reality of the deployed system; this is the current configuration point. All other configuration points are implicitly present in the configuration space, but have to be computed dynamically at runtime. This is precisely what happens when a failure is detected. When a failure occurs, current configuration point is marked as faulty, specifically some component(s) or node(s) are marked as faulty, rendering corresponding row(s) or column(s) of the the *C2N* matrix with 0 markings and a constraint that it cannot be used in future unless the fault has been removed. For example, consider a scenario where multiple configuration points maps one or more components to a node. If this node fails, then all aforementioned configuration points are rendered faulty. Given these concepts of configuration space and points, recovering from failure essentially involves self-reconfiguration of the system by finding a new valid configuration point and determining actions required to transition from current (faulty) configuration point to the new (desired) configuration point. As such, configuration points and their transitions form the very core of our self-reconfiguration mechanism.

For a more detailed description of a configuration space and its configuration points, please refer to our previous work (Pradhan et al., 2015), which presents a feature model that we use to represent a configuration space.

4. Motivating scenario

Consider a mobile cyber-physical 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 satellite cluster, can come from different organization. This architecture can realize the functions of monolithic satellites at a reduced cost and with improved adaptability and robustness (Brown and Eremenko, 2006). 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 foundation for applications running simultaneously using shared resources.

Individual satellite modules of a fractionated satellite 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 objective by periodically using their thrusters to adjust their position. In addition to this critical objective, other objectives can be added by hosting different applications. Fig. 3 presents an overview of

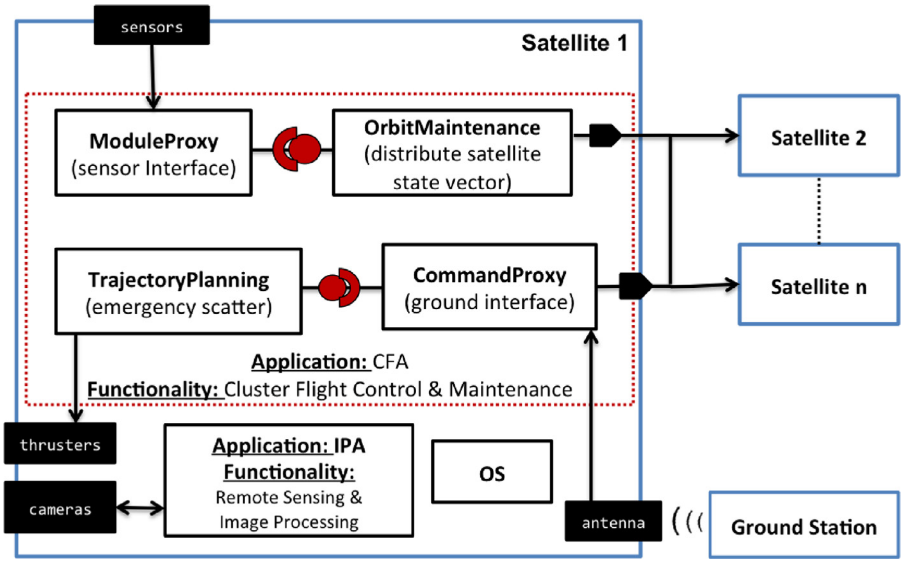
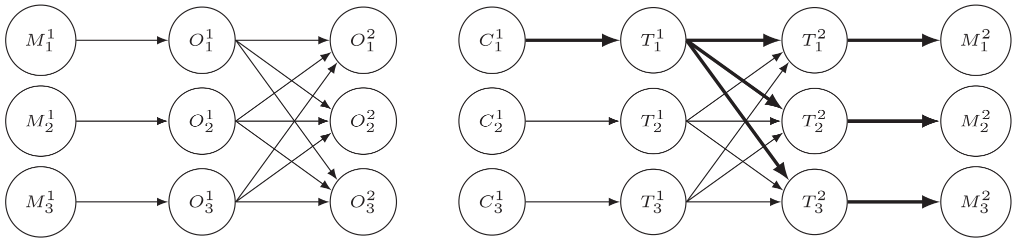


Fig. 3. Mixed-criticality distributed deployment.



Task	Component	Activity
M^1	*ModuleProxy*	Inform O^1 of new state
M^2		Activate engine
O^1	*OrbitalMaintenance*	Publish new state
O^2		Subscribe to new state
T^1	*TrajectoryPlanning*	Publish new command
T^2		Subscribe to new command
C^1	*CommandProxy*	Inform T^1 of command

Fig. 4. Tasks performed by components of the *cluster flight application*. For these tasks, the subscript represents the ID of the node onto which a task is deployed. The total latency of the interaction $C_1^1 \rightarrow M_N^2$ represents the total latency between receiving the *scatter* command and activating the thrusters. This interaction pathway is in bold.

a fractionated satellite cluster hosting two different component-based applications with mixed criticality. The first application is a high-priority *Cluster Flight Application* (CFA), which is responsible for maintaining flight control. The second application is lower priority *Image Processing Application* (IPA), which is responsible for capturing real-time images and processing them.

As shown in Fig. 3, the high-priority CFA application comprises four different components. Next, we briefly describe the different functions provided by these components. A schematic overview of the associated tasks performed by these components is presented in Fig. 4.

- *ModuleProxy*: This component behaves as an interface between different satellite sensors and the *OrbitMaintenance* component, allowing the *OrbitMaintenance* component to access available sensors.
- OrbitMaintenance: This component is responsible for tracking the state of a cluster satellite. To perform this task, it uses the *ModuleProxy* component to acquire the latest information, such as location co-ordinates. Once appropriate information is collected, this component is also responsible for disseminating this information as a packaged structure to all other satellites in the cluster. As every satellite runs an instance of CFA, each node periodically receives updates from the other nodes.
- CommandProxy: This component performs the task of receiving commands from a ground station. When a command is received, it sends the command to its local *TrajectoryPlanning* component. Furthermore, commands received from a ground station are also forwarded to other satellite nodes in a cluster.
- TrajectoryPlanning: This component is responsible for performing the task of receiving commands from the local *CommandProxy* component and responding to those commands using satellite thrusters, if required, to perform highly critical, hard real-time tasks.

The lower priority IPA is a comparatively simpler application, which comprises a component that uses the camera to capture real-time images (sensing) and another component that processes the captured images. These are periodic CPU-intensive tasks that are temporally isolated from each other. As mentioned before, the

IPA is a lower priority application when compared to the CFA. As such, the IPA tasks are executed by application threads that have lower priority than that of the CFA.

5. Problem statement

A mobile cyber-physical platform can host numerous mission critical cyber-physical applications. Each application consists of components providing different functions to meet various objectives and therefore the mission goal. As such, it is of utmost importance to make sure that all functions and their corresponding components required to maintain a system's goal are preserved in the face of failures and anomalies. Therefore, for cyber-physical platforms, such as the fractionated satellite cluster described as a motivating scenario in Section 4, resilience is a key requirement. We adopt the definition of resilience from Laprie (2008): "The persistence of the avoidance of failures that are unacceptably frequent or severe, when facing changes." Although a truly resilient system needs to be resilient against failures, changes (intended or unintended), and updates, in this paper we only focus on resilience against failures. In addition to hosting mission critical applications, mobile cyber-physical platforms are remotely deployed and therefore require the resilience mechanism to be autonomous.

We identify the following as requirements that need to be satisfied in order to achieve cyber-physical platforms that are capable of supporting autonomous resilience:

Requirement 1 - Design-time analysis tools for admittance checking: Application requirements, such as timing and network QoS requirements, and properties like resilience should be analyzed and validated at design-time.

Requirement 2 - Runtime mechanism to facilitate autonomous resilience: We require a distributed runtime mechanism capable of providing autonomous resilience. Any such mechanism should be able to monitor, detect, diagnose, and mitigate failures.

Multiple solutions related to *Requirement 1* have been published as part of our prior work; we briefly describe these in this paper. Furthermore, as a minor contribution, we present a novel design-time tool capable of performing resilience analysis. Most of the work presented in this paper focuses on *Requirement 2*. There exists significant amount of research literature related to failure monitoring, detection, diagnosis, and mitigation. Most of this can be leveraged for mobile cyber-physical platforms, however, existing solutions for failure mitigation cannot be used as they do not take into account the scale and the dynamic nature of these platforms. As such, our contribution in this paper is a novel self-reconfiguration mechanism that can be used by mobile cyber-physical platforms to facilitate autonomous resilience.

6. Solution approach overview

An overview of our solution approach is shown in Fig. 5; it comprises design-time and runtime aspects. The design-time aspect of the solution includes a graphical modeling tool developed using the Generic Modeling Environment (GME) (Ledeczi et al., 2001), associated model interpreters, a set of design-time analysis tools (described in Section 7), and a database to store artifacts generated and analyzed by aforementioned interpreters and analysis tools. The database is also part of the runtime aspect. We can view this database as a medium through which relevant information is shared between entities of the design-time and runtime entities. In addition to the database, the runtime aspect also includes a management infrastructure, a managed system, a monitoring infrastructure, and a resilience infrastructure. These runtime entities form an autonomous resilience loop akin to a *sense-plan-act* loop, which is the basis of our approach to realizing a self-reconfiguring system.

The monitoring infrastructure performs the task of sensing; it is responsible for monitoring a managed system to detect and diagnose failures. The resilience infrastructure performs the task of planning; it is responsible for covering the self-reconfiguration mechanism. Finally, the management infrastructure performs the task of acting; it is responsible for undertaking actions computed (planned) by the resilience infrastructure. In our implementation, which is described in detail in Section 8, the monitoring infrastructure comprises distributed monitors for failure detection, the resilience infrastructure comprises a Satisfiability Modulo Theories (SMT) (Barrett et al., 2009) based solver, and the management infrastructure comprises distributed Deployment Managers (DMs), where a single DM is deployed on every node.

A typical workflow is as follows. The user begins by creating a model and defining components, which provide the basic units of functionality. The definition of a component includes its communication ports and timing requirements. Next, one or more applications are created by assembling components together and configuring their communication. For instance, the output port of one component may be connected to the input port of another

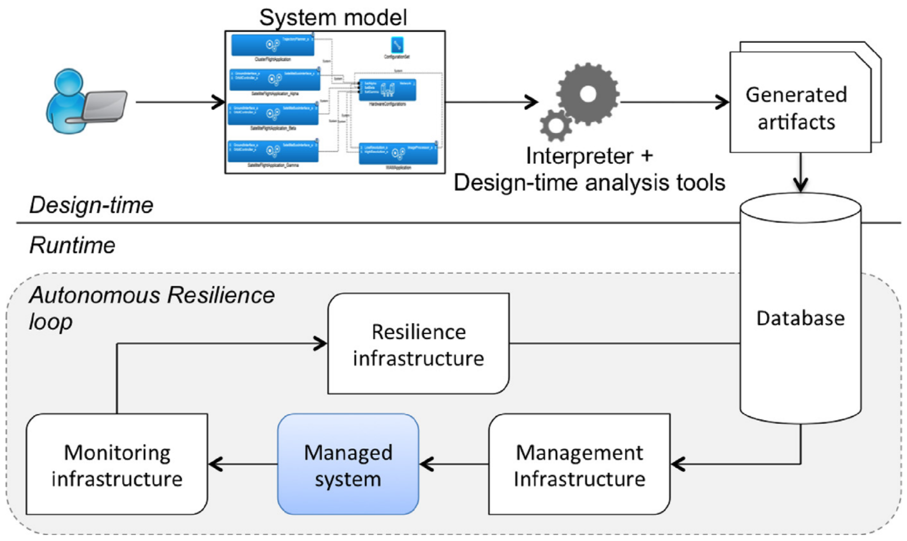

System model
Generated artifacts
Interpreter + Design-time analysis tools
Design-time
Runtime
Autonomous Resilience loop
Resilience infrastructure
Database
Monitoring infrastructure
Managed system
Management Infrastructure

Fig. 5. Overview of the approach.

component. Based on applications that need to be deployed on the target platform, mission goals (see Section 3.1) are defined. At this point design-time analysis tools and model interpreters are used to analyze the design-time model and generate appropriate artifacts that represent a configuration space and initial configuration point (see Section 3.4.2) for the modeled system. These are stored in the database and is later used at runtime; the initial configuration point is used for initial deployment and the configuration space is used to compute new configuration points at runtime in order to reconfigure the system. We describe this process in detail in Section 8.2.

7. Design-time analysis and validation tools

In this section we present three different design-time analysis tools. Among these three tools, the first two are related to timing and network QoS analysis. These two tools are described briefly as they are not the main contribution of our work presented in this paper; they are part of our prior work but discussed here in order to present a bigger picture of how different design-time analysis tools can be used for analyzing and verifying different aspects of a mobile, dynamic distributed system. The third tool we present in this section is a novel resilience analysis tool, which we describe in detail.

7.1. Overview of CPN-based timing analysis

Real-time systems are characterized by strict deadlines. Delayed response times and missed deadlines can have a catastrophic effect on the health of the system, especially in the case of safety and mission-critical scenarios. Many of these scenarios also execute software in remote environmental conditions where quick access to the software is difficult. Therefore, it is imperative that the temporal behavior of any configuration that can be deployed at runtime, is sufficiently tested and analyzed at design-time.

Our work, which is presented in detail in Kumar et al. (2014); Kumar and Karsai (2015), uses a Colored Petri Net-based (CPN) (Jensen and Kristensen, 2009) analysis model to analyze the structural and behavioral properties of the deployed configuration in order to verify system properties such as lack of deadlocks and timing violations. Such behavioral properties not only improve the confidence in the deployed configuration but it can also be used as a deciding metric when choosing a reconfiguration plan.

To analyze a set of interacting components (both spatially and temporally), we capture the structural and behavioral properties of the system with a domain-specific model. The structural model includes information about (a) component interfaces, ports, timers, (b) application assembly and wiring, and (c) software deployment aspects. These attributes are later parsed and mapped into colored *tokens* in appropriate *places* of the CPN model. The behavioral model encapsulates the sequence of events or *steps* that are executed in every component operation. In our timing model, each operational step is accompanied by a worst-case execution time obtained via prototypical testing. These steps directly affect the state of the executing thread (blocking or unblocking effects) and influence the behavior of the system. The CPN model is equipped with lists of tokens that capture these behavioral properties succinctly.

Analyzing the behavioral properties of the modeled system involves exploring its bounded state space. Here, the initial state represents the point in time right after successful deployment and configuration of application components but right before the first operational trigger. Each state space node is a list of tokens describing the state of each place in the model. By using standard state space queries and graph searching techniques, the system can be evaluated for both qualitative and quantitative measures of health. For simulation and analysis of our CPN models, we use *CPN Tools* (Ratzer et al., 2003). Different system-level properties can be verified using this methodology e.g. absence of deadline-violations, absence of deadlocks or livelocks, satisfactory worst-case trigger to response times, and estimated processor utilization.

7.2. Overview of network QoS analysis

At design time, component developers supply their component implementation models with information about the implementations' network QoS requirements. These requirements contain information about the component's produced network bandwidth as a function of time, the maximum size afforded to the buffer, and the maximum tolerable network buffering delay.

We have developed a paradigm for modeling the components' and system's network QoS requirements which is similar to Network Calculus' traffic arrival curves and traffic shaper curves, (Le Boudec and Thiran, 2001). Using each component implementation's network QoS constraints, together with the network service characteristics provided by the system, the feasibility of the modeled application and system deployment can be determined. The deployment is only feasible if the component's constraints are met by the node for all components on all nodes of the system.

Component profiles model how components' traffic generation changes with respect to discrete time, and system profiles model how the network bandwidth (e.g. bits per second) between nodes of the cluster varies with respect to discrete time. These deterministic models describe exactly the data generated by the component and the data which could be sent through the network. By time-integrating the component and system profiles, the data generation and data throughput as functions of time can be determined for components and the system, respectively. Convolving these profiles results in a profile describing the data actually transmitted on the network link.

The network analysis techniques described above can be applied to different component-based applications that need to be hosted on a mobile cyber-physical platform. The results of this analysis inform us whether or not the applications and services they provide can execute reliably. Consider the motivating scenario of a fractionated satellite cluster presented in Section 4. In this example, the network profiles for the resources provided by the system's network links follow periodic patterns since they are governed by the orbital mechanics of the satellites in the cluster and because of which the distances between the satellites varies periodically as a function of time. Similarly, many applications on the cluster, such as the mission- and safety-critical cluster flight software are also periodic in nature, as they periodically retrieve sensor data and disseminate those data to the rest of the nodes in the cluster. Since these are the most critical tasks in the system, they will preempt all other application and system tasks. As such, we can analyze their network characteristics using these techniques to ensure that they are able to meet their latency and memory constraints throughout the duration of each orbit. The remaining network capacity will be used by other applications, and can be iteratively analyzed according to their priority-based network resource sharing.

For detailed description of our network QoS analysis tool, we refer the reader to our previous work (Emfinger et al., 2014).

7.3. Resilience analysis

In addition to timing analysis and network QoS analysis, resilience analysis is another design-time analysis that is useful in context of mobile cyber-physical platform for which resilience is of utmost importance. In general, the task of a resilience analysis tool is to provide feedback to the system integrator about how resilient a system design is, before the system is actually deployed.

Table 1
Constraint primitives

Primitive	Description
Assign(i, j)	**Input:** a component instance and a node. **Effect:** a constraint that assigns component *i* to node *j*
TurnToBinaryResource(c, n, c_list)	**Input:** a component instance, a node, and a set of components that are using the resource. **Effect:** a constraint that assigns component c to node n, and collocates all the client components present in c_list with c.
Enabled(i)	**Input:** a component instance. **Effect:** a Boolean expression that is true if the component is assigned to a node; false otherwise. The constraint sums the row of the component instance and checks if it is greater than zero.
CollocateComponents(c1, c2)	**Input**: two component instances. **Effect:** a constraint that ensures that the component instances must be assigned to the same node.
DistributeComponents(c_list)	**Input:** a list of component instances. **Effect:** a constraint that ensures that the component instances must be assigned to different nodes.
Communicates(i,j)	**Input:** two component instances. **Effect**: a constraint that makes sure that there is a link between the nodes,the components are deployed on. If the,two components are on the same node, the constraint is still satisfied.
ForceExactly(f, c_list, n)	**Input:** a function and a list of components; a positive integer n. **Effect:** a constraints that makes sure that exactly n of the components in the list must be enabled, to provide the function.
ForceAtleast(f, c_list, n)	**Input:** a function and a list of components; a positive integer n. **Effect:** a constraints that makes sure that at least n of the components in the list must be enabled to provide the function.
ForceAtmost(f, c_list, n)	**Input:** a function and a list of components; a positive integer n. **Effect:** a constraints that makes sure that at most n of the components in the list must be enabled to provide the function.

Our resilience analysis tool calculates two fundamental resilience metrics as a pair of integers: a lower bound and an upper bound on the degree of resilience. When these bounds are calculated, all possible remedial actions are considered. The lower bound measures the least number of faults that can (but not necessarily) lead to a complete system failure. The upper bound is the maximum number of faults the system can possibly tolerate due to the remedial actions of the reconfiguration engine; a higher number of faults will lead to a system failure, regardless of redundancy.

In simple terms, the upper bound is an optimistic resilience metric and the lower bound is a pessimistic resilience metric. If we are designing a safety critical system, the system designer will evaluate design choices based on the pessimistic criteria. However, for a regular enterprise system, a number within the two bounds will be used. It can be argued that the system requires a larger degree of redundancy around critical components to achieve a larger lower bound, increasing the overall cost of the system. Below we provide a formal definition of these two resilient metrics.

Definition 6. A reliability block diagram *RBD*(*Src, Snk, C, N, Dep*) is a directed graph whose nodes are the system components or source/sink nodes ($Src \cup Snk \cup C \cup N$). An edge between a node A and a node B means that B depends on A. *Dep*: $Src \cup C \cup N \times Snk \cup C \cup N$.

Definition 7. The worst-case resilience is defined as the least number of failures that will render one or more system goals unachievable. Alternatively, the worst-case resilience is the number of the node disjoint paths in the RBD.

Definition 8. The best-case resilience is defined as the maximum number of failures that can be sustained while the system goals are met. Alternatively, the best-case resilience is the number of parallel paths in the RBD.

At design-time, we compute these metrics for a system design. We consider a system design as a pair: (a) an initial design i.e. the initial configuration point and (b) the configuration space. While the initial design describes the initial state of the system without any failure, the configuration space implicitly describes all the feasible designs. If a primary fault causes secondary faults of other system entities, it is captured by the logical constraints. For example, to enforce that the failure of a component brings down its host process, we can add a constraint that enforces that. However, our current implementation only considers node and component failures.

In order to perform resilience metrics computation, we formulate the problem as a SMT problem, and use the Z3 solver (de˜Moura and Bjørner, 2008) to solve the SMT problem. We describe the deployment as an adjacency matrix (see Definition 5). The constraints are defined in Table 1. These primitives are translated to equations over the adjacency matrix. For example, the primitive *Enabled* and *Assign* are mapped as shown in Listing 1. The *Enabled* function returns true if a component with index i is assigned to any node. To do this it checks if the sum of the row corresponding to that component is greater than 0. The *Assign* function ensures a component is assigned to a particular node. The assignment is valid only if the component is not in faulty state. Thus, the component being enabled implies the assignment, which means that the element in the component's row and the node's column must be one. The variable *c2n* represents the adjacency matrix in Z3, and *Implies* is the Python wrapper around the implication in the Z3 library.

Once the components and the constraints have been generated, the solver is able to compute solutions. As shown in Algorithms 1 and 2, we find the worst case resilience metric by performing a breadth-first search in the search space of injected faults. We inject faults in the solver by specifying constraints, typically disabling one or more components or nodes. The second phase of the algorithm is the recursive Breadth-First Search (BFS). Essentially, we increase the number of faults each round in which we call the BFS. The BFS combines together all the variations of faults, and when the level counter reaches zero, the current state is evaluated by making all the components/nodes in the list fail and checking if there is a solution. We save the solver state before this computation (*push*) and restore it afterward on each control path (*pop*). Since the order is provided by a BFS, whenever we cannot find a solution, we have found a minimal number of faults. This is our worst case faults.

The computation of the best-case metric is rather indirect. Instead of finding the maximum number of faults, we find a minimal configuration, and then subtract its number of elements from the maximum number of elements in the initial model. This makes our computation much more efficient because we can phrase this problem as a set of constraints for Z3. We express the number of nodes and components as an integer variable *n* for the solver, and we keep calling the solver by requesting a solution with a smaller

```
# num_comp: Number of components.
# num_nodes: Number of nodes.
# c2n: A num_comp X num_nodes adjacency matrix, where values
#       are 0 or 1. 0 meaning not enabled and 1 meaning enabled.
# i: Index of a component to enable.
def Enabled(self, i):
    return Sum([self.c2n[i][j] for j in range(self.num_nodes)])>0

# i: Index of a component to enable.
# j: Index of a node on which the component should be enabled
def Assign(self, i, j):
    return Implies(self.Enabled(i), self.c2n[i][j] ==1)
```

Listing 1. Sample implementation of primitive constraints *enabled* and *assign*.

Algorithm 1 Worst Case Metric Computation — Phase 1: Calling BFS

INPUT: Adjacency Matrix and Constraints
OUTPUT: Worst case metric

```
1: min = 0
2: for e 0 to $|nodes \bigcup components| - 1$ do
3:     res = $get\_min\_faults\_bfs\_r(0, [], e)$
4:     if res == true then
5:         return
```

Algorithm 2 Worst Case Metric Computation — Phase 2: BFS Traversal — *get_min_faults_bfs_r*

INPUT: start, list, level
OUTPUT: Worst case metric

```
1: if level == 0 then
2:     solver.push()
3:     for each element e in list do
4:         fail e
5:     solver.check()
6:     if no solution then
7:         save min as metric
8:         solver.pop()
9:         return true
10:    solver.pop()
11:    return false
12: for e in range of start and $|nodes \bigcup components| - 1$ do
13:    res = $get\_min\_faults\_bfs\_r(n + 1, list + [e], level - 1)$
14:    if res == true then
15:        return true
16: return false
```

n. If the solver cannot find a solution, the previous solution is the smallest model.

We have introduced several metrics, such as, a weighed metric $w = WM$, where $M = (m_{worst}, m_{best})$, and W is an appropriate weighing vector. We extended the notion of this metric for subsystems. The vector M for the system can be expressed as $(min(m^i_{worst}), sum(m^i_{best}))$. The first element takes the minimum of m^i_{worst} for all critical subsystems i. The second element adds all the m^i_{best}. Based on the distance in Definition 9, it is possible to assign distance metric to the worst and best case metrics. If the distance is weighed, we need to modify the BFS algorithm to Dijkstra's path finding algorithm to compute the worst case, and instead of the number of nodes, the distance must be expressed for the solver for the best case metric.

8. Runtime infrastructure for self-reconfiguration

This section presents our runtime self-reconfiguration infrastructure. First, we provide an architectural overview of our distributed infrastructure. Then, we present the reconfiguration mechanism.

8.1. Architecture overview

Fig. 6 presents an overview of our resilient reconfiguration infrastructure. There are two kinds of nodes - a computation node, and a solver node. Each computation node hosts (a) applications, (b) an instance of the distributed database to store the configuration space, the initial configuration point, the current configuration point, and deployment actions, (c) a Deployment Manager (DM) that is responsible for managing lifecycle of different applications, and (d) a monitor to detect failures. There is a notion of a *leader* among different computation nodes and we use the existing capabilities of the distributed database to determine a leader.Unlike a computation node, a solver node only hosts an instance of the distributed database and a Resilience Engine (RE) that can compute a new configuration point when a system needs to migrate from one configuration point to another to mitigate failures. Further descriptions of these are provided below.

***Applications* :** As mentioned in Section 3, *applications* hosted by our system are component-based. Therefore, each application is a set of components that interact with each other or with components of different applications. All information required to deploy and configure these applications on the target system is stored in a distributed persistent backend.

***Distributed database* :** A *distributed database* is required to store relevant information such as (a) the configuration space, (b) the current configuration point, and (c) the initial configuration point. The initial configuration point is provided by a system architect and is used as the baseline for further reconfiguration when required. Well known faults are also stored in the database as part of the configuration space. We use MongoDB (MongoDB Incorporated, 2009) as our choice of database and deploy multiple instances of this in a replica set.

***Monitor* :** A *monitor* is responsible for monitoring, detecting and diagnosing failures. This is an important aspect of a resilient system. However, there exists significant amount of existing work in this particular research area, including our prior work presented in Mahadevan et al. (2011b); Dubey et al. (2011b); Mehrotra et al. (2012). As such, our work, presented in this paper, does not focus on failure monitoring, detection and diagnosis. For our experiments presented in Section 9, we simply inject failures by updating system configuration stored in the database. In essence, monitors in

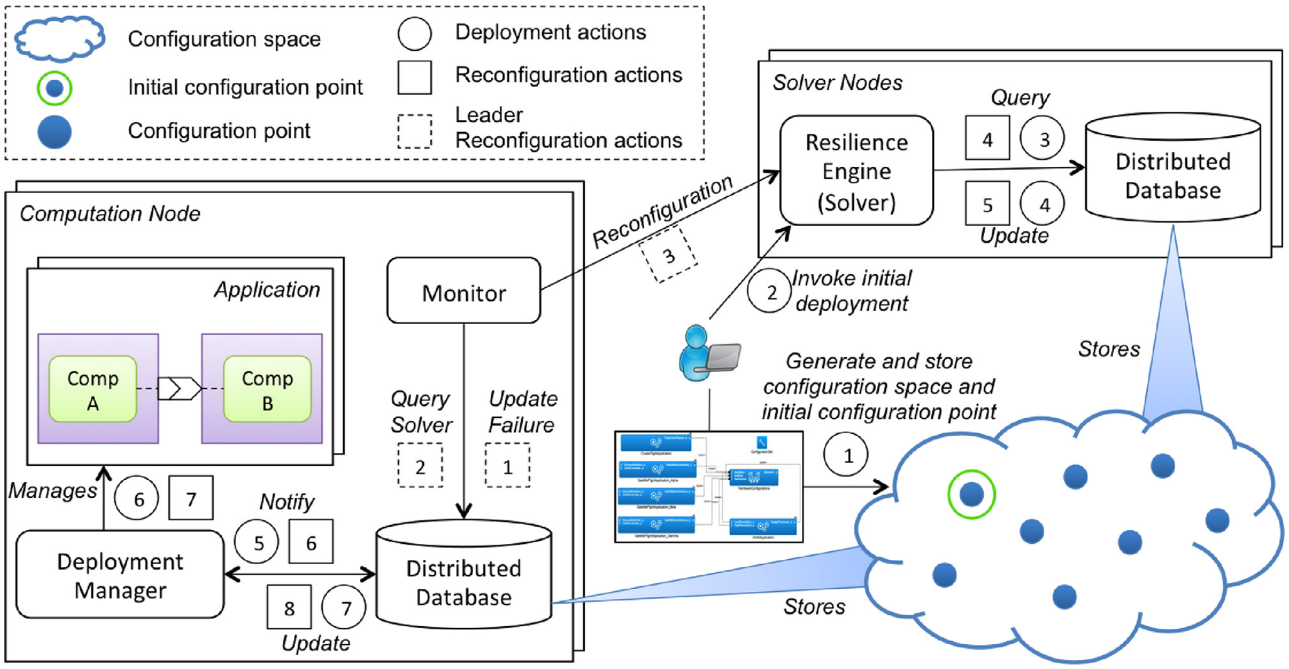


Fig. 6. Overview of the distributed self-reconfiguration infrastructure with *deployment* and *reconfiguration* action sequences. Initial deployment is triggered when a user/system integrator generates and stores the configuration space and the initial configuration point for a system using the design-time modeling tool. Once this is done, a resilience engine (RE) is invoked to instigate initial deployment. The RE then computes the required deployment actions and stores them in the database. At this point, the deployment managers (DMs) that are responsible for taking these actions are notified after which they execute those commands locally to complete initial deployment. Reconfiguration is similar, however, unlike a user/system integrator instigating the process, it is a monitor that instigates the process by logging information about any detected failure to the database and invoking the RE. This should only be done by a single monitor, as such we dedicate this task to the leader monitor, i.e., the monitor running on the leader node.

our architecture, as shown in Fig. 6, are responsible for reporting diagnosed failures to the database and invoking the Resilience Engine (RE) in order to initiate system reconfiguration.

***Resilience engine* :** A Resilience Engine (RE) is primarily responsible for computing a new configuration point at runtime when faults occur in the current state. This is important because once a new configuration point is computed, reconfiguring the system involves moving the system from the current configuration point, which is considered faulty, to the new configuration point. The mechanism used to calculate a new configuration point is described in Section 8.2. In general, a RE first determines different entities affected by the failure, and then it uses the Z3 solver to compute a new configuration point, considering various constraints and available resources. If a new configuration point is successfully computed, the local database instance is updated accordingly. This updated information is later used by appropriate DMs to reconfigure the system.

***Deployment manager* :** In our architecture, *Deployment Managers (DMs)* running on each node are used as local adaptation engines that are responsible for managing the lifecycle of application components deployed on its node. Therefore, collectively, these DMs form a distributed deployment and configuration infrastructure that manages various distributed applications by performing (a) initial deployment and configuration, (b) runtime adaptation via reconfiguration, and (c) termination. In our prior work (Pradhan et al., 2014), we identified key requirements for resilient deployment and configuration infrastructure and implemented a prototype. Here we address the same requirements but our implementation is different and relies heavily on capabilities already provided by the distributed database. For example, dynamic group membership is one of the key requirements identified and implemented in Pradhan et al. (2014). However, for the work presented in this paper, we simply make use of group membership capability (i.e., replica set) supported by MongoDB.

As shown in Fig. 6, DMs in our architecture simply listen for notification from their local database instance. Although MongoDB does not include notification service, we implement a simple notification mechanism based on MongoDB replica set Oplog, which is a database collection that stores every database event. Once a DM is notified of an event of its interest, it queries the database to obtain a set of application management actions that it should perform. These actions will be related to application components hosted on locally on the same node; a DM in one node cannot manage application components deployed on a different node. Once a DM takes required actions, it updates the database accordingly.

8.2. Self-reconfiguration mechanism

Once a failure is detected, our two-phase resilient recovery mechanism outlined in Algorithms 3 and 4 ensures that the system undergoes the required reconfiguration. The different actions involved in this mechanism are also illustrated in Fig. 6.

Phase 1 — Computing a new configuration point: The first phase of the reconfiguration mechanism is instigated once a RE is invoked after detection of a failure. In this phase, the RE computes a new configuration point by using information about the failure, the current configuration point, and relevant deployment and resource constraints in the configuration space. In order to do so, the RE uses aforementioned information to form a SMT problem and feeds it to the Z3 solver as we did for the design-time resilience analysis (Section 7.2).

Algorithm 3 presents the different steps involved in computing a new configuration point. In the beginning of this algorithm (step 1), a component-to-node (C2N) matrix (see Definition 5) is constructed using information about different available components and nodes. The next step (step 2) creates a SMT constraint over

Algorithm 3 Configuration point computation algorithm.

```
Input: functions ($fn$), components ($c$), nodes ($n$), failure ($fl$)
Output: a valid configuration point
 1: $c2n$ = a C2N matrix constructed using $c$ and $n$   ▷ See Definition 5.
 2: $cst\_1$ = an assignment constraint over $c2n$   ▷ Ensures that a component is only deployed in a single node
 3: $r2n$ = a R2N matrix constructed using nodes in $n$ and resources provided by each node   ▷ R2N matrix is a resource-to-node matrix.
 4: $r2c$ = a R2C matrix constructed using components in $c$ and resources required by each component   ▷ R2C matrix is a resource-to-component matrix.
 5: $cst\_2$ = a resource constraint over $r2n$ and $r2c$   ▷ Ensures that resource requirements of components are met.
 6: $cst\_3$ = a failure constraint using $fl$   ▷ Ensures that a failed node is empty or failed a component is not re-deployed.
 7: $solver = create\_Z3\_solver()$
 8: add constraint to $solver$ using $fn$   ▷ Ensures that all functions are provided.
 9: add constraint $cst\_1$, $cst\_2$, and $cst\_3$ to $solver$
10: $solution = null$
11: while true do
12:     $result = solver.check()$
13:     if $result$ == unsat then
14:         $solver.pop()$
15:         if $solution$ == null then
16:             return null
17:         else
18:             return $solution$
19:     else
20:         $solution = solver.model()$
21:         add distance constraint to the solver using $solution$
```

Algorithm 4 Reconfiguration commands computation and reconfiguration execution algorithm.

```
 1: procedure COMPUTE RECONFIGURATION COMMANDS   ▷ Executed by the RE that computes a new configuration point.
 2:     $commands = null$
 3:     $c2n\_new$ = C2N matrix of the new configuration point   ▷ This is computed using Algorithm 3
 4:     $c2n\_cur$ = C2N matrix of the current configuration point
 5:     for component $c$ in $c2n\_new$ do
 6:         for node $n$ in $c2n\_new$ do
 7:             if $c2n\_curr_{cn} < c2n\_new_{cn}$ then   ▷ Component missing in current
 8:                 create start command for component $c$ in node $n$
 9:                 add command to $commands$ list
10:             if $c2n\_curr_{cn} > c2n\_new_{cn}$ then   ▷ Component missing in new
11:                 create stop command for component $c$ in node $n$
12:                 add command to $commands$ list
13:     store $commands$ in the database
14:
15: procedure RECONFIGURATION EXECUTION   ▷ Runs infinitely in each DM.
16:     if reconfiguration notification received then   ▷ One per command.
17:         retrieve the $reconfiguration\_command$ from the database
18:         if $reconfiguration\_command$ is for this node then
19:             if $reconfiguration\_command$ == START then
20:                 create a new process and save $pid$ in the database
21:             if $reconfiguration\_command$ == STOP then
22:                 use component name to retrieve $pid$ from the database
23:                 kill the process using $pid$
24:             mark $reconfiguration\_command$ as executed in the database
```

the C2N matrix such that a component is only deployed in a single node. This constraint is encoded in a way to ensure that the sum of each row of the C2N matrix is exactly one.

The next step (step 3) creates a resource-to-node (R2N) matrix using information about different nodes and resources provided by those nodes. The R2N matrix comprises resources as rows and nodes as columns, and each element of the matrix is the value of a particular resource provided by the corresponding node. Similarly, a resource-to-component (R2C) matrix is created in the next step (step 4) using information about different components and resources required by those components. The R2C matrix comprises resources as rows and components as columns, and each element of the matrix is the value of a particular resource required by the corresponding component. The next step (step 5) of this algorithm is to create a resource constraint using the aforementioned R2N and R2C matrices. This constraint ensures that the resources required by components deployed on a node is satisfiable.

Once the assignment and resource constraints are encoded, the next step (step 6) in Algorithm 3 is to encode a failure constraint related to the failure that was initially detected. If the failure was a node failure, we encode the constraint such that no components are deployed on the failed node. Whereas, if the failure was a component failure, then we encode the constraint such that the component is not re-deployed. The failure constraint related to component failure could be relaxed by ensuring that a component gets re-deployed but not on nodes where it has previously failed. At this point, all constraints are encoded and the next few steps (steps 7– 9) involves creating a Z3 solver and adding different constraints to the solver.

Definition 9. Assume configuration points are represented using component-to-node matrices. The distance between two configuration points CP_1 and CP_2 can be expressed as $CP_{diff} = C_1 - C_2$, $configuration\_distance = sumabs(C_{diff}, (i))$.

After adding constraints to the solver, the next set of steps (steps 10–21) is responsible for computing a configuration point that is the least distance (see Definition 9) away from the current configuration point. In order to do so, we use a recursive logic, which upon every successful solution computation (step 20) adds a distance constraint (step 21) and invokes the solver again. The distance constraint is encoded using distance between computed solution and the current configuration point. It is encoded to ensure a new solution (one that will be computed in the next round of iteration) is lesser distance away from the current configuration point in comparison to the solution computed in this iteration. As such, by adding this constraint, we are asking the solver to find a better solution in every iteration of successful solution computation. There will come a point when the solver will not be able to find a better solution (step 13), in which case we check if the solution from previous step is valid (step 15) and return that as the closest configuration point (step 18). This is an important heuristic as we do not want the system to deviate too much from its current configuration. It also guarantees minimal reconfiguration time as the number of changes required will be minimal due to the fact that the distance between the configuration points is the least possible.

Once a new configuration point is computed, it is stored in the database as a desired state. The next phase of our self-reconfiguration mechanism is responsible for using this configuration point to compute reconfiguration commands required to transition from the current configuration point to the new configuration point; we discuss this is detail below. Here, it is important to note that our current implementation of the runtime self-reconfiguration mechanism does not analyze or verify different properties (timing, network QoS) of the configuration points computed at runtime. As mentioned before, this is an important processes, specially for mobile CPS that host mission critical applications. However, our work presented in paper demonstrates our initial effort towards achieving autonomous resilience; integrating analysis and verification of different properties with the runtime reconfiguration loop is part of our future work.

Phase 2 — Computing reconfiguration commands and reconfiguring the system using those commands: The second phase of our reconfiguration mechanism, shown in Algorithm 4, is responsible for computing reconfiguration commands and performing the reconfiguration itself. In order to compute the reconfiguration commands required to transition from one configuration point to another, the *compute reconfiguration commands* procedure of Algorithm 4 is used. As shown, this procedure takes C2N matrices of newly computed configuration point (step 3) and current configuration point (line 4) to determine different reconfiguration commands (steps 5–13). To determine reconfiguration commands, we check how each element of the aforementioned matrices are different when compared to each other (steps 7 and 10). Depending on the difference we either create a *start* command or a *stop* command. The former results in creation of a new process, whereas, the latter results in termination of an existing process.

Once all reconfiguration commands are computed and stored in the database, the *reconfiguration execution* procedure of Algorithm 4 is used to ensure all reconfiguration commands are executed. This procedure is executed infinitely by each DM.

When a DM is notified about a reconfiguration command (step 16), it checks if it should execute that command (step 17). A DM should only execute commands that are targeted for its host node. Once a DM determines a command that it should execute, it will check whether the command is a start or stop command and execute the command accordingly (steps 19–23). Finally, after executing a command, the DM updates the database to acknowledge that the command has been executed. Once this happens for all reconfiguration commands pertaining to a configuration point transition, we can claim that the system has successfully self-reconfigured.

9. Case study: fractionated satellite cluster

In this section, we first present our use case scenario comprising three applications deployed on a cluster of fractionated satellite. Second, we demonstrate the design-time resilience metrics computation. Finally, we demonstrate the runtime self-reconfiguration mechanism using a small scale system, and evaluate it using a larger system.

9.1. Scenario

In order to perform different demonstrations and evaluation, we use a fractionated satellite scenario where we have a simple system with the following objectives: (a) satellite flight applications to control the position of each satellite, (b) an imaging application to capture images, and (c) a cluster flight planning application to coordinate the flight paths and positions of the different satellites. The software (application) model for this system is shown in Fig. 7. We use a GME (Ledeczi et al., 2001) based modeling front-end that allows a user to model complex systems using a graphical modeling language.

As shown in Fig. 7, this system is comprised of three different kinds of applications: (a) a single instance of*ClusterFlightApplication*, which is responsible for satisfying the objective of coordinated flight planning, (b) three different instances of a flight control application called *SatelliteFlightApplication*, one for each node for a three-node initial deployment scenario, and (c) a single instance of *WAMApplication*, which is responsible for satisfying the imaging objective. The *HardwareConfigurations* for this application suggest the requirements for the three nodes, which are named *SatAlpha*, *SatBeta*, and *SatGamma* in the model. The initial deployment maps the *ClusterFlightApplication* to node *SatAlpha*, the *SatelliteFlightApplication* instances to all three nodes, and three different components of the *WAMApplication* to the three different nodes.

Each instance of the *SatelliteFlightApplication* is composed of three components (a) an *OrbitController* component, which is responsible for manipulating thrusters to control satellite movement and position, (b) a *GroundInterface* component, which is responsible for communicating with a ground station, and (c) a *SatelliteBusInterface* component, which is responsible for interacting with the satellite bus. The *ClusterFlightApplication* contains a single component, a *TrajectoryPlanner* component, which is responsible for planning and co-ordinating flights paths of the different satellites. The *WAMApplication* is composed a *HighResolutionImageGrabber* component, a *LowResolutionImageGrabber* component, and a *ImageProcessor* component; the first two components are responsible for capturing images of varying resolution while the third component is responsible for processing different images.

Not shown in Fig. 7 are the different devices present on each node. For our scenario, all three nodes host a *BusController* device to control the satellite bus and a *GroundInterface* device to communicate with a ground station. In addition, node *SatAlpha* also hosts an *HR_Camera* device to capture high resolution images, an *LR_Camera* device to capture low resolution images, and a *GPU* device to process captured images. Similarly, node *SatBeta* also hosts a *GPU* device, and node *SatGamma* also hosts an *HR_CAMERA* device. In addition to representing the system configuration after the initial deployment, Fig. 8 also shows all these different devices with respect to their hosting nodes.

9.2. Resilience metrics calculation

Our current implementation of the design-time resilience analysis tool only considers software component (application) failures and node failures. As such, the resulting resilience metrics are true only for component failures and node failures. In other words, the minimum and maximum number of failures tolerable are strictly component failures or node failures. Resilience analysis of system configuration presented in Fig. 8 results in resilience metrics of (1, 12), where 1 is the worst-case metric and 12 is the best-case metric. This means that the system is capable of tolerating at least one failure, regardless of what the failure is, and at most twelve failures. Again, these failures are either component or node failures. The thirteenth failure, regardless of which node or component it is, will definitely cause the system to fail.

To further explain the computed resilience metrics, let us examine the worst-case metric. Since the worst-case metric of 1 tells us that the system is always capable of tolerating a single failure, let us come up with a scenario where two failures result in the system to be non recoverable. If node *SatGamma* fails followed by the failure of node *SatAlpha*, we lose both image capturing components of which at least one is required by the *ImageProcessor* component on node *SatBeta*. Therefore, in this scenario, the two node failures was enough to render the system non recoverable.

Similarly, we can evaluate the best-case metric by coming up with one or more scenarios that shows how the system can

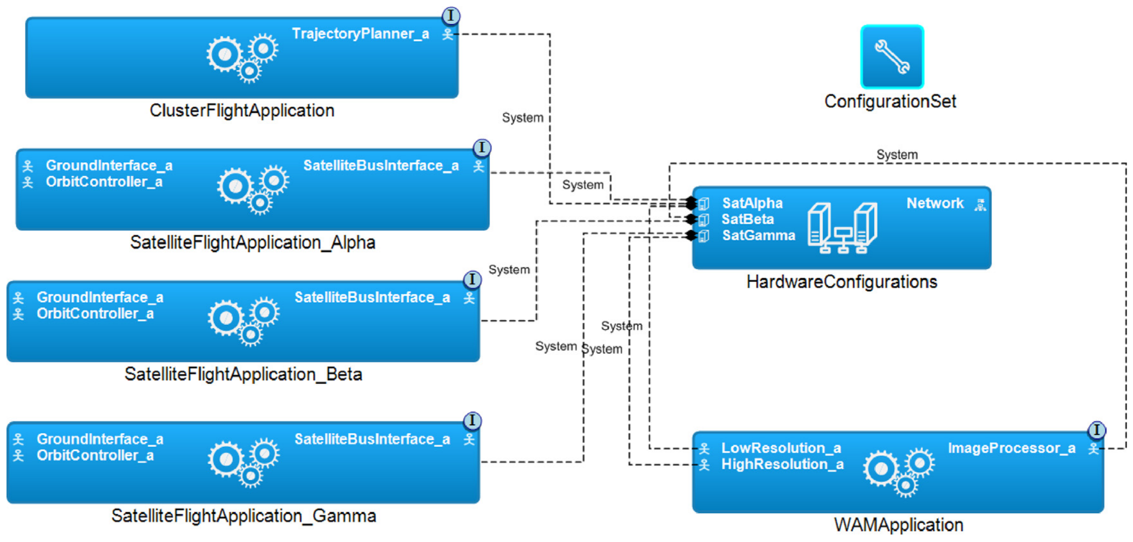


Fig. 7. Software model design using GME based modeling language.

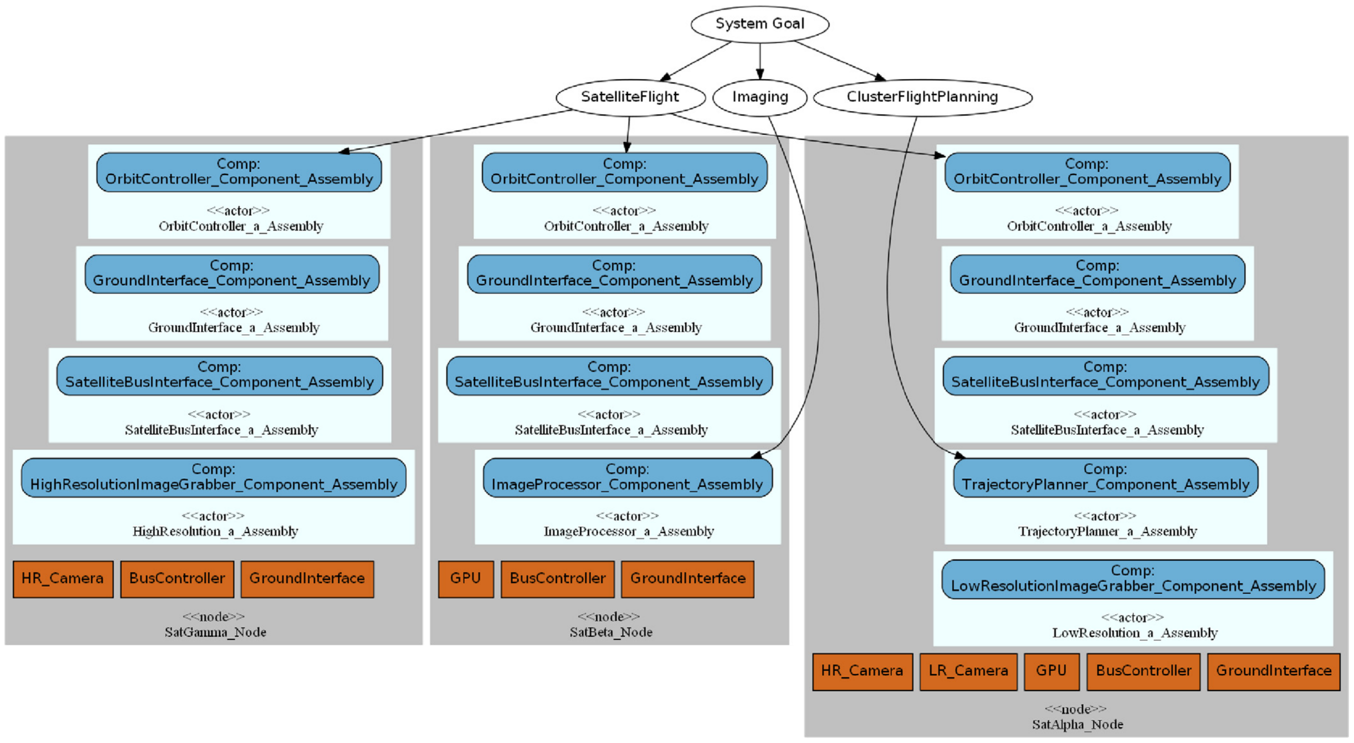
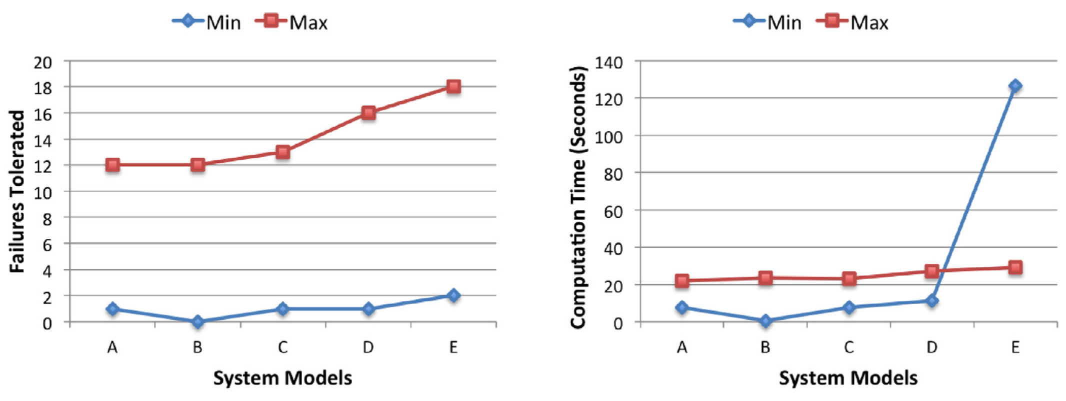

System Goal
SatelliteFlight
Imaging
ClusterFlightPlanning
Comp: OrbitController_Component_Assembly
<<actor>> OrbitController_a_Assembly
Comp: GroundInterface_Component_Assembly
<<actor>> GroundInterface_a_Assembly
Comp: SatelliteBusInterface_Component_Assembly
<<actor>> SatelliteBusInterface_a_Assembly
Comp: HighResolutionImageGrabber_Component_Assembly
<<actor>> HighResolution_a_Assembly
HR_Camera
BusController
GroundInterface
<<node>> SatGamma_Node
Comp: ImageProcessor_Component_Assembly
<<actor>> ImageProcessor_a_Assembly
GPU
<<node>> SatBeta_Node
Comp: TrajectoryPlanner_Component_Assembly
<<actor>> TrajectoryPlanner_a_Assembly
Comp: LowResolutionImageGrabber_Component_Assembly
<<actor>> LowResolution_a_Assembly
LR_Camera
<<node>> SatAlpha_Node

Fig. 8. System configuration after initial deployment of model in Fig. 7.

tolerate twelve failures. One such scenario is as follows - (a) failure of the *GroundInterface* component on node *SatGamma*, which has no effect as there are two other *GroundInterface* components, (b) failure of the *GroundInterface* component on node *SatAlpha*, which doesn't require instantiation of the component on another node but it does result in the *TrajectoryPlanner* component being restarted on node *SatBeta* since this node has a functioning *GroundInterface* component to receive important commands from ground station, (c) failure of the *HighResolutionImageGrabber* component on node *SatGamma*, which results in this component being restarted on node *SatAlpha*, (d) failure of the *SatelliteBusInterface* on node *SatGamma*, (e) failure of the *OrbitController* component on node *SatGamma*, (f) failure of node *SatGamma* itself, (g) failure of the *TrajectoryPlanning* component on node *SatBeta* resulting in it being restarted on node *SatAlpha*, (h) failure of the *ImageProcessor* component on node *SatBeta* resulting in it being restarted on node *SatAlpha*, (i) failure of the *LowResolutionImageGrabber* on node *SatAlpha*, which has no affect as the *ImageProcessing* component only requires one out of two image capturing components, (j) failure of the *GroundInterface* on node *SatBeta*, which results in system still being functional but not able to receive any new ground commands, (k) failure of the *SatelliteBusInterface* component on node *SatBeta*, and finally (l) failure of the entire node *SatBeta*, which results in all remaining components being hosted on node *SatAlpha*.

The purpose of computing these resilient metrics is to quantify the resilience of a system. Using these metrics we can compare between different deployments or versions (for example, with different resources) of the same system and determine the most



(a) Resilience metrics for different system models.
(b) Resilience metrics computation time for different system models.

Fig. 9. Resilience metrics (left) and corresponding computation time (right) for different variation of system model presented in Fig. 7. **A** represents the default model (shown in Fig. 7), **B** represents a model in which a *GPU* device is removed from *SatAlpha*, **C** represents a model in which a *HR_Camera* is added to *SatBeta*, **D** represents a model in which a new node similar to *SatGamma* is added to the system, and **E** represents a model in which a new node similar to *SatAlpha* is added to the system.

resilient. Based on this analysis, we can judge the tradeoffs between resilience and the different system designs, and make a well-informed decision before deploying a system.

Fig. 9 presents resilience metrics (Fig. 9(a)) and corresponding computation time (Fig. 9(b)) for different variations of system model presented in Fig. 7. As shown in the figure, maximum failure computation time ranges between 20 and 30 s. Similarly, minimum failure computation time ranges between 0 and 20 s for system models *A–D*. However, minimum computation time for system model *E* is considerably higher at 126.63 s. This is because when a new node with five different devices is added to the default model, the search space expands considerably for each failure scenario considered. However, this is acceptable as this analysis is done during design-time.

9.3. Runtime self-reconfiguration mechanism demonstration

In this section, we demonstrate the self-reconfiguration capability provided by our runtime infrastructure. Fig. 8 shows the system configuration after the initial deployment. From this figure, it is clear that the system requires three different objectives – *SatelliteFlight, Imaging*, and *ClusterFlightPlanning* – to satisfy its high-level goal. To test resilience, we first inject a component failure[1] by failing the *ImageProcessor* component in node *SatBeta*. Once a *Resilience Engine* receives this failure report, it computes a new configuration point and a list of reconfiguration commands to transition the system from its current configuration point (faulty) to the new configuration point. In this particular scenario, the *Resilience Engine* computes a solution that requires the *ImageProcessor* component to be moved from node *SatBeta* to *SatAlpha*; this makes sense because the *ImageProcessor* component requires *GPU* device and the only other node with a *GPU* device is node *SatAlpha*. The resulting configuration is presented in Fig. 10.

9.4. Runtime self-reconfiguration mechanism evaluation

Fig. 11 presents result of two experiments we performed to evaluate our self-reconfiguration mechanism. First, we compare the time taken to compute new configuration points after failures in the system model presented in Fig. 7. As shown in the Fig. 11(a), we consider four failures; first two failures are component failures (*ImageProcessor* and *LowResolutionImageGrabber*), third failure is a node failure (*SatBeta*), and fourth failure is another component failure (*TrajectoryPlanner*). The time taken to compute a new configuration point for all four failure cases is 0.34 s on average, with minimum 0.31 s and maximum 0.36 s. The range here is 50 milliseconds. This is because all four failures are invoked in the same system, which means the size of the C2N matrix will be the same.

Second, we compare the time taken to compute new configuration points for different system models. As shown in Fig. 11(b), we use seven different system models and for each system model we compute the average configuration computation time with regards to the same four failures that we used for our first evaluation experiment (Fig. 11(a)). Although all seven system models comprise similar nodes and components, the number of nodes and components in each system model is different. System model A is the simplest and resembles the basic system model shown in Fig. 7; it comprises of three nodes and thirteen components. System model B comprises five nodes and nineteen components. System model C comprises eight nodes and twenty-eight components. System model D comprises ten nodes and thirty-four components. System model E comprises twelve nodes and forty components. System model F comprises fifteen nodes and forty-nine components. System model G comprises eighteen nodes and fifty-eight components. So we can see that these system models have increasing complexity. As we can clearly see in Fig. 11(b), systems with higher complexity have higher average configuration computation time. The reason behind this is the size of the C2N matrix over which all constraints are encoded. Furthermore, the size of the C2N matrix also tells us about the size of the resource matrices (R2C and R2N). Since, we are considering increasing scale of the same system model, we can argue that the size of the R2C and R2N matrices will also increase as the system complexity increases. As such, the more complex a system, the larger its configuration space (all three different matrices and corresponding constraints written over them) and therefore the increase in time taken by the underlying Z3 solver to find a solution.

Here, we would like to state that, although Fig. 11(b) presents result based on single iteration of the experiment, we were able to reproduce similar results for multiple iterations of the same experiment.

From these results we can see that configuration computation time increases with increase in system complexity. As such, we have to be careful when choosing what classes of systems this solution is applied to. For example, it might not be feasible to deploy

[1] Failure injection is as simple as changing the status of the device to mark it as failed.

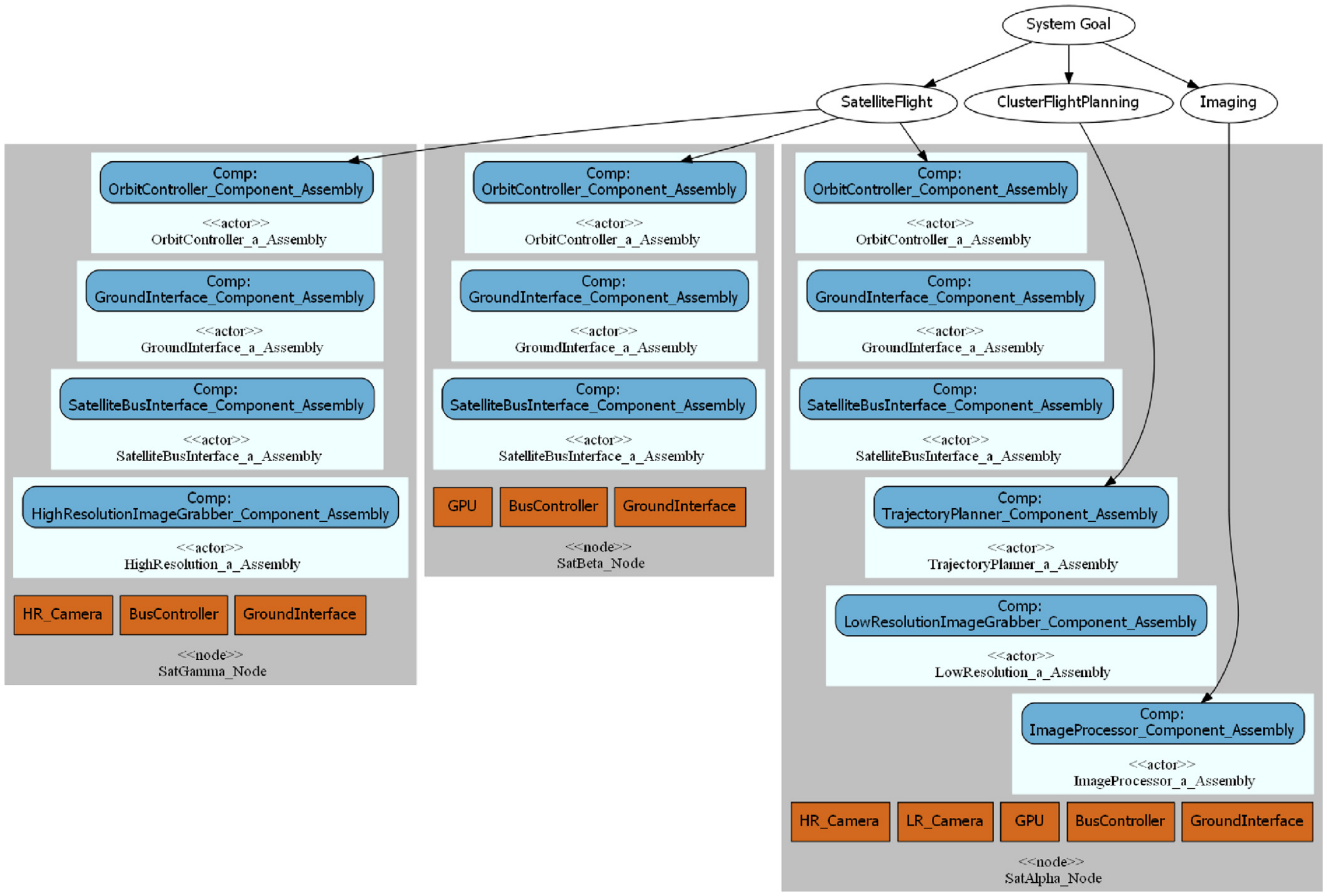
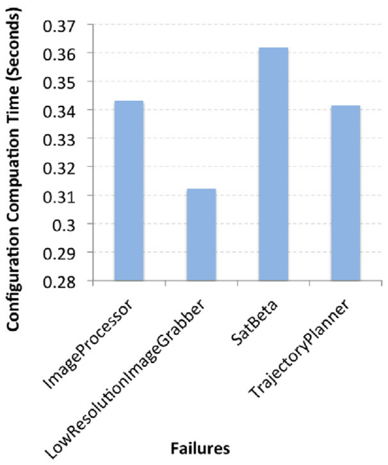
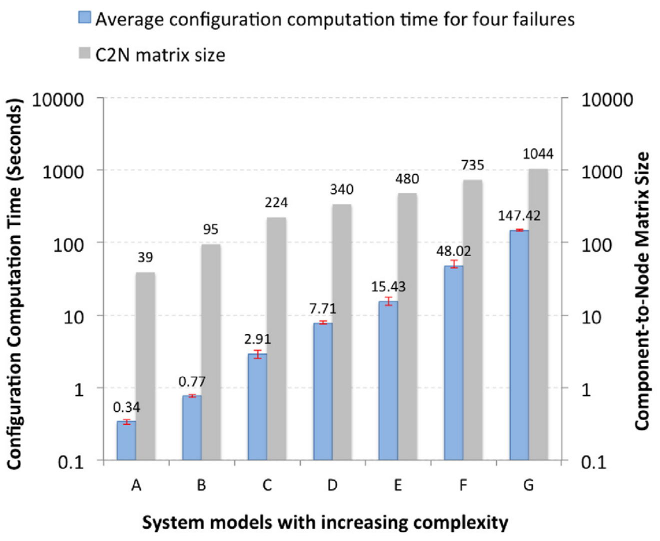

System Goal
SatelliteFlight
ClusterFlightPlanning
Imaging
Comp: OrbitController_Component_Assembly
<<actor>> OrbitController_a_Assembly
Comp: GroundInterface_Component_Assembly
<<actor>> GroundInterface_a_Assembly
Comp: SatelliteBusInterface_Component_Assembly
<<actor>> SatelliteBusInterface_a_Assembly
Comp: HighResolutionImageGrabber_Component_Assembly
<<actor>> HighResolution_a_Assembly
HR_Camera
BusController
GroundInterface
<<node>> SatGamma_Node
GPU
<<node>> SatBeta_Node
Comp: TrajectoryPlanner_Component_Assembly
<<actor>> TrajectoryPlanner_a_Assembly
Comp: LowResolutionImageGrabber_Component_Assembly
<<actor>> LowResolution_a_Assembly
Comp: ImageProcessor_Component_Assembly
<<actor>> ImageProcessor_a_Assembly
LR_Camera
<<node>> SatAlpha_Node

Fig. 10. System configuration after recovering from *ImageProcessor* component failure in node *SatBeta*. Compare it to the initial configuration shown in Fig. 8.



(a) Configuration computation time for failures in the simple system model presented in Figure 7.



(b) Average configuration computation time for four failures in different system models. Both y-axis are in logarithmic scale.

Fig. 11. Configuration computation time for failures in a simple model (left) and average configuration computation time for four failures in different system models (right). The different system models have increasing complexity; **A** has three nodes and thirteen components, **B** has five nodes and nineteen components, **C** has eight nodes and twenty-eight components, **D** has ten nodes and thirty-four components, **E** has twelve nodes and forty components, **F** has fifteen nodes and forty-nine components, and **G** has eighteen nodes and fifty-eight components.

this solution as is to large-scale hard real-time systems. To resolve this issue, we are currently working on developing a variation of the self-reconfiguration mechanism presented in this paper. The self-reconfiguration mechanism presented in this paper follows a reactive approach, where the system reacts to failures. However, instead of reacting to failures, another interesting approach would be to *look-ahead* for failures, and pre-compute and store solutions before failures happen.

10. Conclusions

Mobile distributed Cyber-Physical Systems (CPS) – such as fractionated spacecraft, UAV clusters and team-centric autonomous robots – hosting heterogeneous embedded applications have grown in popularity in recent years. Such platforms are typically deployed by composing several smaller CPS units, each with its own complex physical dynamics and time-varying resource requirements. Resilience is an important attribute for such distributed systems since these platforms could potentially be hosting mixed-priority mission-critical applications with various functional goals. Systems such as fractionated spacecraft and UAV clusters are remotely deployed, which necessitates resilience to be autonomous since human intervention is very limited. Resilience autonomy is important also because these systems can be very complex for manual reconfiguration. It is therefore necessary to study such systems both prior to deployment and during runtime lifecycle management to ensure that failures are either completely avoided or safely handled.

In this respect, we have identified two key requirements: (a) the need to analyze the system at design-time before deployment in order to admit the deployment units as sufficiently safe for operation, e.g., with respect to timing and network requirements, and (b) the need for a runtime infrastructure to handle failures by performing self-reconfiguration in an autonomous manner. To address these requirements, this paper makes the following contributions: (a) a set of design-time analysis tools to perform timing, network QoS, and resilience analysis, and (b) a runtime infrastructure that governs the self-reconfiguration mechanisms. The timing and network analysis tools were evaluated in our prior work. In this paper we present empirical evaluation of the design-time resilience analysis tool and the runtime self-reconfiguration infrastructure. The algorithms and computations discussed here include the result of multiple iterations and lessons learned while implementing DREMS (Levendovszky et al., 2013).

In the future, we intend to extend the work presented in this paper by focusing on following runtime infrastructure improvements: (a) integrating design-time tools with the runtime infrastructure such that new configuration points calculated at runtime can be analyzed and validated before reconfiguring the system, (b) adding comprehensive monitoring, detecting, and diagnosing capabilities, (c) adding mechanisms to handle temporary and intermittent failures, and (d) implementing a complete look-ahead algorithm to reduce configuration computation time.

Acknowledgments

The DARPA System F6 Program, the National Science Foundation (under awards CNS-1035655 and CNS-1238959), the Air Force Research Laboratory (under award FA8750-14-2-0180), and Siemens Corporate Technology supported this work. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not reflect the views of DARPA, NSF, AFRL, or Siemens Corporate Technology.

References

Alur, R., Dill, D.L., 1994. A theory of timed automata. Theor. Comput. Sci. 126, 183–235.
Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W., 2004. Times: a tool for schedulability analysis and code generation of real-time systems. In: Larsen, K., Niebert, P. (Eds.), Formal Modeling and Analysis of Timed Systems. In: Lecture Notes in Computer Science, Vol. 2791. Springer, Berlin Heidelberg, pp. 60–72. doi:10.1007/978-3-540-40903-8_6.
Andrade, S.S., de Araújo Macêdo, R.J., 2009. A non-intrusive component-based approach for deploying unanticipated self-management behaviour. In: Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on. IEEE, pp. 152–161.
Arshad, N., Heimbigner, D., Wolf, A.L., 2007. Deployment and dynamic reconfiguration planning for distributed software systems. Softw. Qual. J. 15 (3), 265–281.
Asmare, E., Gopalan, A., Sloman, M., Dulay, N., Lupu, E., 2012. Self-management framework for mobile autonomous systems. J. Netw. Syst. Manage. 20 (2), 244–275.
Audsley, N., Burns, A., Davis, R., Tindell, K., Wellings, A., 1995. Fixed priority pre-emptive scheduling: an historical perspective. Real-Time Syst. 8 (2-3), 173–198. doi:10.1007/BF01094342.
Balasubramanian, D., Dubey, A., Otte, W., Levendovszky, T., Gokhale, A., Kumar, P., Emfinger, W., Karsai, G., 2015. DREAMS ML: a wide spectrum architecture design language for distributed computing platforms. Sci. Comput. Program..
Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C., 2009. Satisfiability modulo theories, Handbook of satisfiability 185, 825–885.
Brown, O., Eremenko, P., 2006. The Value Proposition for Fractionated Space Architectures AIAA Paper 2006-7506.
Derler, P., Feng, T.H., Lee, E.A., Matic, S., Patel, H.D., Zhao, Y., Zou, J., 2008. PTIDES: A Programming Model for Distributed Real-Time Embedded Systems. Technical Report UCB/EECS-2008-72. EECS Department, University of California, Berkeley.
Dubey, A., Karsai, G., Mahadevan, N., 2011a. Model-based software health management for real-time systems. In: Aerospace Conference, 2011 IEEE. IEEE, pp. 1–18.
Dubey, A., Karsai, G., Mahadevan, N., 2011b. Model-based software health management for real-time systems. In: Aerospace Conference, 2011 IEEE. IEEE, pp. 1–18.
Emfinger, W., Karsai, G., Dubey, A., Gokhale, A., 2014. Analysis, verification, and management toolsuite for cyber-physical applications on time-varying networks. In: Proceedings of the 4th ACM SIGBED International Workshop on Design, Modeling, and Evaluation of Cyber-Physical Systems. ACM, New York, NY, USA, pp. 44–47. doi:10.1145/2593458.2593459.
García-Valls, M., Uriol-Resuela, P., Ibáñez-Vázquez, F., Basanta-Val, P., 2014. Low complexity reconfiguration for real-time data-intensive service-oriented applications. Future Gener. Comput. Syst. 37, 191–200.
Harbour, M.G., Garcia, J.J.G., Gutierrez, J.C.P., Moyano, J.M.D., 2001. Mast: modeling and analysis suite for real time applications. In: In 13th Euromicro Conference on Real-Time Systems, p. 125.
Heineman, G.T., Councill, W.T. (Eds.), 2001. Component-based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
Jensen, K., Kristensen, L.M., 2009. Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer.
Karsai, G., Balasubramanian, D., Dubey, A., Otte, W., 2014. 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, pp. 1–8. doi:10.1109/ISORC.2014.36.
Kumar, P., Karsai, G., 2015. Integrated analysis of temporal behavior of component-based distributed real-time embedded systems. In: Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW), 2015 IEEE International Symposium on Real-time Computing (ISORC), pp. 50–57. doi:10.1109/ISORCW.2015.56.
Kumar, P.S., Dubey, A., Karsai, G., 2014. Colored petri net-based modeling and formal analysis of component-based applications. In: 11th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2014, p. 79.
Kurtoglu, T., Tumer, I.Y., Jensen, D.C., 2010. A functional failure reasoning methodology for evaluation of conceptual system architectures. Research in Engineering Design 21 (4), 209–234.
Laprie, J.-c., 2008. From dependability to resilience. In: In 38th IEEE/IFIP Intenational Conference on Dependable Systems and Networks. Citeseer.
Le Boudec, J.-Y., Thiran, P., 2001. Network Calculus: A Theory of Deterministic Queuing Systems for the Internet. Springer-Verlag, Berlin, Heidelberg.
Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P., 2001. The generic modeling environment. Workshop on Intelligent Signal Processing.
Levendovszky, T., Dubey, A., Otte, W., Balasubramanian, D., Coglio, A., Nyako, S., Emfinger, W., Kumar, P., Gokhale, A., Karsai, G., 2013. Drems: a model-driven distributed secure information architecture platform for managed embedded systems.
Levendovszky, T., Dubey, A., Otte, W.R., Balasubramanian, D., Coglio, A., Nyako, S., Emfinger, W., Kumar, P., Gokhale, A., Karsai, G., 2014. Distributed real-time managed systems: a model-driven distributed secure information architecture platform for managed embedded systems. Softw. IEEE 31 (2), 62–69.
Macariu, G., Cretu, V., 2010. Timed automata model for component-based real-time systems. In: Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on, pp. 121–130. doi:10.1109/ECBS.2010.20.

Mahadevan, N., Dubey, A., Balasubramanian, D., Karsai, G., 2013. Deliberative, search-based mitigation strategies for model-based software health management. Innov. Syst. Softw. Eng. 9 (4), 293–318.

Mahadevan, N., Dubey, A., Karsai, G., 2011a. Application of software health management techniques. In: SEAMS, pp. 1–10.

Mahadevan, N., Dubey, A., Karsai, G., 2011b. Application of software health management techniques. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. ACM, New York, NY, USA, pp. 1–10. doi:10.1145/1988008.1988010.

Masri, A., Bourdeaud_huy, T., Toguyeni, A., 2009. A component-based approach based on high-level petri nets for modeling distributed control systems. Int. J. Adv. Intell. Syst. 2 (2 et 3), 335–353.

Mehrotra, R., Dubey, A., Abdelwahed, S., Krisa, R., 2012. RFDMon: a real–time and fault-tolerant distributed system monitoring approach. In: The Eighth International Conference on Autonomic and Autonomous Systems, pp. 57–63.

MongoDB Incorporated, 2009. MongoDB. http://www.mongodb.org.

de Moura, L.M., Bjørner, N., 2008. Z3: an efficient smt solver. In: TACAS, pp. 337–340.

Pradhan, S., Dubey, A., Otte, W.R., Karsai, G., Gokhale, A., 2015. Towards a product line of heterogeneous distributed applications. ISIS 15, 117.

Pradhan, S., Otte, W., Dubey, A., Szabo, C., Gokhale, A., Karsai, G., 2014. Towards a self-adaptive deployment and configuration infrastructure for cyber-physical systems. ISIS 14, 102.

Ratzer, A.V., Wells, L., Lassen, H.M., Laursen, M., Qvortrup, J.F., Stissing, M.S., Westergaard, M., Christensen, S., Jensen, K., 2003. CPN tools for editing, simulating, and analysing coloured petri nets. In: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets. Springer-Verlag, Berlin, Heidelberg, pp. 450–462.

Renault, X., Kordon, F., Hugues, J., 2009a. Adapting models to model checkers, a case study : analysing AADL using time or colored petri nets. In: Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on, pp. 26–33. doi:10.1109/RSP.2009.30.

Renault, X., Kordon, F., Hugues, J., 2009b. From AADL architectural models to petri nets: checking model viability. In: Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC '09. IEEE International Symposium on, pp. 313–320. doi:10.1109/ISORC.2009.11.

Schaeffer-Filho, A., Lupu, E., Sloman, M., 2014. Federating policy-driven autonomous systems: interaction specification and management patterns. J. Netw. Syst. Manage. 1–41.

Schmidt, D.C., 2006. Guest editor's introduction: model-driven engineering. Computer 39 (2), 0025–31.

Sha, L., Abdelzaher, T., Arzen, K.-E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K., 2004. Real time scheduling theory: a historical perspective. Real-Time Syst. 28 (2-3), 101–155. doi:10.1023/B:TIME.0000045315.61234.1e.

Singhoff, F., Legrand, J., Nana, L., Marcé, L., 2004. Cheddar: a flexible real time scheduling framework. In: Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-time &Amp; Distributed Systems Using Ada and Related Technologies. ACM, New York, NY, USA, pp. 1–8. doi:10.1145/1032297.1032298.

Srivastava, A., Schumann, J., 2011. The case for software health management. In: Fourth IEEE International Conference on Space Mission Challenges for Information Technology, 2011. SMC-IT 2011., pp. 3–9.

Sztipanovits, J., Karsai, G., 1997. Model-integrated computing. Computer 30 (4), 110–111.

Valls, M.G., López, I.R., Villar, L.F., 2013. iland: an enhanced middleware for real–time reconfiguration of service oriented distributed real-time systems. Indus. Inf. IEEE Trans. 9 (1), 228–236.

Varga, A., Hornig, R., 2008. An overview of the OMNeT++ simulation environment. In: Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), ICST, Brussels, Belgium, Belgium, pp. 1–10.

Subhav Pradhan is a graduate research assistant at ISIS at Vanderbilt University. His research focuses on resilient deployment and configuration of distributed component-based software applications. He received his MSc in computer science from Vanderbilt University in 2012.

Abhishek Dubey is an assistant professor of Computer Engineering and Computer Science Department in Vanderbilt University. He received his PhD in electrical engineering from Vanderbilt University in 2009. His interests include distributed fault-tolerant real-time systems and autonomic computing.

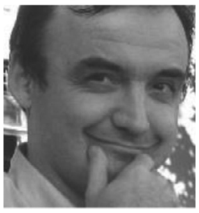
Tihamer Levendovszky is a research assistant professor at Vanderbilt University. He received his PhD from the Budapest University of Technology and Economics. His interests include automated software engineering, model-based engineering, computer security, and performance analysis of software systems.

Pranav Srinivas Kumar is a graduate research assistant at ISIS at Vanderbilt University. His research focuses on modeling, analysis and verification techniques for distributed component-based software applications. He received his B.E. in electronics and communications engineering from Anna University, India in 2011.

William Emfinger is a post-doctoral researcher at Institute for Software Integrated Systems at Vanderbilt University. His research focuses on networking for critical systems. He received his B.E. in electrical engineering and biomedical engineering from Vanderbilt University in 2011.

Daniel Balasubramanian is a Research Scientist at Institute for Software Integrated Systems at Vanderbilt University. He received his PhD in Computer Science from Vanderbilt University. His interests include the lightweight application of formal methods and analysis to model-based development.

William R. Otte is a research scientist at Institute for Software Integrated Systems at Vanderbilt University. He received his PhD in computer science from Vanderbilt University. His interests include middleware for real-time embedded systems and deployment and their configuration.



Gabor Karsai is professor of Electrical and Computer Engineering and Computer Science at Vanderbilt University and senior research scientist at Institute for Software Integrated Systems. He conducts research in model-integrated computing (MIC), design automation for model-driven development processes, automatic program synthesis, and the application of MIC in various government and industrial projects. He is a senior member of the IEEE Computer Society.