2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems

Using Dynamic Adaptive Systems in Safety-Critical Domains

Ethan T. McGee
Clemson University
McAdams Hall
Clemson, SC 29632
etmcgee@clemson.edu

John D. McGregor
Clemson University
McAdams Hall
Clemson, SC 29632
johnmc@clemson.edu

ABSTRACT

The development of safety-critical Cyber-Physical Systems (CPS) is expanding due to the Internet of Things' promise to make high-integrity applications and services part of everyday life. This expansion is seen in the dependencies some connected vehicles have on cloud services that provide guidance and accident avoidance / detection features. Such systems are safety-critical since failure could result in serious injury or death. Due to the severe consequences of failure, fault-tolerance, reliability and dependability should be primary driving qualities in the design and development of these systems. However, the cost of the analysis, evaluation and certification activities needed to ensure that the possibility of failure has been sufficiently mitigated is significantly higher than the cost of developing traditional software.

Our group is exploring the addition of dynamic adaptive capabilities to safety-critical systems. We postulate that dynamic adaptivity could provide several enhancements to safety-critical systems. It would allow systems to reason about the environment within which they are sited and about their internal operation enabling decision making that is context-specific and appropriately prioritized. However, the addition of adaptivity with the associated overhead of reasoning is not without drawbacks particularly when hard real-time safety-critical systems are involved. In this brief position paper, we explore some of the questions and concerns that are raised when dynamic adaptive behavior is introduced into safety-critical systems as well as ways that the Architecture Analysis & Design Language (AADL) can be used to model / analyze such systems.

CCS Concepts

•Software and its engineering → Software organization and properties; Designing software;

SEAMS'16, May 16-17, 2016, Austin, TX, USA

 ISBN 978-1-4503-4187-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2897053.2897062

Keywords

Safety Critical Systems; Dynamic Adaptive Systems; Software Product Lines; Dynamic Software Product Lines

1. INTRODUCTION

Cyber-Physical Systems (CPS) and the Intenet of Things (IoT) are seeing expanded use in a wide variety of domains. They are enabling the automation of common household devices, and they are also playing a major role in the implementation of connected / autonomous vehicles [23]. Prototype autonomous vehicles are relying on infrastructures powered by the IoT [14] for features including Global Positioning System (GPS) navigation, adaptive cruise control and automated accident response / detection among others. Especially for autonomous vehicles, these systems would be considered safety-critical, where system failure could result in serious injury or death. We consider adding dynamic adaptive capabilities to safety-critical systems as a way to extend the life, deploy-ability and functionality of the systems. This poses significant challenges to the analysis and certification that systems are safe.

Dynamic Adaptive Systems (DAS) are systems that monitor select properties of their environment and reason about the measured values in order to determine what behaviors the system should adopt in order to best operate in the detected environment [24]. Introducing dynamic adaptive capabilities into safety-critical systems would address several issues.

First, it could increase the fault-tolerance of safety-critical systems. In many safety-critical systems, if the system fails in an unexpected or unforeseen way, there is generally no prescribed method of handling this failure. Since DAS inherently include a degree of self-monitoring, provided the correct system properties are monitored, it would be possible to determine that the system has failed in the current configuration. Systems could also provide a fail-safe configuration that could be transitioned to in the event of a failure.

Second, it could increase the deploy-ability of safety-critical systems. Since DAS monitor and adapt to their environment, it would be possible to build a safety-critical system that could support multiple environments rather than building multiple systems, one per environment. Doing so would greatly reduce development and certification costs as only one, not multiple, system would be considered.

Finally, the inclusion of dynamic adaptivity in safety-critical software could allow it to respond to unexpected situations in a manner that prevents loss of life. Alaska Air-

lines Flight 261 was a flight that crashed killing 88. The crash was caused by a screw sheering in such a way that the flight control module which controls pitch was not oriented correctly [27]. If this system had been able to correctly detect its current orientation, the software of the flight control module could have reoriented itself, allowing the plane to land safely.

Despite the benefits possible, adaptive behavior significantly hinders safety analysis and certification. At the current time, the safety analysis methods used for adaptive systems are formal methods [28, 15], which are generally advocated to be used in addition to reasoning-based safety analysis not standalone [16, 3]. Formal methods are generally more difficult to understand, modify and maintain [16, 3], and mathematically-based reasoning techniques, like Software Fault Tree Analysis (SFTA) [19], are easier to construct while still maintaining the mathematical rigor of formal methods. In our work, we use reasoning-based techniques over formal methods due to the difficulty involved in building a correct mathematical specification of the system needed for evaluation by formal methods. The specification requires specialized training in mathematical proofs / specification to build and, like any large development artifact, can be prone to errors. Reasoning based techniques have no such constraint.

In this position paper, we explore adding dynamic adaptive capabilities to safety-critical systems. The primary contributions of this paper are as follows:

- We consider the difficulties of testing dynamic adaptive software products due to the large number of possible run-time configurations.
- We consider the use of a partially-tested DAS in a safety-critical environment and how the use of monitoring and dynamic reconfiguration could be used as assurance for the partially tested system.
- We consider the use of the Architecture Analysis & Design Language (AADL) [12] as a method of designing, analyzing and modeling DAS.

For each item, we will discuss our in-progress research that provides potential solutions, and we also introduce questions and concerns arising from each topic that are open work.

The remainder of this paper is structured as follows. Section 2 contains the background information for this work, and section 3 contains issues that we have found preventing adaptive designs from being used in safety-critical systems as well as work we are conducting towards providing a solution for the issue. Section 4 contains related work, and finally, section 5 concludes this position paper.

2. BACKGROUND

We now provide the information necessary for understanding the work that we present. We provide overviews of safety-critical systems and AADL. We also give an overview of variation management.

2.1 Safety-Critical Systems

Safety-critical systems are systems where failure can result in serious injury, loss of life or property / environmental damage. These devices have been employed in the domains of aviation and transportation as well as numerous other domains including medical and nuclear energy [17]. Such systems are becoming more commonplace, particularly as humans are increasingly relying on machines for situations, such as autonomous vehicles, where humans cannot always react quickly enough.

Safety-critical systems are designed and developed differently than traditional software systems. They require extensive analysis, design and testing which all contribute assets to the certification process, and an assurance case to show that potential failures have been sufficiently mitigated. The necessity of extensive work both initially in design and later in testing is owed to the fact that the certification process is, in general, expensive and long. Any errors found during the certification process require that the software be corrected and resubmitted starting the certification process over from the beginning.

Safety-critical systems have different development concerns than traditional software, especially since human lives are at risk. Unintended behavior must be prevented from emerging when separately developed components are brought together. An example of such emergent behavior can be seen in the failure of the London ambulance dispatch system in 1992. The deployed system worked initially, but a combination of interactions caused the system to fail increasingly frequently throughout the day which resulted in service delays and up to 46 deaths [20]. In addition to extra work during development, additional work is needed during requirements analysis to prevent incorrect assumptions about the software. An example of this type of failure is the Mars Climate Orbiter which crashed because different units of measurement were employed by the ground and on-board control systems [17]. These are only a few examples of the additional concerns while developing safety-critical software.

2.2 Architecture Analysis & Design Language

AADL is an architecture representation language and analysis tool designed for embedded system technologies. It is used to represent both hardware and software architectures [12] and is an approved standard of the SAE [10]. AADL is a highly extensible language structured as a set of annexes around a core language. As an example, the behavior annex provides the ability to specify the runtime behavior of a component. Other annexes provide support for verification, error specification and requirements among others. AADL's Integrated Development Environment (IDE), OSATE, also provides tools for analyzing and verifying the architecture specification. Some tools evaluate only the architecture specification while others utilize information from the annexes to aid analysis.

AADL has been used as a method of representing SPL, a methodology used to build DAS [2], as its rich typing and high extensibility support the intricacies involved in modeling SPL. One such feature is that of AADL's *extends* keyword which can be used to represent the variation / variation point structure used in product lines. *Extends*, an example of which is shown in figure 1, is used to denote a system that inherits properties from a parent system. The child system will receive the *features* (inputs and outputs), *subcomponents*, *connections* and annexes of the parent. Children are allowed to add to or to refine the parent's structure, but they are not allowed override the structure of the parent. This provides an ideal representation for SPL variation points / variations as variation points are representations of "holes"

to be filled. The "hole" is normally represented as an object that contains input / output types or other structural information. Variants extend the variation point's "contract" to ensure that they can be properly instantiated in the variation point.

```
system my_variation_point
end my_variation_point;

system my_variation extends my_variation_point
end my_variation;
```

Figure 1: AADL *Extends* Example

2.3 Variation Points & Variations

SPL is a development methodology that decreases the cost associated with development by allowing developers to separate out a set of common assets that are shared across a family of products. It allows developers to more intensely focus on product-specific issues rather than focusing on shared assets [26]. The shared assets are morphed into what is known as a product family architecture. This architecture represents all common functionality to the entire suite of products, but the family architecture contains "holes" known as variation points. In order for a new product to be created within the context of the product line, each variation point must be substituted with an implemented variant [26]. This instantiation of a variation point can happen at any time during the product life-cycle, including runtime. Variations and variation points are also used by DPSL, where instantiation of a variation point does not necessarily lead to a new product but could lead to new run-time configuration.

Variation points possess two primary properties: state and variant addition state [26]. We add to this list the variation point's type [5]. The state of a variant can be one of three mutually exclusive values:

- **Implicit** - These are variation points that have not yet been specifically created in the architecture. They represent uncertainty or variability that has not yet been decided and has not yet been explicitly left as a decision to be bound by the product. These types of variation points tend to be created early in the project life-cycle and are migrated to designed variation points as the project progresses.
- **Designed** - These are variation points where the design decision has been deliberately left unanswered.
- **Bound** - A variation point that has been replaced by a variant.

[26] The second property, variant addition state, deals with whether it is possible to add new variants to the system. This property can assume one of two values:

- **Open** - An open variation point is one that can still have additional variants added to its list of potential variants.
- **Closed** - A closed variation point cannot have any further variants added to its list of potential variants.

[26] The final property is the variant type. This property can also assume one of two values:

- **Static** - A static variation point is a variation that, once bound, cannot be rebound. This binding occurs as the result of a decision outside the system.
- **Dynamic** - A dynamic variation point is a variation point that can be rebound, even after binding. This binding usually, but not always, occurs as a result of a decision made autonomously by the system itself.

[5] Each of these properties can be represented in the feature tree of the product line. With AADL, it is possible to create a property set that will also allow these properties to be expressed as part of the AADL model which in turn allows variation point properties to be evaluated during architecture analysis.

3. ISSUES

Our motivation for undertaking this work is to explore how safety-critical systems can benefit from the work of the dynamic adaptive community. The intention is to build safety-critical systems that are capable of operating optimally in a wide range of situations, especially environments that threaten successful execution. To provide a motivating example, consider that some autonomous vehicles rely on external services provided by the IoT. These external services could provide non-safety-critical features like entertainment, but they could also provide safety-critical features such as accident alerts, road condition warnings and navigation. Each year as new vehicles are released, the new vehicles will likely rely on new, upgraded versions of each service that may not yet be available in all areas. Much as cell phones are capable of adapting to weak cellular signals, it is desirable for these vehicles to be both capable of detecting and adapting to the absence of the services on which they rely.

Despite the benefits that DAS offer to safety-critical systems, there are concerns that hinder the composition of the disciplines. In particular, we present three concerns:

- What constitutes "sufficient pre-deployment testing" of a safety-critical DAS? (discussed in section 3.1)
- What is the nature of the failures that can arise from executing untested configurations containing faults and how can they be mitigated? (discussed in section 3.2)
- How can safety-criticality be incorporated into the design of DAS? (discussed in section 3.3)

For each of these concerns, we will discuss the concern then provide work our group is undertaking that we postulate will provide a resolution to the stated concern.

Throughout our discussion, we will use standard terminology from SPL. We use *variation point* to designate a decision that has been deliberately left undecided. We use the term *variant* to represent a candidate decision that can be bound to, or swapped into, a variation point. For the purposes of this discussion, we assume that the variation point is dynamic and can be bound and rebound at run-time, unless explicitly stated otherwise. We will use the term *context* to mean the current operating environment, or the current set of values measured by the system. Finally, we use *configuration* to define the current bindings for the set of variation points. Since we are dealing with dynamic variation points, we also assume, unless explicitly stated otherwise, that the

configuration can be changed autonomously by the system at run-time.

3.1 Pre-deployment Testing of Safety-Critical DAS

A single dynamically adaptive system defines multiple product configurations and therefore can be represented and managed using SPL methods [2]. As such they share many of the same concerns with regards to testability. The shared concerns include:

- **Large Number of Test Cases** - As additional potential variants for the system's variation points are identified, the number of possible run-time configurations increases combinatorially.
- **Reusable Components** - Variants could be reused between adaptive systems, but which artifacts of the variants (testing results, test cases, certification assets) are to be shared and how much they will reduce the effort in testing the new system is not well understood.
- **Variability** - SPL, and subsequently DAS, have two types of variation points, static and dynamic. How testing the various types of variation points should be handled or if they should be handled differently is not known.

[9] Our work focuses primarily on the first concern.

For a given safety-critical SPL, one approach to ensuring sufficient test coverage would be to automate the testing of all possible combinations. While this method may be acceptable in some cases, there also exist cases where this is not an option. One such example would be a situation where there is not enough time for the test suite to run before the system is to be deployed. In such situations, it is necessary to determine which tests will capture the largest set of faults in the system. One method of reducing the number of potential test cases would involve ranking the configurations of the system by the likelihood that they will be used. After ranking, the top k configurations could be selected for testing based on how much time is available for testing and how long it takes to test a single configuration. This method was employed by [1].

Our group is currently using an additional complementary technique based on the error ontology given in the Error Annex to AADL [6]. We produce models of each component of the system, and we augment each model with the ontology of errors that can propagate from that component. When a configuration of the system is composed from the models of each component, the different error propagations of each configuration can be examined. We postulate that this information can be leveraged in order to determine which tests should be run against a configuration.

Consider that we have a variation point, A, with two variants, *A'* and *A"*. A can propagate errors *BadValue* and *MutexError*. This means that the variant could produce an incorrect value or that it could produce an error due to threads improperly using a mutex. *A"* can propagate errors *BadValue*. It does not propagate *MutexError* as it, unlike *A'*, does not use multiple threads. Based on this information, we determine that it is not necessary to run multi-threading or mutex tests against configurations of the system in which *A"* has been swapped in for variation point A while it is necessary to run tests for multi-threading / mutexes against configurations that use *A'*. By examining the error propagations, we can not only limit our testing to the configurations that are most likely to be used, but we can also limit our tests based on the types of errors that each configuration can produce. Our hypothesis is this will allow us to test a greater number of configurations than if we ran every test against a configuration.

3.2 Mitigation of Failures at Run-time

Assuming that we have a limited testing budget and a sufficiently large system, it is probable that an untested configuration containing latent faults will be entered into at run-time. Due to the configuration's not being tested, safety-critical DAS should have some method of both detecting and handling execution-time failures that could occur as a result of the faults.

A primary strategy is that of performing a series of execution-time tests just before a configuration is deployed, as is done by [21]. Components provide a series of dependencies and provisions, and the tester ensures that all dependencies are satisfied by some component. Another possibility is that each component contains test cases, and these cases are used to perform the pre-deployment testing. In general, a configuration that fails to pass the run-time tests will not be deployed. Pre-deployment testing can help mitigate the probability of failure, but testing alone cannot guarantee that failure will not occur.

Since tests alone cannot guarantee that failure will not occur, it is necessary to have some method of handling failures if and when they occur. [22] proposed a method of maintaining a list of potential alternatives for a given variant in a configuration. If the variant fails, each alternative can be attempted until all possibilities have been exhausted or an alternative resolves the failure. A natural concern is what should happen if all possibilities have been attempted but the failure persists. In some domains, it would be possible to halt the system, but in safety-critical systems, this is rarely possible. Another concern is how the occurrence of a failure affects the hard real-time requirements of a system.

Our proposed method is to provide a failure configuration that is entered into immediately upon failure. This configuration will spawn two co-operative tasks that execute side by side. The first task provides basic functionality so that the system can continue to receive and transmit information meeting its hard real-time requirements. The second task attempts to resolve the failure by changing the configuration. In our current work, the change is to simply rollback to the previous configuration. We would like to incorporate the alternative list used by [22] rolling back only once all alternatives have been tried. Our approach currently raises several unresolved concerns. First, detecting failure occurrence, loading the failure configuration and switching into the failure configuration is not an instantaneous process. For systems with real-time requirements in the millisecond range, it is possible that our process could still cause the system to fail to satisfy its requirements. Second, our reason for wanting to incorporate the list of alternatives stems from the fact that rolling back to the previous configuration can cause the system to re-enter the failing configuration. This happens when the rollback occurs but the context has not changed. The previous configuration detects that another configuration is better suited to handle the context, the failing configuration, and switches. An idea currently

being explored is the concept of banning configurations that have failed as is done by [13], discussed later.

Even though run-time testing cannot prevent failure, it still offers a potential mitigation that makes it less likely that we will need to rely on run-time failure resolution. The manner in which the execution-time tests are performed, however, do raise concerns for safety-critical systems. Unlike [21], it is rarely possible for a safety-critical system to halt execution in order to perform run-time tests. Once again, considering hard real-time requirements, we currently have the reconfiguration process run concurrently alongside the current configuration. This has several benefits. First, the system can continue normal operation while the new to-be-deployed configuration is tested and validated. Second, if the to-be-deployed configuration fails validation, the configuration process can simply terminate with no additional actions. The current configuration that has been running alongside the validation process may be able to continue uninterrupted. If the to-be-deployed configuration succeeds, then and only then is the current configuration interrupted and the new configuration swapped in. There are also several drawbacks to this approach. First, as previously stated, switching contexts is not instantaneous. For systems with real-time requirements in the millisecond range, uninterrupted execution may not be possible. Second, additional hardware is needed to allow concurrent execution of configuration testing on smaller embedded devices. Finally, what is necessary to conduct run-time testing varies depending on the test to be executed. Our group has also considered the possibility of representing the entire architecture as this would allow complex analysis tasks like flow, dependency and latency analysis to be performed on the system at runtime. This would be particularly beneficial since the system may have to consider never-before-seen variants. However, safety-critical systems can be small embedded devices with limited memory constraints. It may not be possible to load a representation of the entire architecture or the entire feature model into memory all at once for analysis. This would require multiple read operations from storage increasing the time required for the run-time testing of the reconfiguration process. Even if size were not a concern, which types of tests to perform is. It is likely that the reconfiguration process will not have enough budget or resources to execute all of the analysis tests.

Our group has also considered the possibility of the system context as a source of failure. Once again consider an autonomous vehicle. The vehicle is traveling on a sunny, cloud-free day and, using sensors on the windshield / underbody, determines that the traction control system should be configured to support mostly dry road conditions. As the vehicle is traveling along it encounters wet roadways due to a fire hydrant that is in the process of being depressurized. The car under-body detects wet road conditions, but the windshield sensor determines that conditions are still sunny so the current configuration is not changed. As a result the car begins to slide on the wet roadway possibly ending in a collision. In the example, an untested configuration was not the source of the fault; the combination of the context and configuration resulted in a fault. Our work on providing a minimal configuration for fault situations provides one potential solution to this concern, but our group is also considering other avenues. We are currently investigating an extension of [13] so that a system's failure or success in a particular context is recorded and, at a later point in time, can be used to determine if a configuration should be deployed in that context again. Continuing with the example, we encounter wet roads but sun again. The vehicle can leverage the collected history to determine that dry road traction control should not be used as last time it resulted in a failure. This history of configurations and the contexts they have failed in could also be shared with other vehicles allowing them to "learn" and become more reliable with time.

3.3 Safety-Critical DAS Design Techniques

The extensive analysis required by safety-critical systems needs special tooling for modeling and design. Current architecture tools, such as AADL and UML, do not explicitly support dynamic structural adaptation, but they possess several features that could be useful to the analysis and design of DAS.

AADL and its IDE, OSATE, have many features that would be beneficial to the modeling, analysis and design of DAS. Their support of annexes would allow for tooling to be built for the modeling of DAS that use structural adaptation, and our group is currently considering the development of an annex specific to that goal. The current features we feel would be of use to the DAS community, and that have been particularly useful in the work we have undertaken, would be the features used to model the unique features of SPL [11], modes, behavior and error flows.

```
system receiver
 features
   inevent: in event port;
 modes
   nominal: initial mode;
   recovery: mode;
end receiver;

system implementation receiver.i
 subcomponents
   sub1: system sub_receiver.i in modes (nominal,recovery);
   sub2: system sub_receiver.i in modes (recovery);
 internal features
   triggerrecovery: event;
   triggernormal: event;
 modes
   nominal-[triggerrecovery]->recovery;
   recovery-[triggernormal]->nominal;
end receiver.id;
```

Figure 2: AADL *Modes* Example

AADL allows developers to specify state transitions using a Finite State Machine (FSM) notation, and the states specified can be used to enable or disable components as well as to drive analysis. As an example, consider figure 2. The system receiver has two modes *nominal* and *recovery*. It also has two sub-components *sub1* and *sub2*. The system starts in its initial mode, *nominal*, and continues execution until a *triggerrecovery* event occurs. Following the rules specified under *modes* in *receiver.i*, the system transitions to the *recovery* mode. Under normal operation only the *sub1* subcomponent is available to the system, but in *recovery* mode *sub2* can be used as well. Both *sub1* and *sub2* are usable until the system returns to *nominal* mode via the *trigger-*

nominal event.

The mode can also be used to drive decision modeling as both the error and behavior annexes can reference the current mode. The behavior annex provided by AADL allows the behavior of the system or a sub-component to be specified and modeled [7], and there are some commercial tools available that can utilize this annex when modeling or simulating the system represented by the architecture [8]. An error model, defined using the AADL error annex, allows developers to specify how errors propagate throughout the system. The errors and their types can be specified as well as which errors the component can handle. Errors that cannot be handled by a component are propagated further up the composition hierarchy until they reach a component that handles them [18].

Since the behavior annex is used to model dynamic behavior in a system, the modes could be leveraged as a method of allowing certain types of adaptation only when the system is in a specific mode. AADL could also be used to explicitly model and trace the context of the system through its extensible type interfaces. The context could then also be referenced by the error and behavior annexes. Commercial tools and analysis interfaces in OSATE would, as a result, be able to directly predict the adaptive system's behavior. This can be used as an additional method of testing the adaptive system, and it offers high-level insight into the operation of the dynamic software.

4. RELATED WORK

We now cover work with similar focus to ours identifying areas of overlap. We also provide how our work differs and if our work could benefit from incorporation of ideas expressed by these similar ideas.

[1] provides a method of automatically calculating the different configurations that a system can assume. The reliability of the system is then evaluated using a Markov chain. Our work differs in that we explicitly assume that even if the number of possible configurations is determinable, the number of configurations will be too large to test in a reasonable amount of time. It is also the case that a reliable system does not necessarily mean that the system is safe, however a safe system cannot be unreliable. Our work uses analysis of the system's architecture to limit the number of tests that need to be executed against the system.

[15] provides a method, using DCCA, of proving that an adaptive system is more dependable than a conventional system. The check is based on temporal logic; however, the method is not a direct proof of safety, but it could be used to contribute to the case for safety. The method takes into account that a failure could be resolved by the reconfiguration process, but it doesn't take into account that the failure could also be re-introduced by the reconfiguration process. This method is complimentary to our own. It could be used as an additional safety case or as the basis for developing an analysis method tailored to dynamic adaptive systems.

[4] presents an overview that describes the need for run-time metric measurement in adaptive systems, and [25] provides a method of measuring specific quality attributes. This method could be used as an assurance that adaptation correctly places safety as a high-priority. Its monitoring capabilities could be extended to support error monitoring during run-time. The process, as defined, does not consider safety or run-time monitoring, but the goals of seeking to maximize specific properties of software are parallel to our own. We would like to, in future work, adapt this method to execute at run-time on an adaptive system so that attributes of the system can be measured as they change.

Our work differs from the above in that we explicitly consider safety-critical systems unlike [4] and [25]. We differ from [15] and [1] in that we believe the reconfiguration process needs additional safeguards that will prevent faulty configurations that have been resolved from being redeployed into the same context in which they failed. [15]'s goal is very similar to ours but it, as mentioned, lacks the focus on additional reconfiguration guards. Each of the ideas expressed however could eventually be incorporated into our work to strengthen the assurance case for system safety.

5. CONCLUSION

In this position paper, we explored several concerns that have been raised by our research considering incorporating dynamic adaptivity into safety-critical systems. Our interest in applying dynamic adaptivity stems from the unique properties and behaviors that DAS offer in regards to designing and deploying safety-critical systems. One such possibility explored in this paper is the utilization of dynamic adaptivity to increase a safety-critical system's reliability.

Even though DAS offer many exciting possibilities for safety-critical software, there are many unanswered questions. Testing of DAS, DAS representation and how DAS handle faults that occur during run-time all raise concerns that must be either mitigated or resolved. In future work, we plan to further explore how DAS can be certified for safety-critical environments. We are currently conducting research to determine which, if any, current safety analysis techniques could be adapted to use dynamic adaptive software. There are several possibilities which we are still in the process of exploring. We also plan to further explore what role context plays when deciding if a configuration that has previously failed should be deployed again, and we are investigating run-time testing of safety-critical DAS to mitigate failures.

6. ACKNOWLEDGEMENTS

The work of the authors was funded by the National Science Foundation (NSF) grant # 2008912. We also thank those that have reviewed our work offering recommendations for improvement.

7. REFERENCES

[1] R. Adler, M. Förster, and M. Trapp. Determining configuration probabilities of safety-critical adaptive systems. In *Advanced Information Networking and Applications Workshops, 2007, AINAW'07. 21st International Conference on*, volume 2, pages 548–555. IEEE, 2007.

[2] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *SPLC (2)*, pages 23–32, 2008.

[3] J. Bowen and V. Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8(4):189–209, 1993.

[4] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM*, 55(9):69–77, 2012.

[5] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[6] J. Delange and P. Feiler. Architecture fault modeling with the aadl error-model annex. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 361–368. IEEE, 2014.

[7] P. Dissaux, J. Bodeveix, M. Filali, P. Gaufillet, and F. Vernadat. Aadl behavioral annex. In *Proceedings of DASIA conference, Berlin*, 2006.

[8] Ellidiss.com. Aadl inspector, 2016.

[9] E. Engström and P. Runeson. Software product line testing–a systematic mapping study. *Information and Software Technology*, 53(1):2–13, 2011.

[10] H. Feiler, B. Lewis, and S. Vestal. The SAE architecture analysis and design language (AADL) standard. In *IEEE RTAS Workshop*, 2003.

[11] P. Feiler. Product line modeling with aadl, 2005.

[12] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical report, DTIC Document, 2006.

[13] B. J. Garvin, M. B. Cohen, and M. B. Dwyer. Using feature locality: can we leverage history to avoid failures during reconfiguration? In *Proceedings of the 8th workshop on Assurances for self-adaptive systems*, pages 24–33. ACM, 2011.

[14] M. Gerla, E.-K. Lee, G. Pau, and U. Lee. Internet of vehicles: From intelligent grid to autonomous cars and vehicular clouds. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 241–246. IEEE, 2014.

[15] M. Gudemann, F. Ortmeier, and W. Reif. Safety and dependability analysis of self-adaptive systems. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 177–184. IEEE, 2006.

[16] A. Hall. Seven myths of formal methods. *Software, IEEE*, 7(5):11–19, Sept 1990.

[17] J. C. Knight. Safety critical systems: challenges and directions. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 547–550. IEEE, 2002.

[18] B. Larson, J. Hatcliff, K. Fowler, and J. Delange. Illustrating the aadl error modeling annex (v.2) using a simple safety-critical medical device. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, pages 65–84, New York, NY, USA, 2013. ACM.

[19] N. G. Leveson and P. R. Harvey. Software fault tree analysis. *Journal of Systems and Software*, 3(2):173–181, 1983.

[20] E. Musick. The 1992 london ambulance service computer aided dispatch system failure.

[21] D. Niebuhr, A. Rausch, C. Klein, J. Reichmann, and R. Schmid. Achieving dependable component bindings in dynamic adaptive systems-a runtime testing approach. In *Self-Adaptive and Self-Organizing Systems, 2009. SASO'09. Third IEEE International Conference on*, pages 186–197. IEEE, 2009.

[22] K. Paulsson, M. Hübner, and J. Becker. Strategies to on-line failure recovery in self-adaptive systems based on dynamic and partial reconfiguration. In *Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on*, pages 288–291. IEEE, 2006.

[23] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.

[24] A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng. A taxonomy of uncertainty for dynamically adaptive systems. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '12, pages 99–108, Piscataway, NJ, USA, 2012. IEEE Press.

[25] G. Tamura, N. M. Villegas, H. A. Müller, J. P. Sousa, B. Becker, G. Karsai, S. Mankovskii, M. Pezzè, W. Schäfer, L. Tahvildari, et al. Towards practical runtime verification and validation of self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 108–132. Springer, 2013.

[26] J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on*, pages 45–54. IEEE, 2001.

[27] Wikipedia. Alaska airlines flight 261, 2016.

[28] J. Zhang, H. J. Goldsby, and B. H. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 161–172. ACM, 2009.