# Modeling and Testing a Family of Surgical Robots: An Experience Report

Niloofar Mansoor*
Dept of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68502-0115
nmansoor@cse.unl.edu

Jonathan A. Saddler*
Dept of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68502-0115
jsaddle@cse.unl.edu

Bruno Silva
Dept of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68502-0115
bsilva@cse.unl.edu

Hamid Bagheri
Dept of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68502-0115
hbagheri@cse.unl.edu

Myra B. Cohen
Dept of Computer Science & Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68502-0115
myra@cse.unl.edu

Shane Farritor
Dept of Mechanical & Materials Engineering
University of Nebraska-Lincoln
Lincoln, NE, 68502-0526
sfarritor@unl.edu

## ABSTRACT

Safety-critical applications often use dependability cases to validate that specified properties are invariant, or to demonstrate a counter example showing how that property might be violated. However, most dependability cases are written with a single product in mind. At the same time, software product lines (families of related software products) have been studied with the goal of modeling variability and commonality, and building family based techniques for both analysis and testing. However, there has been little work on building an end to end dependability case for a software product line (where a property is modeled, a counter example is found and then validated as a true positive via testing), and none that we know of in an emerging safety-critical domain, that of robotic surgery. In this paper, we study a family of surgical robots, that combine hardware and software, and are highly configurable, representing over 1300 unique robots. At the same time, they are considered safety-critical and should have associated dependability cases. We perform a case study to understand how we can bring together lightweight formal analysis, feature modeling, and testing to provide an end to end pipeline to find potential violations of important safety properties. In the process, we learned that there are some interesting and open challenges for the research community, which if solved will lead towards more dependable safety-critical cyber-physical systems.

## CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; **Formal software verification**; *Model-driven software engineering*;

## KEYWORDS

software product lines, Alloy, testing and analysis, cyber-physical systems

**ACM Reference Format:**
Niloofar Mansoor, Jonathan A. Saddler, Bruno Silva, Hamid Bagheri, Myra B. Cohen, and Shane Farritor. 2018. Modeling and Testing a Family of Surgical Robots: An Experience Report. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275534

## 1 INTRODUCTION

Modern surgery is moving towards the cyber-physical, using robots, controlled by surgeons from a console. These systems have tightly interwoven hardware-software controls with the hardware impacting which software is selected, and the software constraining the limits of the hardware. These robots can be configured in multiple ways, for different types of surgeries and can use different physical and virtual components. For instance, they can perform dissections, cautery, or sew an entry wound closed. They can be used for general, cardiac and/or gynecologic surgeries and on different types of patients. In essence, such systems can be viewed as a family of robots (i.e. a software product line) leading to hundreds if not thousands of possible configurations that may be used by a surgeon to satisfy his or her personal preferences. Yet, these systems are also safety-critical, and if they do not interact in a reliable and safe manner with the end user (the surgeon), this can lead to potentially severe consequences.

Current approaches to assuring safety-critical systems include using model-based techniques [5], formal methods [11, 13], architecture-based safety analysis [15], and techniques based on real world types and type checking [16]. The majority of these approaches, however, are subject to a common limitation: they are intended to ensure safety in a single system, but fail to be cognizant of the commonality and variability in the system, i.e., ensuring the dependability of a highly configurable safety-critical cyber-physical system. Other research has examined testing cyber-physical product lines [8]; however, that work does not address the safety-critical

*The first two authors contributed equally to this work.


*ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA*

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3275534

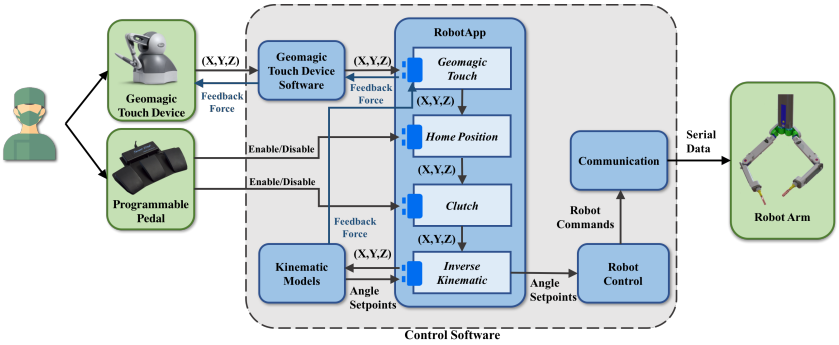


**Figure 1: System components involved in arm movement.**

aspects of the system. There has also been research on test generation for product lines using lightweight formal analyzers such as Alloy [7]; however, this thread of work again does not address safety-critical properties of the system. Last, Proctor et al. proposed an architecture description language extension for AADL for connected medical devices. This can reason about medical apps in general[14]; however, it does not directly provide support for our use case, dependability cases for families of safety-critical devices, such as surgical robots.

In this paper, we set out to understand the challenges and feasibility of assuring the safety of cyber-physical product lines. We use an open-source highly configurable research prototype for a miniature surgical robot, that formed the foundations for a commercialized product (in a proprietary format), currently under FDA approval. With its open source software, this family of surgical robot systems provides a valuable learning playground for us to explore. Motivated by the surgical robot family, we present an approach to ensuring the safety of a system family by constructing both feature and formal models and performing an automated analysis of the models, followed by a guided, yet narrowly scoped, testing phase on concrete family instances. Two essential elements that distinguish our approach from prior efforts at safety analysis are as follows:

- **Family-level reasoning:** By identifying commonality and variability in the system and explicitly modeling them in analyzable specification languages, we are able to perform family-wide reasoning that would be difficult to achieve using static analysis or testing. For example, our analysis can explore all possible systems in which a particular type of robot arm is being installed and check whether the use of that robot arm along with any other software/hardware components can lead to a violation of a safety property.
- **Guided testing on concrete instances:** While rigorous and exhaustive analyses of a formal, yet abstract, model of the system family can help pinpoint potential property violations, one more step is needed to confirm the identified violations are indeed realistic. In particular, we support the formal analysis with targeted testing of the concrete system to verify whether the identified property violation can result in practical issues. The counter examples produced by the formal reasoning are leveraged at this step to guide the testing on concrete family instances.

The contributions of this work are (1) an end-to-end case study to validate an important physical property for a family of surgical robots; (2) demonstration of a potential synergy between a lightweight formal approach and feature modeling techniques for a safety analysis of a family of a surgical robots; (3) a set of lessons learned and discussion of future directions for assuring cyber-physical product lines.

In the next section, we present the family of robots followed by our dependability case (Section 3). We then discuss our findings in Section 4 and end with conclusions and future work.

## 2 OVERVIEW OF THE SURGICAL ROBOTS FAMILY

The Advanced Surgical Technologies Laboratory at the University of Nebraska-Lincoln (UNL) is one of only a handful of institutes in the world developing *in vivo* surgical devices. The latest developments include miniature *in vivo* surgical robots for use in robotic laparo-endoscopic single-site (R-LESS) surgery procedures [3, 9]. These miniature surgical robots are small, do not need a dedicated customized surgical suite or infrastructure, have reusable disposable tools that are familiar to surgeons, and can be operated locally or remotely from a small console that includes haptic feedback and a screen that virtualizes robotic positioning.

The robots that have been developed include multiple modules and plugins for different types of hardware control. This includes different robot arms, some of which have haptic feedback, and some that do not. Different solvers control the physical movements of the arms, tool position tracking, simulation, video, voice communication, etc. The source code that controls the physical aspects of the robot includes both a physical and simulation environment. It is written in C# and is available as open-source [4]. We show the overall architecture of the robot system in Figure 1.

**Physical Components:** The primary device that is given to the surgeons to control the robot arm movements is a *Geomagic Touch Device*. There can be either one or two of these devices, depending on the number of robot arms. Touch is a motorized device that applies force feedback to the user's hand, allowing the surgeon to feel virtual objects and producing true to life touch sensations as the user manipulates the 3D objects on the screen [1].

The *programmable pedals* are assigned to clutch and home position functions to enable/disable these functions in the control software. The *robot arm* is the other physical component of the system, which has a number of motor controllers to which the software layer sends serial data to move the robotic arm. Each robot arm can have a number of joints, and each joint has an angle limit. Each robot arm is also connected to an end effector that can perform

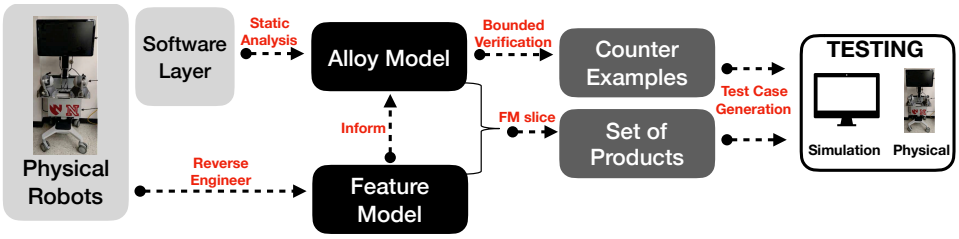


**Figure 2: Overview of our Process**

a specific task in surgery, such as shears for cutting or a cautery hook for use after cutting.

**Software components:** The *Geomagic Touch software* component provides a connection between the physical Geomagic Touch device and the software system. The system receives its coordinates from the Geomagic Touch endpoint and sends the coordinates to other components.

The *Kinematics* software component contains a set of kinematic models, which are specific to the hardware being used (i.e. the arm). They use inverse kinematic solvers for different arms of the robots. The solvers receive the coordinates from the *RobotApp* component, and calculate the joint angles for each joint of the robot arm, and send them back to the RobotApp component.

The *RobotApp* component, which contains a set of plugins, employs the Model-View-View-Model pattern. The use of this pattern facilitates the separation of the development of the graphical layout of the user interface from the development for the back-end logic of the application. A set of plugins are loaded when the software is used so that the robotic arms can be controlled.

The other components also interact with a set of plugins. For instance, the *Geomagic Touch* software component interacts with the *GeomagicTouch* plugin, which facilitates sending the coordinates to the other plugins, such as a *Solver* plugin. A solver plugin sends the coordinates to the correct solver for a chosen robot arm while receiving the joint angles from the *Kinematics* component. There are also other plugins in the system that manipulate the input in other different ways. Some of the plugins are necessary, such as *Clutch* and *HomePosition*, and some of them are only loaded for specific states or actions, such as *GrasperLimits*.

The *Robot Control* component is used to abstract a specific set of motors, control modules, and robot-specific parameters. It handles control and data services to discover, control, configure, and read motor control modules. The *Communication* component provides a mechanism that facilitates the robot-computer communication, supporting serial communication and sending the robot commands as serial data to the robot.

## 3 BUILDING A DEPENDABILITY CASE

We set out to build a dependability case for the family of robots with the aim of understanding the feasibility and challenges.

We selected a critical property of one specific safety feature of the robot that is important in practice. It is a property that ensures the safety of the patient by guaranteeing the surgeon is always aware of the position of the arm within the patient. If violated, the implications are twofold. First, it means that the arm may extend into unsafe regions of the patient cavity. Second, if the arm is extended to its maximum position and torque continues, this could potentially lead to a hardware failure. We do not specify the manner in which the surgeon is notified (i.e. via haptic feedback or via visual messaging). More specifically the property being enforced is:

**Arm movement safety property:** *During the surgery procedure, as the surgeon moves the control device, the actual position of the robot arm should be the same position that the surgeon articulates in the control workspace and he/she should be notified if the arm is pushed outside of its physical range.*

This property is enforced by a robot controller system, consisting of hardware and software components, which monitors and drives the system's physical components. Our dependability case spans the controller system as well as the physical modules involved in the arm movement.

### 3.1 Process

Figure 2 shows an overview of the process that we used to build our dependability case. We incorporate both a lightweight formal analysis using Alloy. At the same time, we reverse engineered a feature model and superimposed these two together to identify sets of products that potentially violate the specified property. We then applied testing using the simulator (we did not implement physical hardware testing at this stage) to validate the counter examples found using the Alloy models. The last step would be to instantiate and validate these test cases on the physical system. We leave that as future work.

**Alloy Models.** We now describe a formal model for the surgical robots family in Alloy [6], a lightweight formal specification language based on a first-order relational logic, with an analysis engine that performs bounded verification of models. There are three main reasons that motivate our choice of Alloy for this study. First, its flexible core, backed with logical and relational operators, makes Alloy an appropriate language for declarative specification of systems and properties to be checked (i.e., assertions). Second, its effective module system allows us to split the overall, complicated family model among several tractable modules. Such a well-structured module system not only facilitates modeling and integrating different aspects of the system, but also enables compositional analysis of the system components. Third, its backend analysis engine, i.e., the Alloy Analyzer, provides an automated analysis for checking assertions and generating counter examples.

To carry out the analysis, we start by defining a common Alloy module that models the fundamentals for the family of surgical robots and the constraints that every family instance must obey. Technically speaking, this module can be considered as a meta-model for the family of surgical robots. Listing 1 partially outlines the meta-model module. The complete version of all Alloy models that appear in this paper are available at the project website [12]. The essential element types are defined as top-level Alloy

```
abstract sig GeomagicTouch {
  input: one Coordinate,
  force: HapticFeedback,
}
abstract sig RobotApp {
  includes: some Plugin
}
abstract sig RobotControl{
  output: set ArmAngle
}
abstract sig SolverFamily{
  calls: one KinematicModel
}
abstract sig KinematicModel{
  solverResult: Coordinate -> ArmAngle
}
abstract sig ArmAngle {}
abstract sig Coordinate {}
abstract sig ArmType {
  anglelimit: set ArmAngle, //set of all the arm angles that are
      less than limit
  inverseKSolver: one KinematicModel
}
abstract sig RobotArm{
  armside: one Side,
  armModel: one ArmType,
  effectorType: one EffectorType
}
// outputs should be in the range of solverResult
fact OutputConstraint {
  all o: RobotControl.output | one a: getArmAngle[KinematicModel
      ,Coordinate] | o = a
}
// return the angles produced from a specific coordinate
fun getArmAngle[s: KinematicModel, c: Coordinate]: one ArmAngle
    {
  s.solverResult[c]
}
// for each coordinate, there exists a set of angle in the
    solver result
fact AngleCalculation{
  all c: Coordinate | some a: ArmAngle, s : KinematicModel| c->a
      in s.solverResult
}
```

**Listing 1: Excerpts from an Alloy specification for the family of surgical robots.**

signatures: GeomagicTouch, SolverFamily, RobotControl, KinematicModel, ArmAngle, Coordinate, ArmType, RobotArm. Note that these signatures are defined as abstract, meaning that they cannot have an instance object without explicitly extending them. Containment relations (e.g., between GeomagicTouch and Coordinate) are defined as Alloy relations. The fact OutputConstraint specifies that the RobotControl output ArmAngles should be produced by a solver in the system, and the fact AngleCalculation specifies that the solver transforms each coordinate to a set of arm angles. To create individual family instances, we extract information about each specific system and extend its corresponding element type in the meta-model.

We then state the property that the model is expected to satisfy as an Alloy assertion. This property is formally specified as Alloy assertion ArmAngleCorrect in Listing 2. Predicate ProducedFeedback describes when the force should be produced and when the HapticFeedback should be enabled. The assertion then relies on the ProducedFeedback predicate to state that all the output angles produced by the solver fall into the set of angle limits. The Alloy Analyzer then explores all possible behaviors of the system and identifies a counter example, if any, that corresponds to a violation of the assertion. The analysis is exhaustive but bounded up to a user-specified scope on the size of the element types.

```
pred ProduceFeedback[output : RobotControl.output] {
  output not in ArmType.anglelimit
  some notification : GeomagicTouch.force | notification =
      HapticsEnabled
}
// assert if the arm angle is in the set of armangle limit
assert ArmAngleCorrect {
all a: RobotControl.output | a not in ArmType.anglelimit
implies ProduceFeedback[a]
}
```

**Listing 2: Assertion on the arm movement safety property.**

**Feature Models.** We conducted a series of interviews with the robot developers focusing on retrieving domain knowledge. We lacked documentation on how the family was constructed. Therefore, we needed to understand the necessary and optional components of each robot, extract constraints and dependencies and map this to features. We used FeatureIDE as our tool for creating the final model, which allowed us to reason about slices of the product line[2].

From interviews, we learned the robot is to be a combination of two sets of configurable hardware components, namely arm types and effectors on the ends, and configurable software components. The software components are collectively called *plugins*, an array of plug and play configurable elements that can be used interchangeably to drive all 15 arm types and 4 effectors in specific ways. We describe our findings in more details in the next section.

**Testing.** Our approach for testing the surgical software relies on Microsoft CodedUI [10] plugin, a tool for testing user interfaces. It is capable of generating test cases based on manual interactions with the GUI. It can replay the tests, though it is not able to reverse engineer the interface to create a model of the system. CodedUI generates test cases automatically, but the generated code is tightly coupled, and if modifications are made, they will be discarded after building the project. Therefore, there is a need to extract the most relevant pieces of code, such as how to navigate between interfaces, the input values, and to verify assertions. We have extracted the code generated by CodedUI into an auxiliary class and refactored it, creating a class encapsulating the most important functionality of a test case, which is then used as a template. Individual robot classes can call this class, and it will perform the following steps: (1) Load configuration; (2) Go to solver plugin and select arm type, Go to the controller and input values to move the arm; (3) Go to the solver and verify the output. With all this information, it is then possible to generate a replayable test case for individual robots, as they will follow the same steps, only varying in the solver, type of robot arm and input values.

## 3.2 Results

We present our results and describe our challenges for each part of the process next.

**Finding Alloy Counter Examples.** As we built the Alloy model to search for a counter example, we used guidance primarily from static analysis. In order to cover the space of products of this robotic system, we needed to develop different models for each different robotic arm. This resulted in 15 Alloy models. The necessary features for the Alloy models include the *Arm Type*, *Solver*, *Geomatic Touch*, *Haptic Feedback*, and two plugins created to manipulate inputs from Geomagic Touch, named *Clutch* and *HomePosition*. Each

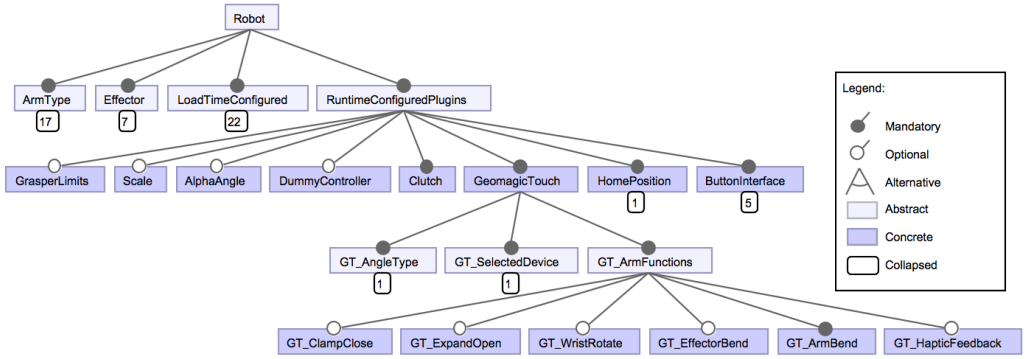


**Figure 3: Feature Model: Runtime Configurations**

model actually represents 88 different products from the robot family, rather than a single robot. However, this fact was not obvious as we built the analysis.

Since the undefined features were not part of the static analysis and did not contribute to the counter example, they do not appear in the Alloy model. However, we cannot be sure that the analysis is precise and leaving out some features may in fact mean that we have over or under approximated the existence of the counter examples (see our discussion below in testing). We did find a counter example for each of the models that did not include the Haptic Feedback feature. The robots that do use Haptic Feedback, do not lead to this counter example – i.e. the haptics feature of the system provides physical feedback to the surgeon anytime he or she tries to move the arm beyond its maximum range. We next discuss the results of the feature modeling and its mapping back to these counter examples.

**Feature Model.** Figures 3 and 4 show the feature model that we developed from two different perspectives. The full feature model is too large to show, so we have elided some features in each figure. Our full feature model (in an XML format) can be found on our project website [12]. In total (with cross-tree constraints) there are 1,320 potential surgical robots supported by this system. Figure 3 shows the high level features (*Arm Type*, *Effector*, *Load Time Configuration Options*, *Runtime Configuration Plugins*). In this figure we focus on the Runtime configuration plugins, in particular we show the branch of the feature model that includes the Haptic Feedback (last leaf on right).

Figure 4 shows the breakout for the ArmType and Effectors. The Arm type was further broken down during modeling because the developers pointed out that only 4 arm types are currently in active use. The other 11 are physical arms that are no longer used. However, since this distinction is based solely on domain knowledge and discussion with developers, it is not reflected in the Alloy models. For the Alloy models, all 15 arm types were modeled because the code is still active and discovered during static analysis.

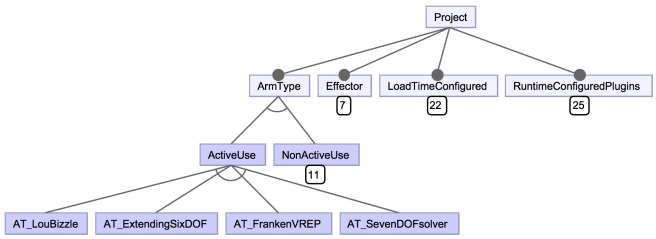


**Figure 4: Feature Model: Arms**

The feature model also has multiple cross tree constraints (not shown). These were determined via both discussion with the developers and by studying the code and configuration panels as selections are made. This was a challenging and iterative part of the process. It turns out that there is a highly constrained hierarchy between the hardware and software. Each arm type uses a single solver and each arm type either has haptic feedback or not. Other constraints include physical limits of the graspers, for instance. Most of these are hard coded into the software which means when any arm type is selected in FeatureIDE, we immediately have a small slice of the product containing only 88 of the 1,320 products. However, there are still 88 products that must be tested for each counter example if we are to confirm the existence of the faulty property. We discuss this next.

**Testing.** Five of the robot arms led to the counter example (FiveDOFSolver, FourDOF_needle, FrankenVREP, SevenDOFSolver and TomBot). To validate that these are not exhibiting false positives we built concrete test cases for each and observed the output. A failing test case shows that the arm location stays fixed at the same point once it is pushed out of range. A correct behavior shows a negative value in simulation when this occurs. We confirmed this by also testing the robots that did not exhibit the counter example.

Our first problem for testing stemmed from the fact that each of the Alloy configurations represents a set of robots (88 robots). We used the robot simulation mode for testing, however, the simulator does not capture some of the hardware components that lead to the larger number of robots. For instance, there are five different effectors that provide physical movements such as shearing, cautery, grasping, etc. These are related to the robot hand, which sits below the arm, and are not part of the simulator, and do not impact the solver output which is needed for the counter example to change.

We, therefore, ignored the features that do not impact the arm extension and/or impact whether or not the feedback is produced and tested only a single instance for each set of 88 products. This created a savings for us in terms of number of tests, however, the validity of this approach is dependent on the quality of our static analysis. The features that we were not able to capture in our simulation include, Arm side (Left or Right), Effector Type (5 different effectors) and specific modules to move the hand which are related

to the effectors (Clamp Close, Expand Open, Wrist Rotate, Effector Bend). The behavior of Grasper Limit and Scale plugins is not captured in our simulation of the system either, as they do not affect the output angles of the robot arm.

For the five robots that we were able to simulate, we selected a range of input values/angles on the console. As is common with configurable software, the configuration layer is orthogonal to the input layer. We did not have an automated generation tool. We selected values from a range that we expected would push the robot beyond a valid extension point (i.e. we used domain knowledge to help us find the important boundary values). Using this approach we were able to confirm that the counter examples do exist and the robot can be pushed outside of its limit with no feedback returned. As the robot goes out of range, in the systems without haptic feedback, the arm simply stops moving and records the same position over and over again once it reaches its limit.

Interestingly one robot, TomBot, printed a message to the debug console telling the developer that the arm was out of range. Theoretically, this could be passed to the physician console, but it is not propagated, so this information is lost when the robot is used outside of the debugging environment.

## 4 LESSONS LEARNED

We present our lessons learned next.

- **Architecture Plays a Large Role and can Help Analysis.** The way a system is designed and implemented has a significant impact in conducting a safety analysis. While dependencies among the various robot software components and the external components made it challenging to get the software running and working, its modular, plug-in-based nature helped us achieve a clear understanding of the system and the event flow between various components, which in turn facilitates the process of creating the dependability case.
- **Developers Should Consider the Family of Products.** One of the challenges we faced in concretizing counter examples and validating them was the unavailability of the configuration files for the entire surgical robot family. We only had access to the configurations for a small subset of robot instances that were currently being used by the engineers working with the system. To check the property for the rest of the robotic arms, we needed to create new configuration files which involved a tedious process of loading and validating each of necessary plugins for a particular arm.
- **We Need Methods to Map Feature Models to Alloy.** Our two views of the family of robots (Alloy and Feature models) differed in their granularity and focus. The feature model included both hardware and software and had some arbitrary divisions (e.g. the arm types), where as the Alloy model contained only the code-based features that led to the counter example. However, together they tell the full story of our robot and its potential safety properties. New methods are needed to merge these disparate models together.

## 5 CONCLUSIONS AND FUTURE WORK

In this paper we presented an experience report working with a cyber-physical safety-critical software product line, a robotic surgery system. We used both lightweight formal analysis and feature modeling to reason about (1) a counter example that allows the arm to move outside of range without providing feedback and (2) the variability across the product line. We then applied testing to validate the counter examples discovered. While our Alloy models and feature models overlap, they are extracted using two different approaches and hence differ in granularity. This led us to synthesize several lessons learned and propose that researchers can use those to develop novel techniques for merging feature and Alloy models, for modularizing their architectures and for more easily discovering configurations for all necessary products. Future work includes expanding our properties, adding more rigorous testing and building physical test platforms.

## ACKNOWLEDGEMENT

We thank L. Cubrich for his help with domain knowledge and for providing us with an open source robotic surgery code base. This work was supported in part by an NSF EPSCoR FIRST award, a University of Nebraska Collaboration Initiative Seed Grant, and awards CCF-1755890, CCF-1618132 and CCF-1745775 from the National Science Foundation.

## REFERENCES

[1] 2018. Geomagic Touch Device. https://www.3dsystems.com/haptics-devices/touch.
[2] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. 2016. Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2016)*. 173–177.
[3] Lou P. Cubrich. 2016. Design of a Flexible Control Platform and Miniature in vivo Robots for Laparo-Endoscopic Single-Site Surgeries.
[4] Lou P. Cubrich. 2018. Surgical Robot Control Software. https://github.com/surgical-robots/robot-control-app/tree/tel-surge-update.
[5] Majdi Ghadhab, Sebastian Junges, Joost-Pieter Katoen, Matthias Kuntz, and Matthias Volk. 2017. Model-Based Safety Analysis for Vehicle Guidance Systems. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security, SAFECOMP*. 3–19.
[6] Daniel Jackson. 2006. *Software Abstractions - Logic, Language, and Analysis*. MIT Press.
[7] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. 2011. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development (AOSD '11)*. 57–68.
[8] Urtzi Markiegi. 2017. Test Optimisation for Highly-Configurable Cyber-Physical Systems. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B (SPLC '17)*. 139–144.
[9] Eric Markvicka. 2014. *Design and Development of a Miniature In Vivo Surgical Robot with Distributed Motor Control for Laparoendoscopic Single-Site Surgery*. Ph.D. Dissertation. University of Nebraska-Lincoln, Department of Mechanical and Materials Engineering.
[10] Microsoft. 2018. Coded UI. https://msdn.microsoft.com/en- us/library/dd286726.aspx.
[11] Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. 2011. A lightweight code analysis and its role in evaluation of a dependability case. In *Proceedings of the International Conference on Software Engineering, ICSE*. 31–40.
[12] Jonathan A. Saddler Niloofar Mansoor. 2018. Surgical Robot Models. https://sites.google.com/view/FSESurgeryRobots/.
[13] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Proceedings of the International Conference on Computer Aided Verification, CAV, Part II*. 23–41.
[14] Sam Procter, John Hatcliff, and Robby. 2014. Towards an AADL-Based Definition of App Architecture for Medical Application Platforms. In *International Symposium on Software Engineering in Health Care,SEHC*. 26–43.
[15] Danielle Stewart, Michael W. Whalen, Darren D. Cofer, and Mats Per Erik Heimdahl. 2017. Architectural Modeling and Analysis for Safety Engineering. In *Proceedings of the International Symposium on Model-Based Safety and Assessment*. 97–111.
[16] Jian Xiang, John C. Knight, and Kevin J. Sullivan. 2015. Real-World Types and Their Application. In *International Conference on Computer Safety, Reliability, and Security, SAFECOMP*. 471–484.