12-2019

Formal Modeling and Analysis of a Family of Surgical Robots

Niloofar Mansoor
University of Nebraska - Lincoln, niloofar@huskers.unl.edu

FORMAL MODELING AND ANALYSIS OF A FAMILY OF SURGICAL ROBOTS

by

Niloofar Mansoor

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfilment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Hamid Bagheri

Lincoln, Nebraska

December, 2019

FORMAL MODELING AND ANALYSIS OF A FAMILY OF SURGICAL ROBOTS

Niloofar Mansoor, M.S.

University of Nebraska, 2019

Adviser: Hamid Bagheri

Safety-critical applications often use dependability cases to validate that specified properties are invariant, or to demonstrate a counterexample 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 modeling and analysis. This thesis presents a novel approach for building an end to end dependability case for a software product line, where a property is formally modeled, a counterexample is found and then validated as a true positive via testing. There has not been such a study that we know of in an emerging safety-critical domain, specifically of robotic surgery. This thesis will detail a study on a family of surgical robots that combine hardware and software components and are highly configurable, representing over 1300 unique robots. At the same time, these robot systems are considered safety-critical and should have associated dependability cases. We conducted 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.

ACKNOWLEDGMENTS

I want to express my sincere gratitude to my advisor, Dr. Hamid Bagheri, for his visions and encouragement for my research. I appreciate his determination through this process and his belief in my abilities, and I am very grateful for his help and guidance.

I would also like to thank Dr. Myra Cohen for her advice and help while I was working on this research, and for always guiding me in the right direction. I also express my appreciation to my committee members, Dr. Witawas Srisa-an and Dr. ThanhVu H. Nguyen, for their helpful comments and suggestions.

I thank my colleagues, Bruno Silva and Jonathan Saddler, who helped me with this research. I gratefully acknowledge the assistance of Dr. Shane Farritor for giving us the opportunity to work on the robot control system and helping us with the resources we needed to study the system.

Special thanks goes out to my friends in Lincoln for their constant support and valuable advice. I will always be grateful for their kindness, and I feel lucky to have found such wonderful friends.

Last but not least, I would like to express my deepest gratitude to my mom and dad for always supporting me, especially through the difficult times. Their unwavering kindness and patience despite the physical distance between us mean the world to me. I will always be grateful for all the opportunities they have provided for me, and for the freedom they have given me to explore and take risks in my life.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cyber-Physical Systems (CPS) are a class of systems that integrate computation with physical systems. In these systems, computers sense and control processes in the physical world and physical processes affect computations. Recently, these systems have been widely used in safety-critical areas, such as automotive systems, traffic control, aircraft, military systems, and medical devices [26]. Due to their increasing relevance in safety-critical applications, it is of utmost importance to ensure that CPS operates without faults, as faults and undesired behavior in safety-critical systems can lead to catastrophic events. In Cyber-Physical Systems, hardware-software controls are tightly interweaved with hardware, which impacts the selected configuration of the software. Additionally, software computationally enforces the constraints of the hardware in these systems.

The open problem of safety assurance and dependability in safety-critical systems is not a new one, and there is ongoing research on finding improved and more reliable ways to ensure safety. Some well-known examples of failure in safety-critical systems include the Ariane V launch failure [25], the losses of Mars Polar Lander and the Mars Climate Orbiter [4] [6], and the unsafe administration of radiation from the Therac-25 medical linear accelerator that led to severe injury or death in several patients [41]. The most recent example of a high profile cyber-physical and safety-critical system failure is the case of Boeing B-737 MAX fatal accidents [37] that left 347 people dead in two different crashes. All these different cases

of malfunction and failure in safety-critical systems strengthen the argument that system dependability must be a more rigorous and comprehensive process so that these disastrous events can be prevented.

An instance of using CPS in a safety-critical area is robotic surgery, in which robots are controlled by surgeons from a console to perform delicate and complex procedures. These surgical robots can be configured in multiple ways and for different types of surgeries, and they use various 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 gynecologic surgeries and on different types of patients. The robotic surgery systems are highly configurable and can be viewed as a family of robots (i.e., a software product line), leading to hundreds if not thousands of possible configurations. The surgeon can choose their desired configuration for performing specific tasks and satisfying his or her personal preferences. This fact makes ensuring the dependability of these systems more challenging, as their configurability adds many layers of logic and complexity to the system.

Dependability in robotic surgery systems is crucial as some possible outcomes of a system failure are injury and loss of human life. Thus, these systems must be reliable and dependable. One way to establish confidence about dependability is to provide direct evidence that the system satisfies its claimed dependability goals [19]. Ensuring the safety and dependability of safety-critical systems is not an easy task, mainly since such systems usually consist of many software and hardware components. In the case of highly configurable surgery robots, the plug-and-play nature of the robot system makes ensuring dependability a much more challenging task, since all the different valid combinations of system features need to be considered when safety and dependability are verified.

In this thesis, the goal is to understand the challenges and feasibility of assuring the safety of a highly-configurable safety critical cyber-physical system. I present an approach for building a dependability case for such a system that considers both the variability and

safety of the system. This approach uses lightweight formal methods to model systems and performs automated analysis of the models, verifying safety-critical properties. On the other hand, it uses feature modeling to reason about the different configurations and products of a system. The last step of this approach is to test the concrete instances of systems in a guided, yet narrowly scoped manner to check and see whether any existing violation of the properties would manifest in reality. The case study for presenting this approach is a particular research prototype for a family of miniature surgical robots designed and built by the Center for Advanced Surgical Technology [2] at the University of Nebraska Medical Center and the University of Nebraska-Lincoln to perform minimally invasive laparoscopic surgery. The robots are controlled and configured by software designed and implemented for a variety of different robots. The software is open-source and available for all researchers and engineers to add new "plug-ins," which are modules that add new functionality to the robot control software. With its open source software, this family of surgical robot systems provides a valuable learning playground for us to explore. Two essential elements that distinguish this 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, the approach performs family-wide reasoning that would be difficult to achieve using static analysis or testing. For example, the 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, the approach supports 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 main contributions of this thesis are the following:

1. A novel approach for building dependability cases for families of systems, and the 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.

2. An end-to-end case study to validate an important physical property for a family of surgical robots is presented. The approach is implemented on this family to confirm the feasibility and address the challenges.

3. A set of lessons learned and discussion of future directions for assuring cyber-physical product lines.

The remainder of this thesis is organized as follows. Chapter 2 provides the background and related work. Chapter 3 presents an overview of the surgical robot family. Chapter 4 goes into the details of the approach, and Chapter 5 describes the process of constructing the dependability case for the surgical robot family. Chapter 6 provides the analysis and the results obtained after building the dependability case. Chapter 7 presents some discussions, challenges, and lessons learned along the way. Finally, Chapter 8 concludes this thesis with a summary of the contributions and the visions for future work.

Chapter 2

Background and Related Work

In this chapter, I provide background and related work on safety and dependability of safety-critical cyber-physical systems and various approaches to ensuring dependability.

There have been various works on safety and dependability assessment of critical systems. A study on cost-effective, dependable software is presented in [10], and different approaches are suggested for assessing the safety and dependability of software. Traditional approaches for ensuring system safety and dependability are **process-based**, and the software is considered dependable and safe as long as it abides by one or more sets of standards. The process-based approach usually results in low-defect code, but they do not provide arguments about how the system satisfies critical properties. The safety standards recommend processes that must be utilized to achieve different levels of safety [30]. On the other hand, **evidence-based** approaches seek explicit evidence of safety, and define failure models of the software and use various approaches to ensure that the hazardous states are not reached. The results of formal verification of the system specification, model checking, static code analysis, or testing can be considered as evidence for software safety. Kelly and Weaver [23] presented a technique called Goal Structuring Notation (GSN) to improve the structure, rigor, and clarity of safety arguments. In this method, the claims of the argument are established as goals and items of evidence are documented in solutions [32]. Graydon et al. [17] presented a methodology based on GSN to co-develop a system with its assurance case, to enable the assurance needed

to drive development choices in each stage of development to build dependable software. In this approach, the sub-properties that constitute the high-level system property are checked with methods like testing, or formal verification.

A safety case is built for a cruise control system in the automotive domain in [42]. The hazards and the requirements the system needs to fulfill to avoid those hazards are identified, and a framework for constructing a safety case for this domain is suggested. For different hazards and failures in the system, various methods of producing evidence are suggested. Another study [36] suggests a new way to construct safety cases in automotive systems, with the goal of presenting a concrete and organized method to build systems that realize a specific standard. In their approach, the evidence comes from tracing the safety requirements of the system into their respective development artifacts in which they are realized. Graydon et al. have suggested an approach to increase clarity when it comes to claims that the system adheres to the safety goals [16]. They suggest using conformance arguments to clearly show how each sub-claim is satisfied to achieve the overall safety goal, and if the evidence used is sufficient to claim safety. The arguments are also used to clarify how the developers have interpreted the safety requirements. In their case study, they use a formal analysis technique and testing results as evidence to support their sub-claims. Denney et al. describe their approach for creating a safety case for an autopilot software for an aircraft [13]. Their approach adopts goal-based argumentation for linking evidence (e.g., results of software verification) to claims that hazards are mitigated. Bourdil et al. present a methodology for structuring the formal models and the reasoning necessary to prove a claim on system design, relying on verification argument construction. In this method, which is applied to check the reliability of a critical function on an autonomous rover, the claims are linked with formal models and a rationale about how those model elements meet their requirements [7]. Sullivan et al. use apply bounded exhaustive testing to a complex software system to accomplish software assurance. They use the formal specification of the most critical parts of the system,

identify the test oracles, generate a set of program inputs and automatically create test cases for testing that input on the system [39]. Gacek et al. leverage model checking in building an assurance case language for architecture models in [14]. Their framework automatically generates assurance cases based on a system model described in an architectural design language, rules written in a domain specific language, and results of other forms of formal analyses. Another use of verification tools in checking safety critical systems is presented in [40], in which a model checker is used to verify the critical properties of a pacemaker.

Brunel and Chemouil, use Alloy to evaluate the safety of fire detection system [8]. Brunel et al. also use Alloy in [9] to establish the security of an Avionic Architecture. Near et al. construct a dependability case [33] for the control software of a proton therapy machine with modeling the property using the Problem Frames approach, performing lightweight code analysis to extract the relevant information of the code related to the property. Pernsteiner et al. [34] investigate the safety of a Radiotherapy Machine by developing specialized tools to verify the safety properties of individual components, and using Alloy to check whether the overall system-level property holds. A comprehensive study of different types of safety evidence, their structuring, and their assessment is presented in [32].

Our evidence-based approach is closely related to [34, 33], as we also use the property-part diagram [20] to informally build the dependability case, and then we formalize the high-level property it in Alloy. One of the differences in our approach comes from the fact that we are constructing the dependability case for a family of surgical robots. To achieve that goal, we are using code analysis to automatically extract detailed information to build individual Alloy models that conform to the meta-model, and we then check the dependability of the system by checking the property on each of the surgical robots that are defined in and can be controlled with the system. We have also used a combination of verification and analysis in our work, using feature models for studying the variability in the system and finding valid configurations to be tested.

Other recent approaches to assuring safety-critical systems include using model-based techniques [15], architecture-based safety analysis [38], and techniques based on real world types and type checking [44]. 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 recognize commonality and variability in the system. These approaches do not ensure the dependability of a highly configurable safety-critical cyber-physical system. Other research has examined testing cyber-physical product lines [28]; however, this work does not address the safety-critical aspects of the system. There has also been research on test generation for product lines using lightweight formal analyzers such as Alloy [24]; 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 approach can reason about medical apps in general [35] but it does not directly provide support for our use case, dependability cases for families of safety-critical devices, such as surgical robots.

Chapter 3

Overview of the Surgical Robots Family

This chapter presents an overview of the surgery robot system. I discuss the overall architecture of the system, followed by the details of the physical and software components of the system.

As opposed to open surgery, the traditional form of surgery in which the surgeon makes large incisions to access the body part under operation, minimally invasive laparoscopic procedures introduce many improvements in different aspects. These procedures have benefits such as reduced scarring, and less post-operative discomfort due to smaller incisions, quicker recovery times, and shorter hospital stays. Despite the benefits of these surgeries, there are also limitations such as reduced dexterity, and limited motion of medical instruments [11]. Using robotic platforms for performing such procedures expands the workspace, increases dexterity, and it gives the surgeon finer control during the procedure. 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 [29, 11].

These miniature surgical robots are small and do not need a dedicated customized surgical suite or infrastructure. They have reusable disposable tools that are familiar to surgeons, and they can be operated locally or remotely from a small console that includes haptic feedback and a screen that virtualizes robotic positioning. The robot system that has been developed

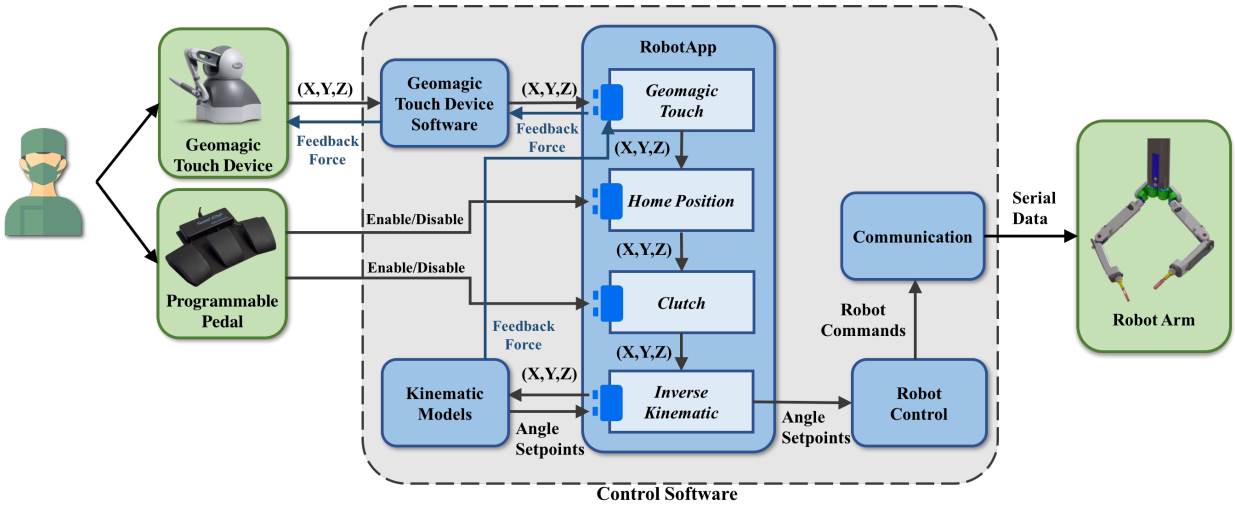
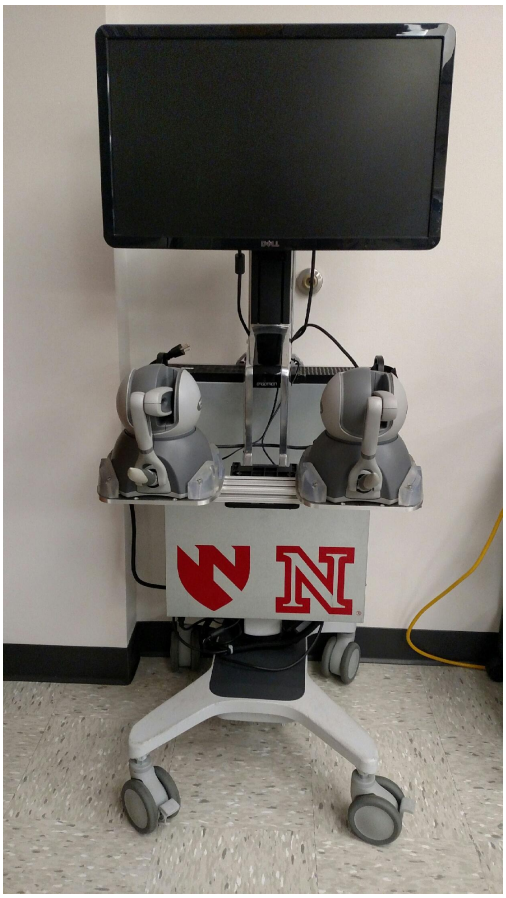


Figure 3.1: System components involved in arm movement.

includes multiple modules and plug-ins for different types of hardware control. The robot control software, which is written in C# and is available as open-source [12], can be manually configured, and the robots can be controlled via a graphical user interface. Two types of graphical user interfaces can be visible while using the robot system. One of these GUIs shows the values of the robotic equation calculations and outputs and is suitable for testing purposes. The other GUI shows the live feed of where the robotic end effector is and what parts are under operation. This system supports different robot arms, some of which have haptic feedback, and some that do not. Different solvers are developed to control the physical movements of corresponding arms, and other plug-ins are designed to perform tool position tracking, simulation, video, and voice communication. The source code that controls the physical aspects of the robot also supports a simulation environment. The simulation support is useful for developing correct solvers for robotic devices that aren't physically available but are virtually built and represented through simulation software.

The control software transforms the movements of the Touch device to set the position of the end effectors of the robot arms. The robot control software receives a set of coordinates from the software layer for the Touch device. Based on the chosen plug-ins and the configura-



(a) Remote surgeon user interface for the robot control platform

(b) Programmable pedals for the robotic surgery system

Figure 3.2: Physical control components of the robotic surgery system

tion, it is specified which inverse kinematic solver should be used for the current robot arm. The solver will then transform the coordinates into corresponding arm angles. The system has multiple architectural layers that correspond to different layers of the software system. The overall architecture of the robot system is presented in Figure 3.1. The physical components of the robotic system are shown in green boxes, and the software layer components are shown as blue boxes. The software has 56 different plug-ins as of November 2019 [12], but I have only included the very basic and essential plug-ins to work in a robot in the architecture figure. More on this topic is presented in section 3.2.

3.1 Physical Components

The surgical robot system's physical control setup is shown in Figure 3.2a 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. Each of them can control one of the 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].

Figure 3.3: A Two Armed Robot

The *programmable pedals* shown in Figure 3.2b are assigned to different functions that the surgeon can use during surgery. Two of the pedals should always be assigned to *clutch* and *home position* functions to enable/disable these functions in the control software. *Clutch* is to temporarily lock the arms in place while the GeomagicTouch device can get freely repositioned to a place more comfortable for the surgeon. *Home Position* clutch is used to reset the software/hardware device space so that the software is properly synced with information on where the effectors are. The third pedal can be assigned to another function, such as scale or cautery. Not that the cautery and scale function cannot be used in the same configuration, as a pedal should be assigned to only one, and the scale function is not necessary while using the cautery function.

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 [11]. 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 a specific task in surgery, such as shears for cutting or a cautery hook for cauterizing the wounds.

3.2 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. This component also receives the haptic feedback force value from the solver plug-in, and a force is applied to the surgeon's hand if necessary in the form of a vibration. The Geomagic Touch software is written in *C++*, and the robot control system developer has developed a software layer that enables the Geomagic Touch software integration with the rest of the C# code base.

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. These solvers contain the necessary calculations to transform coordinates from one space to another. The solvers receive the coordinates from the *RobotApp* component, which receives the coordinates from the aforementioned Geomagic Touch Software component. The corresponding solver to the arm calculates the joint angles for each joint of the robot arm, and sends 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. The graphical user interface is shown in Figure 3.4. The software GUI shows the available plug-ins on the left panel, and based on the desired configuration, the user can

load a number of plugins for each of the arms. The necessary plugins for the basic single robot arm movement are a *Geomagic Touch* plugin, a solver plugin, a clutch plugin, and a home position plugin. When the user clicks on any of the loaded plugins, its configuration panel appears on the right side of the GUI. There are additional options for the plugin, and the user can specify the flow of information from each plugin to another by specifying the values in the Signal Output Mappings view of the panel. For the robots built in CAST [2], the developers and mechanical engineers specified these configurations and saved them as configuration files. The hardware configuration of the corresponding robot is also stored in the configuration part, as controller configuration. Each robot arm has an immutable controller configuration, and the solver that calculates its angles is also an invariant. But the robot arm be configured to use a combination of different software plugins. In a typical scenario, the *Geomagic Touch* plug-in sends the coordinates to the *Home Position* plug-in, which in turn sends the updates plug-in to the *Clutch* plug-in. The *Clutch* plug-in sends its output as an input to the *Inverse Kinematic* plug-in, which connects to the solver to calculate the angle setpoints for the robotic arm.

Other components of the software also interact with a set of plugins. For instance, the *Geomagic Touch* software component interacts with the *Geomagic Touch* 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 to load for the way that the system is designed, 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 [11].

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.

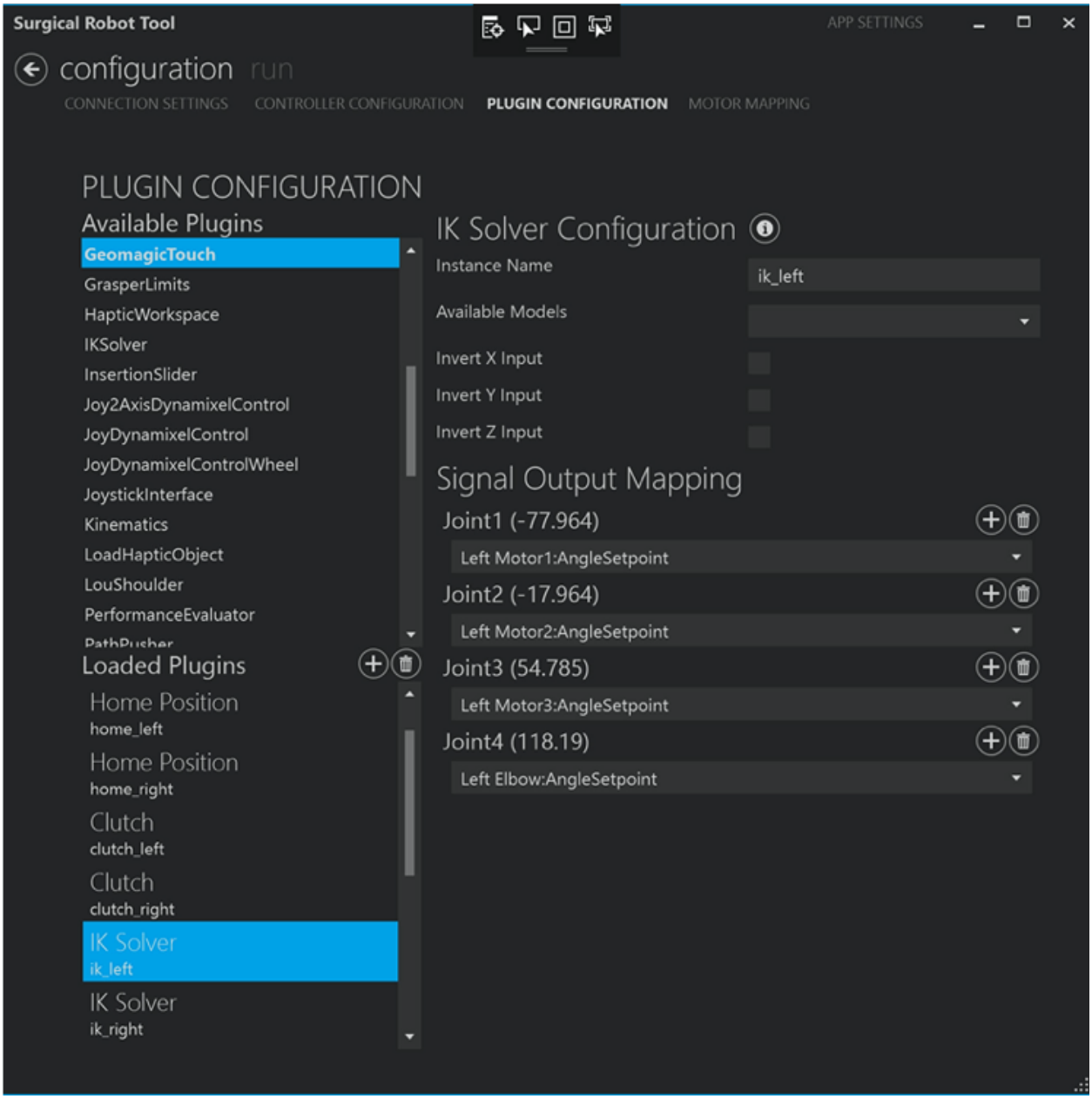


Figure 3.4: A Screenshot of the Robot Control System GUI

Chapter 4

Approach

This chapter presents a novel approach for building a dependability case for families of systems[1]. A dependability case is defined as an explicit, end-to-end argument that a system satisfies a critical property [33]. It is necessary to provide concrete evidence that the property is satisfied. One way to produce such concrete evidence is by utilizing software verification methods [34]. Dependability cases are usually constructed for single instances of systems, whereas our presented approach is designed not only for one single system but with a family of systems in mind. A family of systems can be described as a product line, borrowing the concept of Software Product Lines (SPLs) [43] to describe the variability and commonality within the family. SPLs are a set of systems that share common features, and combinations of those features that follow the system constraints create valid products. In SPLs, units that build the software are called features, and they can modify the functionality of the software system in specific ways. A family of systems can be described in a feature model, which is a model that specifies the features and the system constraints using a hierarchal tree diagram and cross-tree constraints. The concept of SPL can be extended to include different hardware and software features, and configurations of the system that combine different hardware and software components can be created. Constructing a dependability case for an entire

[1]The approach described in this chapter has been presented in my published paper "Modeling and testing a family of surgical robots: an experience report" [27].

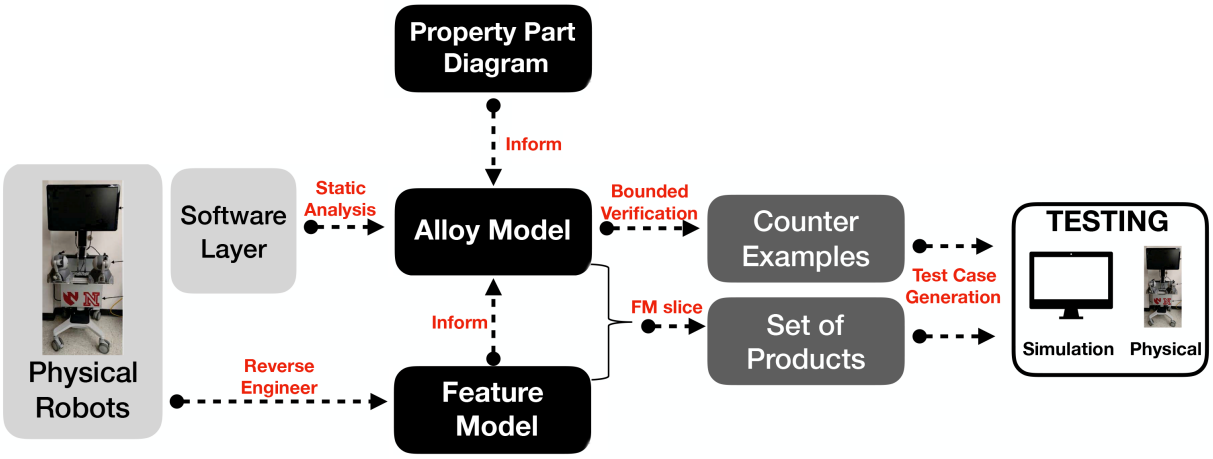


Figure 4.1: Overview of our approach

product line requires formal modeling and feature modeling of the system, and then mapping the models to one another to study the valid configurations and verify the correctness of the safety-critical property in them. The approach incorporates both an informal modeling approach, the problem frames approach [21], to specify the requirements for establishing the safety-critical property, and a lightweight formal language and analysis tool, known as Alloy [18] to determine the necessary components to satisfy the property and analyze the property. However, It is important to note that the approach is independent of any particular modeling language. The reason our team decided on using Alloy as the formal specification language, was its declarative and relational nature and its automated analyzer that facilitates checking the safety-critical properties. At last, if potential problems are found by means of verification, they can be validated via a method such as testing on the real system.

Figure 4.1 shows an overview of the process to construct a dependability case for a family of systems. The natural first step for building a dependability case is getting familiar with the system, its architecture, and the components that are involved in satisfying a safety-critical property. Carrying out this step requires comprehensive studying of the documentation, artifacts, and the codebase of the system. Interviews with domain experts also helps with building a knowledge base about a system when proper documentation is not available. The

choice of the method in which the architectural is described is up to the person who is constructing the dependability case, as they might see a particular language or modeling approach fit for expressing their system.

We then build the feature model using the domain expert knowledge and information from relevant parts of the code. We then define a safety-critical property for the system, a property whose correctness is essential for the system. An informal modeling approach such as the problem frames approach can be used to model the requirements and all the sub-properties that lead to the correctness of overall safety-critical property. A property-part diagram for the safety-critical property of system can help outline the necessities for constructing the formal model, by showing which parts of the system satisfy those sub-properties. We can then use the informal model as a basis for specifying the safety-critical property in a formal manner. We build an Alloy meta-model that describes the necessary system components to satisfy a property, their relationships, and the safety-critical property to be checked. The software layer helps us in creating individual Alloy models of each family in the system, which facilitates the process of reasoning about the system configurations from the feature models.

Since Alloy provides exhausting model searching within an specific and pre-defined bound, it can create instances of the system that can be mapped to the feature model configurations. We do not have to determine all the different features in Alloy, only the ones that can make distinctive difference in verifying the correctness. The Alloy Analyzer is able to give us counterexamples showing instances of property violation, and these instances can be mapped to slices of feature model. When Alloy shows a counterexample, there are some features set in those counterexamples. Our feature model included more features compared to the structural elements we needed to define in Alloy. When applying our approach to a case study, we created feature model slices with the features that were set in the individual Alloy models, and generated all the configurations that could be created with setting those features in the feature model. We also factored in the variability that other features which were not

described in the Alloy model could cause. These two models are superimposed together to identify sets of products that potentially violate the specified property.

The generated configurations in the feature model that are informed by the Alloy counterexamples can help in generating test cases. Formal verification tells us that this configurations are potentially faulty, but formal methods are known to overapproximate. Thus, the last step is to validate the counterexamples by creating the exact configurations from the feature model and testing those configurations. We did this step of the work manually and only applied the idea of testing on a few configurations. A systematic approach to test the potentially faulty configurations is left as future work. A further step would be to instantiate and validate these test cases on the physical system, which is also left as future work.

Chapter 5

Construction of a Dependability Case for a Family of Surgical Robots

In this chapter, I describe the details of constructing a dependability case for a family of surgical robots, using the approach outlined in Chapter 4. The system I am using to portray the details of building the dependability case is the University of Nebraska surgical robot system. The approach is applied to this system as a demonstration on a concrete system, but it can be implemented on any highly-configurable safety-critical cyber-physical system.

As mentioned before, the first step to construct a dependability case is getting familiar with the system, its architecture, and the components that are involved in satisfying a safety-critical property. I carried out this step of the work by studying the source of the system, and conducting a series of interviews with the engineers and developers who worked on the robot control system. The results of these investigations is presented as an overview of the architecture and the physical and software components of the system in Chapter 3.

The next step is to find a safety-critical property of the system to build the dependability case for that property. A safety-critical property is a property that the system needs to maintain at all times, and a dependability case needs to ensure that the property holds in all possible scenarios. I 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, and the arm is not positioned in an irregular manner. 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. The property being enforced is as follows:

> **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.*

Note that the manner in which the surgeon is notified (i.e. via haptic feedback or via visual messaging) is not specified, so I also consider property violation messages or logs shown on a screen as a warning measure designated to inform the surgeon of an undesirable situation.

This property is enforced by the robot controller system, consisting of hardware and software components, which monitors and drives the system's physical components. The dependability case spans the controller system as well as the physical modules involved in the arm movement, as it is considering all the relevant components of the system that have a role in satisfying this property.

The rest of the chapter is outlined as follows. Section 5.1 presents the informal approach to modeling, which provides a basis for the formal modeling. Section 5.2 provides the details of formalizing the model using Alloy as a modeling language and tool. Section 5.3 describes the feature model and the study on the configurability of the system. Section 5.4 presents a mapping approach from the formal model to the feature model, and finally, Section 5.5 describes the process of testing some configurations of the system.

5.1 Informal Modeling in Problem Frames

Modeling and verifying an entire complex software system such as the robot control software requires a lot of precision and can be quite expensive. Thus, I used the concept of trusted bases [22] to informally realize the components that are directly involved in satisfying this critical property. I leveraged the Problem Frames [21] approach to articulate the structure of the system and the underlying relationship thereof to the requirements [22]. This approach has a few key concepts that are briefly described below.

After identifying the safety-critical property, the parts of the system and their interactions are presented using the problem frames approach. This approach distinguishes the existing parts of the world, denoted as **application domain**, from the components that need to be built to solve the problem, denoted as **machines**. In problem frames, a property on a software is the specification that the software realized must fulfill.

Using these concepts, I constructed a problem diagram for the arm movement safety property, shown in Figure 5.1. A box is a representation of a system part that is involved in satisfying the property; the edges between the boxes represent a shared phenomenon that is used for the interaction between the parts.

The problem diagram illustrates the structure of the robot control system, which consists of: (1) physical components, such as the Geomagic Touch device, and the robot arm, (2) the high-level software components involved in satisfying the safety-critical property. The software components run on Windows systems that can be connected to robots. The software is written mostly in the C# programming language, and it distributed across hundreds of source files.

There is shared *phenomena* between the system parts, which is the way that they interact with one another. Examples of these phenomena are signals generated by the Geomagic Touch device sent as a set of coordinates to the software system, and data exchanged between

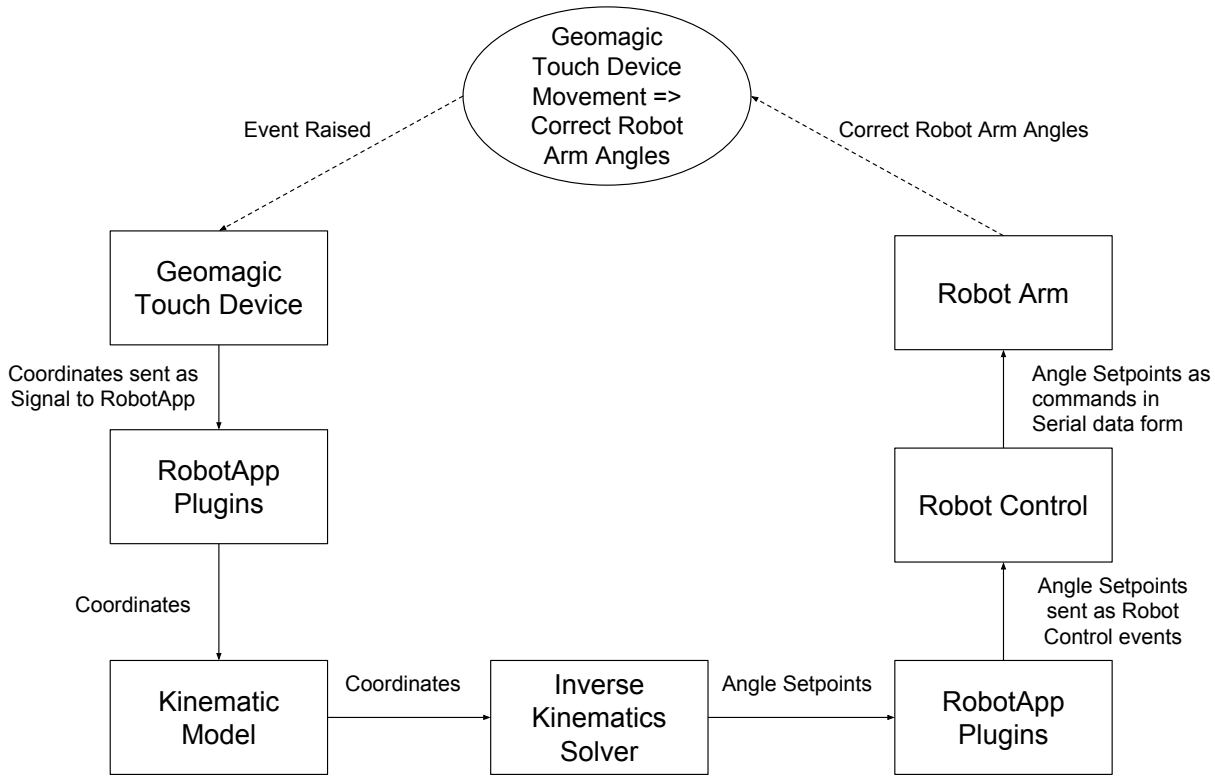


Figure 5.1: Problem diagram for the arm movement safety

different components of software. In the diagram, these phenomena are the labels on the edges between the parts, and the arrow on each edge shows the direction of information flow between the parts.

The problem diagram can show a typical scenario in the robot control system. The parts of the system communicate with one another as follows: The *Geomagic Touch Device* sends the coordinates of the end effector in its workspace to the software component RobotApp Plugins. The RobotApp Plugins send the coordinates to the *Kinematic Model*, which in turn sends the coordinates to the appropriate *Inverse Kinematic Solver* that is associated with the selected robotic arm. The *Inverse Kinematic Solver* calculates the angle setpoints and sends them back to the *RobotApp Plugins*. I abstract the diagram, and I don't go into the detail of all the plugins used since there could be several different plugins that manipulate

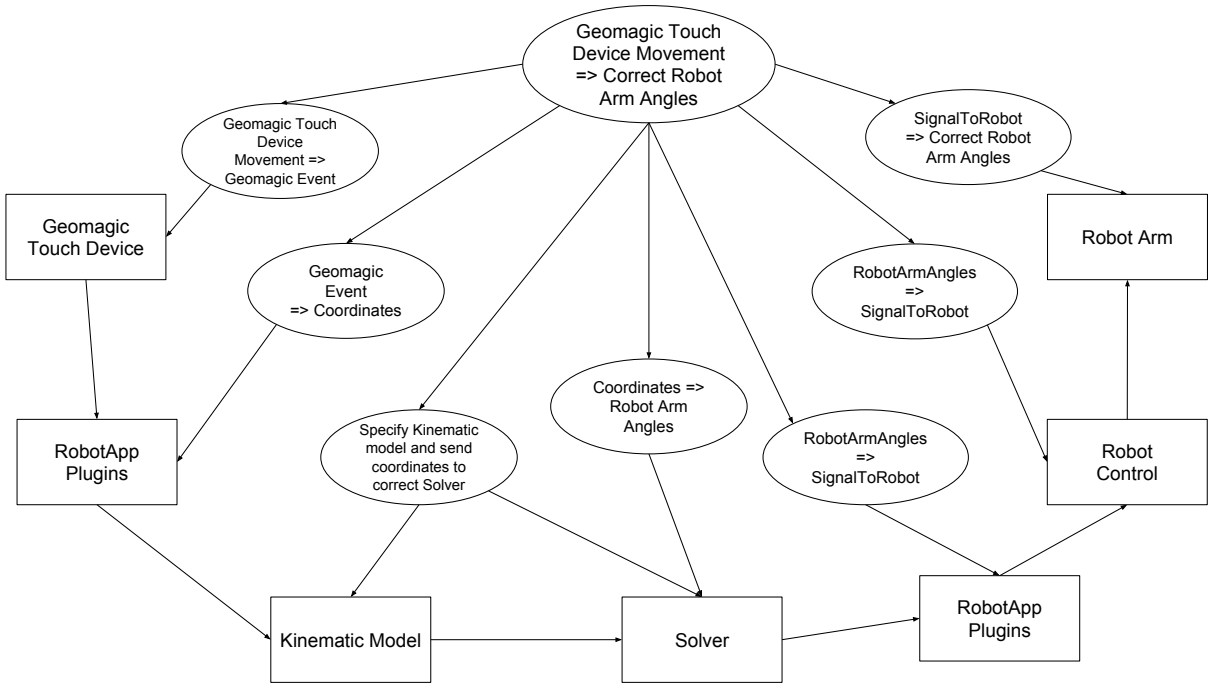


Figure 5.2: Property-part diagram for the arm movement safety

the input, whether before or after the kinematics calculations. After manipulating the input, the *RobotApp Plugins* part sends the angle setpoints as events to the *Robot Control* part, which finds the appropriate addresses of the joints and sends the serial data to the *Robot Arm*, and the robot arm moves to the position that the user intends.

I embellish the problem diagram and build a property-part diagram (Figure 5.2) that shows how the overall requirement for the safety-critical property is divided into sub-properties on the individual parts. All the properties are simply and informally stated as events that lead to other events. This diagram shows how each sub-property is satisfied in either one component or a combination of components. In other words, the diagram also shows the dependencies between the properties and the parts. We can see that each part of the system, while satisfying its own requirement and specification - such as sending coordinates from GeomagicTouch Device to the RobotApp Plugins - also takes part in satisfying the overall

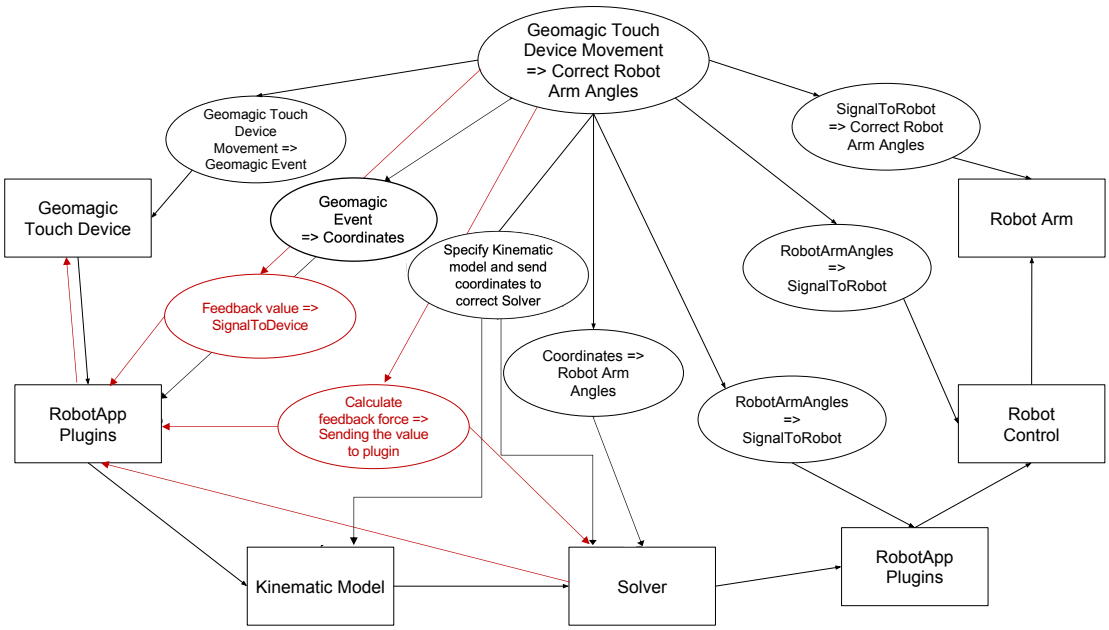


Figure 5.3: Completed property-part diagram for the arm movement safety

safety-critical property. The sequence of events in the system corresponds to the relationship between properties on the system.

After informally specifying the property and the parts involved in satisfying the property, I discussed the property and its specification with the domain experts and developers. I asked them to validate the event flow shown in the diagram. They informed me of a mechanism that is built to send a feedback force to the surgeon's hand via the Geomagic Touch Device when the position articulated by the surgeon is out of bounds for the robot arm. I modified the property-part diagram to include this mechanism (Figure 5.3). I added this mechanism to my safety-critical property, as I realized that failure to produce this feedback or any form of notification, might cause issues and crashes in the system. If the surgeon is trying to position the robot arm's end effectors in a position that is out of the robot's physical bounds and they are not informed of the physical limits, they might try to push the arm further in a direction that it can't reach, and eventually this might lead to the robot arm getting damaged or the

patient getting hurt.

The dependency structure in the property-part diagram correlates with the argument that the system establishes the arm movement safety property. If all the different parts of the system - whether physical or software components - satisfy their specifications, the overall property holds and the system is acting in a safe manner.

The next step is to formalize this specification and argument. Using lightweight formal methods, I model the relationship between the properties involved in satisfying the safety requirement, to establish that these properties can work together to ensure that the property holds. The property can be automatically checked for validity if formally modeled. This formal specification can be used as a reusable model to which all the extracted models for different robots must conform. Section 5.2 provides the details on formal specification of the system.

5.2 Formal Modeling in Alloy

This section describes a formal model for the surgical robots family in Alloy [18], a lightweight formal specification language based on a first-order relational logic, with an analysis engine that performs bounded verification of models. Three main reasons motivate the 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, the Alloy Analyzer, Alloy's backend analysis engine, provides an automated analysis for checking assertions and generating counterexamples.

Before diving into the details of the Alloy specification, I give a brief background on the Alloy language. Alloy is a first-order relational logic in nature. There are values assigned to variables, and the values of expressions evaluated in the context of a given instance, are relations [18]. The Alloy Analyzer is designed to provide fully automated analysis and find instances of a model by searching within the user-specified finite bounds. In a bounded search, failure to find an instance does not guarantee that an instance does not exist, but all the generated instances are valid. The Alloy Analyzer can also find counterexamples to assertions that are written to check some assumptions about the model. Again, since Alloy performs a bounded check, if a counterexample is not found, the correctness of the assumption is not guaranteed. But this problem can usually be ignored because of the small scope hypothesis. This hypothesis argues that a large portion of defects and bugs can be found by performing tests with all inputs within some small scope [5]. Thus, it is very likely for the analyzer to find the counterexamples even in small scopes. Sometimes the reason why the analyzer cannot find a counterexample is that the model is under or over-specified. Constraining the model in a way that allows the analyzer to find valid instances is essential when it comes to designing Alloy models.

The Alloy keyword sig represents a signature, which denotes a set of elements in the universe. Each signature may contain fields, which describes a relation that maps the elements of the signature to those in the field expression. Abstract signatures are those without any elements in them, except for the elements that belong to their extensions, which are specified by the keyword extends. One, lone, and some, and set are some multiplicities that can be used in declarations in Alloy. One indicates that the set contains only element, lone means zero or one element, some indicates a non-empty set, and set means a set that can be empty or not. There are also some quantifiers defined in Alloy that can be used to write logical statements, such as all (universal quantifier), some (existential quantifier), no (quantifier to specify that a set does not exist), lone (zero or one exists), and one (exactly one exists).

Facts are constraints defined in the model that the analyzer will always assume to be true. Predicates and functions are given names, and may or may not be given parameters. They are different in that the predicates always produce either a true or false defined by a formula in the body of the predicate, and functions produce a relation result of a specified type defined by the expression specified inside the function. Finally, assertions are specified using the keyword assert, and are checked using the run command. The show command can be used to ask the analyzer to show the valid instances of the model. More detailed information about the Alloy specification language can be found in [18].

To carry out the analysis, I 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. The property-part diagram (Figure 5.3) and the feature model (Section 5.3) help inform the Alloy model, as the Alloy model is a formal representation of the system parts and features and the analyzer provides an automatic analysis engine to verify the property and find potential flaws. Technically speaking, this Alloy module can be considered as a meta-model for the family of surgical robots.

Listing 5.1 outlines the meta-model module. The essential element types for each robot arm are defined as top-level Alloy signatures. I go through the signatures defined one by one and explain each signature and its corresponding system part. Line 3 defines the input and output of the system, the Coordinate signature defines the coordinates from the input device, and the ArmAngle signature defines the arm angles that are calculated and sent to the robot arm. Next, the hardware components are described. Signatures on lines 5-7 define the arm side, which can either be left or right. The HapticFeedback signature and its extensions define whether or not the feedback is available in a particular model (lines 9-11). The robot arm can be connected to an effector, and based on the interviews our research team did with the domain experts, we knew of five different effector types, which are defined as signatures (lines 13-18). The system also includes a set of pedals for enabling/disabling other functions,

```
module SurgeonRobot
//input and output
abstract sig ArmAngle, Coordinate {}
//hardware components
abstract sig Side{}
lone sig Left extends Side{}
lone sig Right extends Side{}

abstract sig HapticFeedback{}
one sig HapticsEnabled extends HapticFeedback{}
one sig HapticsDisabled extends HapticFeedback{}

abstract sig EffectorType{}
lone sig Cautery_Tissue_Grasper extends EffectorType{}
lone sig Cautery_Shears extends EffectorType{}
lone sig Cautery_Hook extends EffectorType{}
lone sig Tissue_Grasper extends EffectorType{}
lone sig Shears extends EffectorType{}

abstract sig PedalFunction{}
one sig ClutchButton extends PedalFunction{}
one sig HPButton extends PedalFunction{}
one sig ScaleButton extends PedalFunction{}
one sig CauteryButton extends PedalFunction{}

//each button is assigned to one function
sig PedalButton{
  assigned: one PedalFunction
}
abstract sig GeomagicTouch {
  input: one Coordinate,
  force: HapticFeedback,}
abstract one sig Robot {
  arms: some RobotArm}
abstract sig RobotArm{
  armside: one Side,
  armModel: one ArmType,
  effectorType: one EffectorType}
//software components
abstract sig Plugin {}
abstract one sig RobotApp {
  includes: some Plugin}
abstract one sig LoadedPlugins {
  loads: some Plugin}
abstract sig SolverFamily{
  calls: one KinematicModel}
//specifies the solver
abstract sig KinematicModel{
  solverResult: Coordinate -> ArmAngle}
abstract sig ArmType {
  anglelimit: set ArmAngle, //set of all the arm angles that are less than limit
  inverseKSolver: one KinematicModel}
abstract sig RobotControl{
  output: set ArmAngle,}
one sig Clutch_Plugin extends Plugin{}
one sig GeomagicTouch_plugin extends Plugin{}
one sig HomePosition extends Plugin{}
one sig GrasperLimits extends Plugin{}
one sig Scale extends Plugin{}
one sig DummyController extends Plugin{}
one sig ButtonInterface extends Plugin{
  setButtonForPedal : some PedalButton}
abstract sig SolverPlugin extends Plugin{
  solverfamily: one SolverFamily
}
```

Listing 5.1: Fundamental structures of the surgical robot family in Alloy

and there are 4 different pedal functions that can be assigned to them, which are defined as signatures on lines 20-24. I have also defined a PedalButton signature, with a relation that maps each PedalFunction to a PedalButton (lines 27-29). The GeomagicTouch control device is defined as another signature (lines 30-32), including the input and force fields which are related to the device. The signature Robot (lines 33-34) defines a robot, and has a field that relates the robot to its arm. The RobotArm signature (lines 35-38) defines the robot arm, and the fields within the signature define which side, arm type, and effector type are assigned to the robot arm.

The software components are then described in the model. I define the textsfPlugin (line 40) signature, the RobotApp signature (lines 41-42) that includes a number of plug-ins, LoadedPlugins (lines 43-44) that specify the loaded plug-ins in an instance, the SolverFamily (lines 45-46) that calls a specific KinematicModel which in turn calculates the ArmAngle by mapping the coordinates to the arm angle setpoints (shown as the relation on line 49). The ArmType signature (lines 50-52) defines the anglelimit field, which relates to a set of ArmAngle. The number of arm angles calculated for each arm type is defined and is extracted from the code base. Each ArmAngle is also related to one KinematicModel. The RobotControl signature defines the Robot Control component, which sends the output of the system. Lines 55-65 define some plug-ins for the system with some relationships related to them.

Listing 5.2 shows the constraints of the system specified using Alloy facts. Adding these constraints will help the analyzer remove the non-valid instances. The fact OutputConstraint specifies that for any RobotControl.output, ArmAngles should be produced by a solver in the system. SingleKinematicModelForArm and SolverAssignedToArm are facts that specify constraints about the kinematic solver and its relationship to the robot arm. Fact Coordinates-FromGMT specifies that coordinates should come from the GeomagicTouch Device, and the fact PluginsBelongToApp specifies that the plugins are connected to their parent component. The fact AngleCalculation specifies that the solver transforms each coordinate to a set of arm

angles. Listing 5.3 shows the rest of the constraints of the model, such as effector constraints, pedal button constraints, and some configurations that hold for all the different instances of one system.

To create individual family instances, I extract information about each specific robot arm and extend its corresponding element type in the meta-model. The next step to specify the safety-critical property in the meta-model, so that the analyzer can check the property for each individual model. Listing 5.4 shows an individual robot arm model for a robot arm that is named **FrankenVREP**. Note that the signatures are extended from the meta-model, and

```
66 //return the angles produced from a specific coordinate
67 fun getArmAngles[s: KinematicModel, c: Coordinate] : one (ArmAngle) {
68   s.solverResult[c]
69 }
70
71 //Facts:
72
73 //outputs should be in the range of solverResult
74 fact OutputConstraint {
75   all o: RobotControl.output | one a: getArmAngles[KinematicModel,Coordinate] | o
       = a
76 }
77
78 //There is one kinematic model for each robot arm
79 fact SingleKinematicModelForArm {
80     all r: RobotArm | one k: ArmType | r.armModel = k
81 }
82
83 //The kinematic model created for the instance corresponds to the arm type
84 fact SolverAssignedToArm {
85    KinematicModel in ArmType.*inverseKSolver
86 }
87
88 //all coordinates belong to GMT movements
89 fact  CoordinatesFromGMT {
90   all c: Coordinate | all g : GeomagicTouch | c in g.input
91 }
92
93 //all Plugins belong to RobotApp
94 fact PluginsBelongToApp {
95   all p: Plugin | one r: RobotApp | p in r.includes
96 }
97
98 //for each coordinate there exists an angle
99 //and that angle is in the solver result
100 fact AngleCalculation{
101   all c: Coordinate | some a: ArmAngle, s : KinematicModel| c->a in s.solverResult
102 }
```

Listing 5.2: Surgical robot family constraints in Alloy

```
106 //if the cautery effector is used, scale can't be used
107 //if the non-cautery tool is used, Grasper limits
108 //should be added to loads
109 //and Cautery button shouldn't be assigned
110 fact EffectorConstraints {
111   (RobotArm.effectorType = Cautery_Tissue_Grasper or
112   RobotArm.effectorType = Cautery_Shears or
113   RobotArm.effectorType = Cautery_Hook)
114   => ScaleButton not in PedalButton.assigned &&
115   CauteryButton in PedalButton.assigned &&
116   GrasperLimits not in LoadedPlugins.loads &&
117   Scale not in LoadedPlugins.loads
118   else
119   CauteryButton not in PedalButton.assigned &&
120   ScaleButton in PedalButton.assigned &&
121   GrasperLimits in LoadedPlugins.loads
122 }
123
124 fact ScalePedalNeedsScalePlugin {
125   ScaleButton in PedalButton.assigned
126   => Scale in LoadedPlugins.loads
127 }
128
129 fact PedalButtonConstraint {
130   all a, b : PedalButton | a != b implies some (a.assigned or b.assigned)
131 }
132
133 fact Config {
134   one Robot
135   one ArmType
136   one GeomagicTouch
137   one RobotControl
138   one RobotArm
139   one SolverPlugin
140   one SolverFamily
141   some Plugin
142 }
143
144 fact MoreConfig {
145   #ArmType.anglelimit > 2 //up to four
146   #RobotControl.output > 1
147   #PedalButton = 3
148   #ButtonInterface.setButtonForPedal = 3
149   #Coordinate = 1
150 }
```

Listing 5.3: Surgical robot family constraints in Alloy

the values are extracted from the code base. The property is then checked for each individual robot arm.

Next, I present the property that should be checked for any individual robot model. Listing 5.5 shows the property that the model is expected to satisfy. This property is formally specified as Alloy assertion ArmAngleCorrect. Predicate ProducedFeedback describes when the force should be produced and when the HapticFeedback should be enabled. The assertion

```
module frankenVREP

open metamodel/SurgeonBot

sig armangle extends ArmAngle{}
sig xyz_input extends Coordinate{}

//plugins expected from a typical config file
one sig GeomagicTouchPlugin_instance extends GeomagicTouch_plugin{}
one sig HomePosition_instance extends HomePosition{}
one sig Clutch_instance extends Clutch_Plugin{}
one sig IKSolver_plugin extends SolverPlugin{}
one sig ButtonInterface_instance extends ButtonInterface{}

one sig loaded_plugins_of_ extends LoadedPlugins {}{
  GeomagicTouchPlugin_instance +
  HomePosition_instance +
  Clutch_instance +
  IKSolver_plugin +
  ButtonInterface_instance
  in loads
}
one sig IKSolver_family extends SolverFamily{}{
  calls = FrankenBot
}
one sig FrankenBot extends KinematicModel{}
one sig FrankenVREP extends ArmType{}{
  inverseKSolver = FrankenBot
}
one sig FrankenVREPArm extends RobotArm{}{
  armModel = FrankenVREP
}
one sig UsedGeomagicTouch extends GeomagicTouch {}{
  force = HapticsDisabled
}
one sig Current_Robot extends Robot {}{
  arms = FrankenVREPArm
}
fact{
    #ArmType.anglelimit = 4
    #RobotControl.output = 4
  #solverResult = 4
}

check ArmAngleCorrect for 5 but 8 Plugin
```

Listing 5.4: Individual Alloy model for the FrankenVREP robot arm

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 counterexample, 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. The counterexamples will be discussed in detail in Chapter 6.

```
196 pred ProduceFeedback[output : RobotControl.output] {
197   output not in ArmType.anglelimit
198   some notification : GeomagicTouch.force | notification = HapticsEnabled
199 }
200
201 //assert if the arm angle created by movement is in the set of armangle limit
202 assert ArmAngleCorrect {
203 all a: RobotControl.output | a in ArmType.anglelimit
204 implies ProduceFeedback[a]
205 }
206 check ArmAngleCorrect for 4 but 5 Plugin
207 //What you should get as output:
208 //solverResult[Coordinate1] is equal to ArmAngle0 unless ArmAngle0 is not in
    anglelimit
```

Listing 5.5: Excerpts from an Alloy specification for the family of surgical robots.

5.3 Feature Modeling

In parallel to building the formal model of the system, our research team set out to study the variability and configurability of the system, as we intended to build a dependability case that ensures the dependability of the entire family, not just one single configuration of the robot system.

We use the concept of Software Product Lines (SPLs) [43], which are a set of systems that share common features, and combination of those features that follows the system constraints creates a valid product. In SPLs, we consider units that build the software as features. Features can modify the functionality of the software system in certain ways. A configuration is a set of features that construct a product, and each product in an SPL is a unique and valid combination of features that does not violate the feature model's constraints. SPLs can be concisely represented by feature models. A feature model is a model that specifies the features and the system constraints using a feature diagram, which is a hierarchical and-or tree, and the cross-tree constraints. In basic feature models, features can have various parent/children relationships such as *mandatory* and *optional*, which are self-explanatory, *or*, which means at least one child must be selected, and *alternative or*, which means that only one of the children must be selected. When a feature A *requires* feature B, the selection of A implies the selection of B, and when a feature A *excludes* feature B, both features A and B

cannot be present in the same product.

We consider the surgery robot system a product line, as the various software and hardware components work together in different configurations and generating these configurations requires complying with certain constraints. We gathered more information about the product line by conducting a series of interviews with the robot developers focusing on retrieving domain knowledge, as 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 [3].

From interviews, we learned the robot is a combination of two sets of configurable hardware components, namely arm types and effectors on the ends, and configurable software components. As mentioned in earlier chapters, the software components are collectively called *plug-ins*, 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 specified all the different arm types that we found in the code base, 4 of which were of active use in the research lab at the time. We also specified the five effectors that could be connected to the robot arms, and put them in the Cautery_tool and NonElectric_Tool categories. There were specific constraints about each set of effectors, and we categorized them to simplify writing the constraints for the model. An excerpt of the feature model in Figure 5.4 shows the expanded ArmType feature, which specifies 15 different robot arms.

The software components are divided into two parent features, LoadTimeConfigured and RuntimeConfiguredPlugins. The load time configurations are related to the robot arm motor configurations and the solver model that is associated with the robotic arm. Studying the code base helped us learn about the corresponding solvers to each robot arm, and aided us in writing the constraints for this part of the model. We also examined each inverse

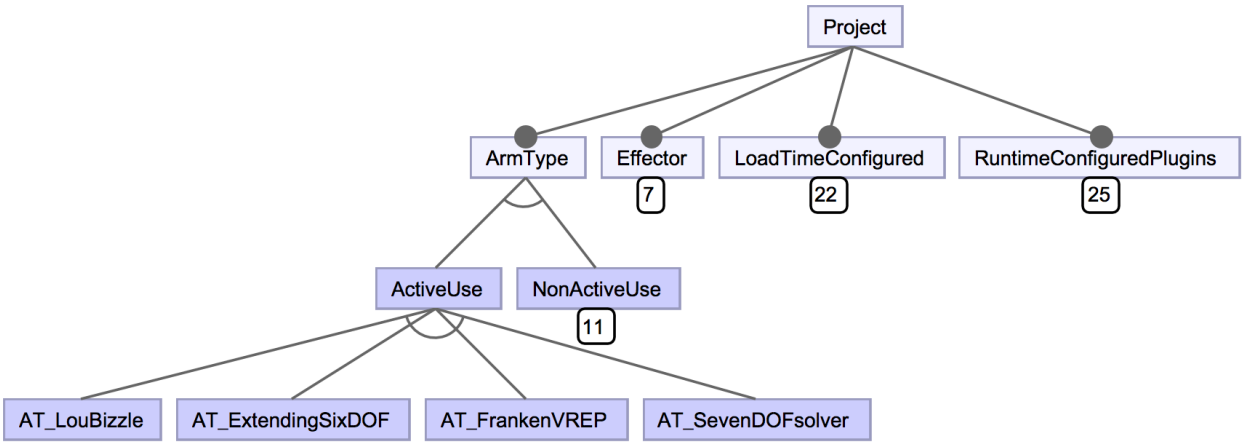


Figure 5.4: Feature Model: Arms

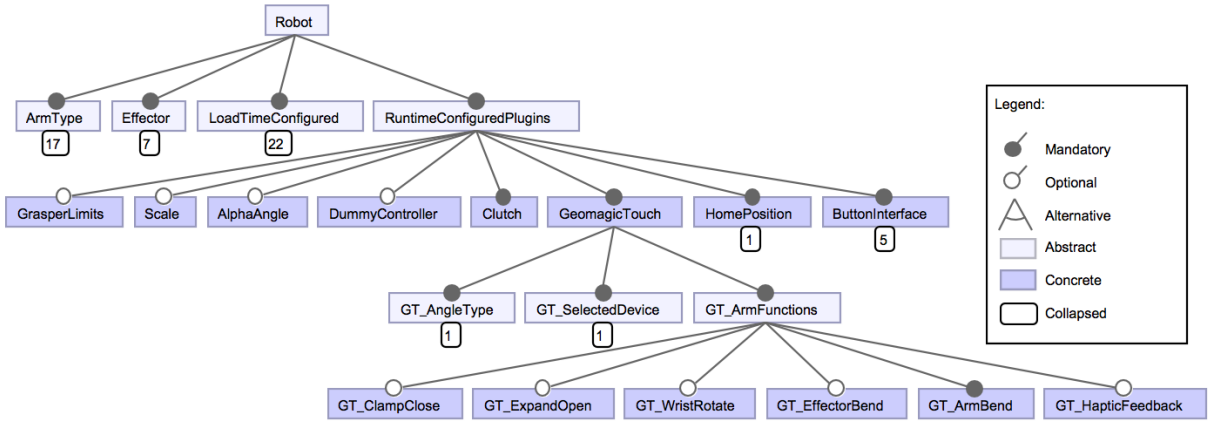


Figure 5.5: Feature Model: Runtime Configurations

kinematic solver code to find whether or not the ability of producing feedback force is programmed in them, which is an essential feature related to our safety-critical property. If a solver calculates the feedback forces, it implies that the configuration includes the GT_HapticFeedback feature. Figure 5.5 shows an excerpt of the feature model that includes the child features of RuntimeConfiguredPlugins. We also included some features of the plug-ins, namely AngleType, SelectedDevice, and ArmFunctions from the GeomagicTouch plugin, that sets some specific features for the geomagic touch device.

We also add the constraints of the robot control system into the feature model. Tables 5.1

and 5.2 present the list of constraints we imposed on the feature model. These constraints 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.

I have categorized the constraints in the tables for easier reading and understanding. The *Haptic feedback constraints* are the ones that describe the features related to the haptic feedback feature in the GeomagicTouch Device. Some of these constraints also specify whether or not a solver produces the feedback force. It is necessary to note that the features starting with *AT* are the arm types, and features starting with *SM* are solver models.

Solver constraints show the corresponding solvers for each arm. The do not work correctly if they are used with a solver other than the one specified for them. If any arm is paired with a wrong solver, the angle setpoint calculations will not be correct. *Effector and pedal constraints* represent the some of the hardware/software constraints of the system. For instance, if the scale plug-in is loaded, it implies that one of the buttons on the pedal is assigned to the scale function. Another example is the constraint that states the fact that the system cannot have the cautery and scale function at the same time. Some constraints about different types of effectors are also presented. *Arm side constraints* represent the constraints about input for different arm sides.

A full version of the feature model in XML format is presented in Appendix A of this thesis.

Constraint category	Constraint imposed on the model
Haptic feedback constraints	AT_LouBizzle ∨ AT_FiveDOFsolver ∨ AT_ExtendingSixDOF ∨ AT_FourDOFsolver ∨ AT_FiveDOFcheater ∨ AT_FiveDOFcheaterVREP ⇒ IK_OutputForces ∧ GT_HapticFeedback
	AT_SevenDOFsolver ∨ AT_FourDOF_needle ⇒ ¬IK_OutputForces ∧ ¬GT_HapticFeedback
	AT_SevenDOFsolver ∨ AT_FourDOF_needle ⇒ ¬IK_OutputForces ∧ ¬GT_HapticFeedback
	SM_KT_TwoArmCoupledShoulder ⇒ ¬NK_OutputForces ∧ ¬GT_HapticFeedback
	SM_KT_TwoArmCoupledShoulder3DOF ∨ SM_KT_CoupledShoulder3DOF ∨ SM_KT_CoupledShoulderAndElbow3DOF ∨ SM_KT_CombinedBot ⇒ NK_OutputForces ∧ GT_HapticFeedback
	AT_FrankenVREP ⇒ ¬GT_HapticFeedback
Solver constraints	AT_TwoArmLouBot ⇒ SM_KT_TwoArmCoupledShoulder3DOF
	AT_MarkBot ⇒ SM_KT_CombinedBot
	AT_TomBot ⇒ SM_KT_TwoArmCoupledShoulder
	AT_TomShortArm ⇒ SM_KT_CoupledShoulderAndElbow3DOF
	AT_LouBot ∨ AT_LouBotWithCamera ⇒ SM_KT_CoupledShoulder3DOF
	AT_SevenDOFsolver ∨ AT_ExtendingSixDOF ∨ AT_FourDOF_needle ∨ AT_FourDOFsolver ⇒ SM_IK_IKSolverNormal
	AT_LouBizzle ∨ AT_FiveD0Fsolver ∨ AT_FiveD0Fcheater ∨ AT_FiveD0FcheaterVREP ⇒ SM_IK_IKSolver5DOF
	AT_FrankenVREP ⇒ SM_IK_FrankenBot

Table 5.1: List of constraints imposed on the feature model

Constraint category	Constraint imposed on the model
Effector and pedal constraints	Cautery_Tool ⇒ BI_CauteryFunction ∧ ¬BI_ScaleFunction
	Scale ⇒ BI_ScaleFunction
	NonElectric_Tool ⇒ GT_ExpandOpen ∧ GT_ClampClose ∧ GrasperLimits
	Cautery_Tool ⇒ ¬GT_ExpandOpen ∧ ¬GT_ClampClose ∧ ¬GrasperLimits
	EF_Cautery_Shears ⇒ ¬GT_WristRotate
	EF_Cautery_Tissue_Grasper ∨ EF_Cautery_Hook ∨ NonElectric_Tool ⇒ GT_WristRotate
	EF_Cautery_Hook ⇒ ¬GT_EffectorBend
	EF_Cautery_Tissue_Grasper ∨ EF_Cautery_Shears ∨ NonElectric_Tool ⇒ GT_EffectorBend
	¬(BI_ScaleFunction ∧ BI_CauteryFunction)
Arm side constraints	MC_AS_Left ⇒ ¬HP_InvertXYZInput
	MC_AS_Right ⇒ HP_InvertXYZInput
	MC_AS_Left ∧ IKSolver ⇒ ¬IK_InvertXYZInput
	MC_AS_Right ∧ IKSolver ⇒ IK_InvertXYZInput

Table 5.2: List of constraints imposed on the feature model

5.4 Mapping Feature Model to Alloy Models

So far in the process, we have Alloy models and feature models that are extracted using two different approaches, and while they overlap, they differ in granularity.

We extracted information about each robotic arm using code analysis, and this resulted in fifteen Alloy models. In the Alloy models we only set the necessary values for some relations, so they include *Arm Type, Solver, Geomatic Touch, Haptic Feedback*, and two plugins created to manipulate inputs from Geomagic Touch, named *Clutch* and *HomePosition*. We set these features and create slices of the feature model for each arm, using featureIDE [3] to calculate the number of products that can be created for each robotic arm.

The process of mapping the features was not systematic or automated. However, we were able to map the models with one another manually. Chapter 6 presents our results after mapping these models.

5.5 Testing

Our approach for testing the surgical software relies on Microsoft CodedUI [31] 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. The testing process was done in a semi-manual manner, and creating an automated testing process is among our goals for future work.

We 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. Chapter 6 presents more on the testing results.

Chapter 6

Analysis

This chapter presents the analysis of the models, and the results of our study.

6.1 Finding Alloy Counter Examples

Our guidance for building Alloy models to search for counterexamples was primarily code analysis. To cover the space of products of this robotic system, we needed to develop different models for each different robotic arm. This resulted in fifteen Alloy models, one of which is for the arm named **FrankenVREP**, and the robot's specification is seen in Listing 5.4. We are going to use this robot arm as a demonstration in this chapter, as we study one of the generated counterexamples for this model and a configuration generated for this arm. As mentioned earlier, 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 individual Alloy 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 some of the features were not part of the code 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, and realized that the configurations of five out of fifteen arms can potentially violate the safety-critical property. 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.

Figure 6.1 shows a counterexample that the Alloy Analyzer finds for **FrankenVREP**, which indicates that the robot could show a faulty behavior and not inform the surgeon on its physical limits. This counterexample describes a scenario in which FrankenBot, the inverse kinematics solver associated with the FrankenVREP arm, has calculated four different angle setpoints, but one the angles is outside the limit, i.e., not in the set of anglelimit. This counterexample indicates that the robot cannot reach as far as the surgeon intends, so naturally, the system should inform the surgeon of this situation. But as we can see, the feedback forces are not calculated and the feedback force feature is disabled in this robot (SurgeonBot/HapticsDisabled), and the surgeon will not be informed of the robot arm reaching

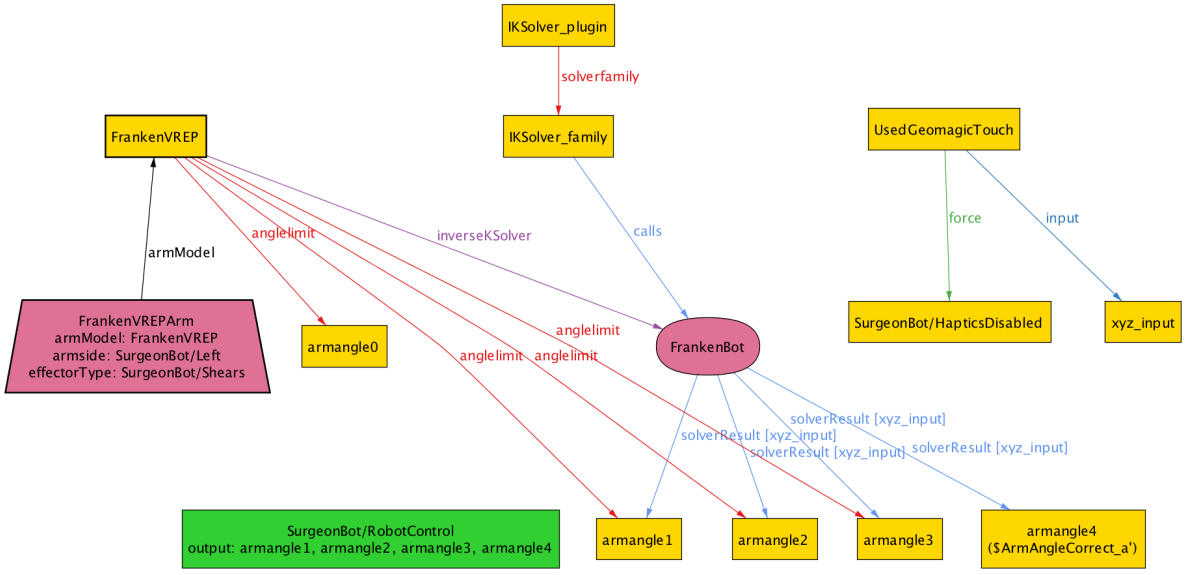


Figure 6.1: Visual representation of a part of a counterexample generated for FrankenVREP

its physical limits. We get various counterexamples for each of the 15 arms, and Figure 6.1 shows one. We next discuss the results of the feature modeling and its mapping back to these counter examples.

6.2 Feature Model

We went over our feature model design in Section 5.3. Our full feature model in an XML format can be found in Appendix A. There are 1,320 valid potential surgical robot configurations supported by this system.

As mentioned earlier, figures 5.4 and 5.5 show the high level features (*Arm Type*, *Effector*, *Load Time Configuration Options*, *Runtime Configuration Plugins*). In these figures 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). We also show 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 code analysis. We also showed the cross tree constraints in tables 5.1 and 5.2.

Most of these found features 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. An example configuration for the **FrankenVREP** that violates the safety-critical property is shown in Table 6.2. As we know, the Geomagic Touch device comes with a built in haptic feedback system. It can optionally be programmed by robot designer/developers who wish to implement a haptic response to collisions detected by the

Figure 6.2: An instance of the RobotApp that creates a violating condition

Arm Type	**FrankenVREP**
Effector	Tissue Grasper
Arm Side	Left
GeomagicTouch Device	OmniController
IKSolver Model	FrankenBot
IK Invert XYZ	False
GT Angle Events	Arm Bend, Effector Bend Wrist Rotation Signal
GT Function Events	**ExpandOpen, ClampClose**
Outputs XYZ Forces	No
HomePosition Offset	Grasper Offset
HomePosition Invert XYZ	False
Clutch Button Assigned	Yes
Home Position Btn. Assigned	Yes
Cautery Button Assigned	No
Scale Button Assigned	Yes
Grasper Limits Assigned	Yes
Angle Ranges	[-180,45],[-90,30], [-90,90],[0,140]
GT Angle Type	Pitch,Roll,Yaw
Control Mode	Relative Step

manipulated arms, by sending feedback of the forces encountered when colliding back to the Geomagic software for processing. Only some of the designs in the existing surgical robots are supportive of this feedback.

Upon close inspection of our models, we noticed FrankenVREP's physics calculator, FrankenBot, does not output directional forces. Because FrankenVREP's inverse kinematic solver does not compute directional forces, it cannot respond with feedback from the arm if we reach a critical zone in the surgeon's workspace. We have a constraint maintaining that if forces are not calculated or outputted by the solver, then the GeomagicTouch does not host its HapticFeedbackSignal feature, and signals don't get sent from this module. We note in our counterexample instance, that output forces is set to false. Correspondingly, we note

in our counterexample instance, that only the Expand and Clamp signals are sent by our Geomagic Touch, but not the haptic feedback signal. This is an omission we can easily detect using our feature model instance checker.

This is only one example of the configurations the feature model can give us. 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. I discuss this next.

6.3 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 code 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.

Chapter 7

Discussion and Lessons Learned

7.1 Discussion

In this thesis, we demonstrated a comprehensive approach to building a dependability case for families of systems and implemented our approach to a family of surgical robots. We found a design flaw in the system, generated the features of all the faulty configurations, and proceeded to test the system to realize if that potential problem can happen in a real scenario in concrete configurations of the system. In this chapter, I discuss some of the challenges and the lessons we learned in such a case study.

One of the challenges we faced in the initial stages of the study was the lack of comprehensive documentation. So creating the models of the system was a challenging task, and still, the models may not be a perfect representation of the system. We received a lot of help from the engineers who developed the system, to create a model that is as accurate as it can be.

One specific characteristic of the system that was helpful for us in building the models is the fact that the robot specifications in the system follow similar structures. This fact enabled us to build a meta-model for the entire system and generate the individual models in a semi-automated matter. If the developers decide to redesign or change the architecture or the method of defining solvers and robots, our meta-model needs to be adjusted to accommodate those changes.

Unfortunately, the process of creating a dependability case cannot be fully automated. We attempted to minimize the human effort by using code analysis to create the individual Alloy models and semi-automatically create the tests; but creating the meta-model and configuration files was a manual effort. Minimizing human effort can facilitate the adoption of the method in practice.

Regarding the creation of test cases, one of the challenges was to create configuration files for each robot arm. Configuration files for the software are crucial, since they contain the plug-ins and motor configurations necessary to control the robot are. The validity of these files is important because the robot will misbehave with an incorrect configuration of the software. Because our testing approach is to exercise the application's GUI to verify if a counterexample holds for a given robot, the arm's configuration file should be loaded by the system. We had some of the robot arm configurations available, since the engineers working with the system were currently using them. Still, to test the rest of the robotic arms, we created the configuration files loading the absolute necessary plugins to control a robotic arm.

Another testing challenge was to construct oracles for test cases for each robot. We had to inspect manually in the interface if a coordinate would place the arm in an invalid position. If positive, we would create an assertion failing, showing that the arm is in a wrong position and should produce force feedback. Otherwise, the test case would assert true.

One last testing challenge was to create completely automated tests. The testing process is still in a semi-automated fashion. It would be necessary to create a tool that can rip the interface to test other aspects of the system. This tool can ease the creation of test cases because they would be created by traversing a data structure (e.g., a graph) derived through the ripping process. Another direction is to use symbolic execution to test different input values to move the arm and verify if it will produce force feedback.

Next, I present the lessons we learned while conducting this study.

7.2 Lessons Learned

Conducting this study was a different experience as we had to consider not only the software aspects of a system, but the physical aspects of the system. Working with a software system designed and implemented by mechanical engineers was a valuable experience, as we witnessed the difference between how software engineers describe the systems they have developed, compared to how other engineers do. We mostly needed to understand the system from a higher-level perspective, to be able to model the software and the property, and we needed an overall idea of how the system operates. The mechanical engineers were eager to talk about the low-level mathematical and implementation details and it took a lot of communication for both teams to be on the same level about what kind of information was necessary and useful to conduct this research.

Another important lesson we learned is that the architecture of the system plays a large role and can help analysis. The way a system is designed and implemented has a significant impact in conducting a dependability 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.

Another lesson we learned is that the 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.

Last but not least, we realized that it's important to have methods to map feature models to Alloy models. 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.

Chapter 8

Conclusions and Future Work

In this thesis, I presented an approach to verifying the dependability of a family of systems or product lines. We combined the problem frames approach and lightweight formal to model a system and its safety-critical properties, and we use feature modeling to study and reason about the variability in the product line. We then combine the information gathered from both models to find configurations that don't satisfy the safety-critical property, and we validate whether or not the property is violated in a concrete configuration of the software system.

To demonstrate our approach, we constructed a dependability case for a cyber-physical safety-critical software product line, a robotic surgery system. In this case study, we used the Alloy specification language and feature modeling to reason about (1) counterexamples that allow 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 counterexamples discovered. While our Alloy models and feature models overlap, they are extracted using two different approaches and hence differ in granularity. This fact 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.

As future work, we aim to construct dependability cases for different properties, as

different properties introduce different challenges in terms of modeling and code analysis. Creating automatic and systematic mapping methods between models that represent the system in different granularity and focus is also an interesting line of work. Another line of future work is adding automated and rigorous testing methods to the approach, as we did our testing semi-manually. At last, building physical test platforms is our goal for building complete end-to-end dependability cases.

Bibliography

[1] Geomagic touch device. https://www.3dsystems.com/haptics-devices/touch, 2018.

[2] Center for advanced surgical technology (cast). https://www.unmc.edu/cast/, 2019.

[3] Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool demo: Testing configurable systems with featureide. In *Proceedings of the ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, GPCE 2016, pages 173–177, 2016.

[4] Arden Albee, Steven Battel, Richard Brace, Garry Burdick, John Casani, Jeffrey Lavell, Charles Leising, Duncan MacPherson, Peter Burr, and Duane Dipprey. Report on the loss of the mars polar lander and deep space 2 missions. 2000.

[5] Alexandr Andoni, Dumitru Daniliuc, and Sarfraz Khurshid. Evaluating the "small scope hypothesis. Technical report, 2003.

[6] Mishap Investigation Board. Mars climate orbiter mishap investigation board phase i report november 10, 1999, 1999.

[7] Pierre-Alain Bourdil, Silvano Dal Zilio, and Eric Jenn. Integrating model checking in an industrial verification process: a structuring approach. 2016.

[8] Julien Brunel and David Chemouil. Safety and security assessment of behavioral properties using alloy. In *International Conference on Computer Safety, Reliability, and Security*, pages 251–263. Springer, 2014.

[9] Julien Brunel, Laurent Rioux, Stéphane Paul, Anthony Faucogney, and Frédérique Vallée. Formal safety and security assessment of an avionic architecture with alloy. *arXiv preprint arXiv:1405.1113*, 2014.

[10] National Research Council. *Software for Dependable Systems: Sufficient Evidence?* The National Academies Press, Washington, DC, 2007.

[11] Lou P. Cubrich. Design of a flexible control platform and miniature in vivo robots for laparo-endoscopic single-site surgeries, December 2016.

[12] Lou P. Cubrich. Surgical Robot Control Software. https://github.com/surgical-robots/robot-control-app/tree/tel-surge-update, June 2018.

[13] E. Denney, G. Pai, and I. Habli. Perspectives on software safety case development for unmanned aircraft. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–8, June 2012.

[14] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: an assurance case language for architecture models. In *ACM SIGAda Ada Letters*, volume 34, pages 19–28. ACM, 2014.

[15] Majdi Ghadhab, Sebastian Junges, Joost-Pieter Katoen, Matthias Kuntz, and Matthias Volk. Model-based safety analysis for vehicle guidance systems. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security, SAFECOMP*, pages 3–19, 2017.

[16] P. Graydon, I. Habli, R. Hawkins, T. Kelly, and J. Knight. Arguing conformance. *IEEE Software*, 29(3):50–57, May 2012.

[17] Patrick J Graydon, John C Knight, and Elisabeth A Strunk. Assurance based development of critical systems. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 347–357. IEEE, 2007.

[18] Daniel Jackson. *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006.

[19] Daniel Jackson. A direct path to dependable software. *Communications of the ACM*, 52(4):78–88, 2009.

[20] Daniel Jackson and Eunsuk Kang. Property-part diagrams: A dependence notation for software systems. Institute of Electrical and Electronics Engineers, 2009.

[21] MA Jackson. Problem frames: Analysing and structuring software development problems pdf. 2001.

[22] Eunsuk Kang and Daniel Jackson. Dependability arguments with trusted bases. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 262–271. IEEE, 2010.

[23] Tim Kelly and Rob Weaver. The goal structuring notation–a safety argument notation. In *Proceedings of the dependable systems and networks 2004 workshop on assurance cases*, page 6. Citeseer, 2004.

[24] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 57–68, 2011.

[25] Gérard Le Lann. The ariane 5 flight 501 failure-a case study in system engineering for computing systems. 1996.

[26] Edward A. Lee. The past, present and future of cyber-physical systems: A focus on models. *Sensors*, 15(3):4837–4869, 2015.

[27] Niloofar Mansoor, Jonathan A Saddler, Bruno Silva, Hamid Bagheri, Myra B Cohen, and Shane Farritor. Modeling and testing a family of surgical robots: an experience report. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 785–790. ACM, 2018.

[28] Urtzi Markiegi. Test optimisation for highly-configurable cyber-physical systems. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume B*, SPLC '17, pages 139–144, 2017.

[29] Eric Markvicka. *Design and Development of a Miniature In Vivo Surgical Robot with Distributed Motor Control for Laparoendoscopic Single-Site Surgery.* PhD thesis, University of Nebraska-Lincoln, Department of Mechanical and Materials Engineering, August 2014.

[30] John A McDermid. Software safety: Where's the evidence? In *Proceedings of the Sixth Australian Workshop on Safety Critical Systems and Software - Volume 3*, SCS '01, pages 1–6, Darlinghurst, Australia, Australia, 2001. Australian Computer Society, Inc.

[31] Microsoft. Coded UI. https://msdn.microsoft.com/en- us/library/dd286726.aspx, June 2018.

[32] Sunil Nair, Jose Luis De La Vara, Mehrdad Sabetzadeh, and Lionel Briand. An extended systematic literature review on provision of evidence for safety certification. *Information and Software Technology*, 56(7):689–717, 2014.

[33] Joseph P. Near, Aleksandar Milicevic, Eunsuk Kang, and Daniel Jackson. A lightweight code analysis and its role in evaluation of a dependability case. In *Proceedings of the International Conference on Software Engineering, ICSE*, pages 31–40, 2011.

[34] Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 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*, pages 23–41, 2016.

[35] Sam Procter, John Hatcliff, and Robby. Towards an aadl-based definition of app architecture for medical application platforms. In *International Symposium on Software Engineering in Health Care,SEHC*, pages 26–43, 2014.

[36] Willem Ridderhof, Hans-Gerhard Groß, and Heiko Dörr. Establishing evidence for safety cases in automotive systems - a case study. In *SAFECOMP*, 2007.

[37] Tommaso Sgobba. B-737 max and the crash of the regulatory system. 2018.

[38] Danielle Stewart, Michael W. Whalen, Darren D. Cofer, and Mats Per Erik Heimdahl. Architectural modeling and analysis for safety engineering. In *Proceedings of the International Symposium on Model-Based Safety and Assessment*, pages 97–111, 2017.

[39] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. *ACM SIGSOFT Software Engineering Notes*, 29(4):133–142, 2004.

[40] L. A. Tuan, M. C. Zheng, and Q. T. Tho. Modeling and verification of safety critical systems: A case study on pacemaker. In *2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, pages 23–32, June 2010.

[41] Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 18(9I62/93):0700–001830300, 1993.

[42] Stefan Wagner, Bernhard Schatz, Stefan Puchner, and Peter Kock. A case study on safety cases in the automotive domain: Modules, patterns, and models. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 269–278. IEEE, 2010.

[43] David M Weiss and Chi Tau Robert Lai. *Software product-line engineering: a family-based software development process*, volume 12. Addison-Wesley Reading, 1999.

[44] Jian Xiang, John C. Knight, and Kevin J. Sullivan. Real-world types and their application. In *International Conference on Computer Safety, Reliability, and Security, SAFECOMP*, pages 471–484, 2015.

Appendix A

Full Feature Model

```
    <?xml version="1.0" encoding="UTF-8" standalone="no"?>
<extendedFeatureModel>
    <properties/>
    <struct>
        <and abstract="true" mandatory="true" name="Project">
            <alt abstract="true" mandatory="true" name="ArmType">
                <alt name="ActiveUse">
                    <feature name="AT_LouBizzle"/>
                    <feature name="AT_ExtendingSixDOF"/>
                    <feature name="AT_FrankenVREP"/>
                    <feature name="AT_SevenDOFsolver"/>
                </alt>
                <alt name="NonActiveUse">
                    <feature name="AT_MarkBot"/>
                    <feature name="AT_TwoArmLouBot"/>
                    <feature name="AT_TomBot"/>
                    <feature name="AT_TomShortArm"/>
```

```
        <feature name="AT_LouBot"/>
        <feature name="AT_LouBotWithCamera"/>
        <feature name="AT_FourDOFsolver"/>
        <feature name="AT_FourDOF_needle"/>
        <feature name="AT_FiveDOFcheater"/>
        <feature name="AT_FiveDOFcheaterVREP"/>
        <feature name="AT_FiveDOFsolver"/>
    </alt>
</alt>
<alt abstract="true" mandatory="true" name="Effector">
    <alt abstract="true" name="Cautery_Tool">
        <feature name="EF_Cautery_Tissue_Grasper"/>
        <feature name="EF_Cautery_Shears"/>
        <feature name="EF_Cautery_Hook"/>
    </alt>
    <alt abstract="true" name="NonElectric_Tool">
        <feature name="EF_Tissue_Grasper"/>
        <feature name="EF_Shears"/>
    </alt>
</alt>
<and abstract="true" mandatory="true" name="LoadTimeConfigured">
    <and abstract="true" mandatory="true" name="MotorConfiguration">
        <alt abstract="true" mandatory="true" name="MC_ArmSide">
            <feature name="MC_AS_Left"/>
            <feature name="MC_AS_Right"/>
```

```
    </alt>
</and>
<alt mandatory="true" name="SolverModelType">
    <and name="IKSolver">
        <alt abstract="true" mandatory="true" name="IKSolver_Type">
            <feature name="SM_IK_IKSolver5DOF"/>
            <feature name="SM_IK_IKSolverNormal"/>
        </alt>
        <feature name="IK_OutputForces"/>
        <feature name="IK_InvertXYZInput"/>
    </and>
    <and name="FrankenKinematic">
        <and abstract="true" mandatory="true" name="FKinematic_Type">
            <feature mandatory="true" name="SM_IK_FrankenBot"/>
        </and>
    </and>
    <and name="NormalKinematic">
        <feature name="NK_OutputForces"/>
        <alt abstract="true" mandatory="true" name="Kinematic_Type">
            <feature name="SM_KT_TwoArmCoupledShoulder3DOF"/>
            <feature name="SM_KT_TwoArmCoupledShoulder"/>
            <feature name="SM_KT_CoupledShoulder3DOF"/>
            <feature name="SM_KT_CoupledShoulderAndElbow3DOF"/>
            <feature name="SM_KT_CombinedBot"/>
        </alt>
```

```
        </and>
    </alt>
</and>
<and abstract="true" mandatory="true" name="RuntimeConfiguredPlugins">
    <feature name="GrasperLimits"/>
    <feature name="Scale"/>
    <feature name="AlphaAngle"/>
    <feature name="DummyController"/>
    <feature mandatory="true" name="Clutch"/>
    <and mandatory="true" name="GeomagicTouch">
        <and abstract="true" mandatory="true" name="GT_AngleType">
            <feature mandatory="true" name="GT_AT_PitchRollYaw"/>
        </and>
        <and abstract="true" mandatory="true" name="GT_SelectedDevice">
            <feature mandatory="true" name="GT_OmniController"/>
        </and>
        <and abstract="true" mandatory="true" name="GT_ArmFunctions">
            <feature name="GT_ClampClose"/>
            <feature name="GT_ExpandOpen"/>
            <feature name="GT_WristRotate"/>
            <feature name="GT_EffectorBend"/>
            <feature mandatory="true" name="GT_ArmBend"/>
            <feature name="GT_HapticFeedback"/>
        </and>
    </and>
```

```
            <and mandatory="true" name="HomePosition">
                <feature name="HP_InvertXYZInput"/>
            </and>
            <and mandatory="true" name="ButtonInterface">
                <and mandatory="true" name="BI_AssignedConsoleButton">
                    <feature mandatory="true" name="BI_ClutchFunction"/>
                    <feature mandatory="true" name="BI_HomePositionFunction"/>
                    <feature name="BI_ScaleFunction"/>
                    <feature name="BI_CauteryFunction"/>
                </and>
            </and>
        </and>
    </and>
</struct>
<constraints>
    <rule>
        <imp>
            <disj>
                <var>AT_LouBizzle</var>
                <disj>
                    <var>AT_FiveDOFsolver</var>
                    <disj>
                        <var>AT_ExtendingSixDOF</var>
                        <disj>
                            <var>AT_FourDOFsolver</var>
```

```
                    <disj>
                        <var>AT_FiveDOFcheater</var>
                        <var>AT_FiveDOFcheaterVREP</var>
                    </disj>
                </disj>
            </disj>
        </disj>
        <conj>
            <var>IK_OutputForces</var>
            <var>GT_HapticFeedback</var>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>AT_SevenDOFsolver</var>
            <var>AT_FourDOF_needle</var>
        </disj>
        <conj>
            <not>
                <var>IK_OutputForces</var>
            </not>
            <not>
```

```
                <var>GT_HapticFeedback</var>
            </not>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <var>SM_KT_TwoArmCoupledShoulder</var>
        <conj>
            <not>
                <var>NK_OutputForces</var>
            </not>
            <not>
                <var>GT_HapticFeedback</var>
            </not>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>SM_KT_TwoArmCoupledShoulder3DOF</var>
            <disj>
                <var>SM_KT_CoupledShoulder3DOF</var>
                <disj>
```

```
                    <var>SM_KT_CoupledShoulderAndElbow3DOF</var>
                    <var>SM_KT_CombinedBot</var>
                </disj>
            </disj>
        </disj>
        <conj>
            <var>NK_OutputForces</var>
            <var>GT_HapticFeedback</var>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <var>AT_TwoArmLouBot</var>
        <var>SM_KT_TwoArmCoupledShoulder3DOF</var>
    </imp>
</rule>
<rule>
    <imp>
        <var>AT_MarkBot</var>
        <var>SM_KT_CombinedBot</var>
    </imp>
</rule>
<rule>
    <imp>
```

```
        <var>AT_TomBot</var>
        <var>SM_KT_TwoArmCoupledShoulder</var>
    </imp>
</rule>
<rule>
    <imp>
        <var>AT_TomShortArm</var>
        <var>SM_KT_CoupledShoulderAndElbow3DOF</var>
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>AT_LouBot</var>
            <var>AT_LouBotWithCamera</var>
        </disj>
        <var>SM_KT_CoupledShoulder3DOF</var>
    </imp>
</rule>
<rule>
    <imp>
        <var>AT_FrankenVREP</var>
        <not>
            <var>GT_HapticFeedback</var>
        </not>
```

```
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>AT_SevenDOFsolver</var>
            <disj>
                <var>AT_ExtendingSixDOF</var>
                <disj>
                    <var>AT_FourDOF_needle</var>
                    <var>AT_FourDOFsolver</var>
                </disj>
            </disj>
        </disj>
        <var>SM_IK_IKSolverNormal</var>
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>AT_LouBizzle</var>
            <disj>
                <var>AT_FiveDOFsolver</var>
                <disj>
                    <var>AT_FiveDOFcheater</var>
```

```
                    <var>AT_FiveDOFcheaterVREP</var>
                </disj>
            </disj>
        </disj>
        <var>SM_IK_IKSolver5DOF</var>
    </imp>
</rule>
<rule>
    <imp>
        <var>Cautery_Tool</var>
        <conj>
            <var>BI_CauteryFunction</var>
            <not>
                <var>BI_ScaleFunction</var>
            </not>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <var>Scale</var>
        <var>BI_ScaleFunction</var>
    </imp>
</rule>
<rule>
```

```
    <imp>
        <var>AT_FrankenVREP</var>
        <var>SM_IK_FrankenBot</var>
    </imp>
</rule>
<rule>
    <imp>
        <var>NonElectric_Tool</var>
        <conj>
            <var>GT_ExpandOpen</var>
            <conj>
                <var>GT_ClampClose</var>
                <var>GrasperLimits</var>
            </conj>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <var>Cautery_Tool</var>
        <conj>
            <not>
                <var>GT_ExpandOpen</var>
            </not>
            <conj>
```

```
                <not>
                    <var>GT_ClampClose</var>
                </not>
                <not>
                    <var>GrasperLimits</var>
                </not>
            </conj>
        </conj>
    </imp>
</rule>
<rule>
    <imp>
        <var>EF_Cautery_Shears</var>
        <not>
            <var>GT_WristRotate</var>
        </not>
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>EF_Cautery_Tissue_Grasper</var>
            <disj>
                <var>EF_Cautery_Hook</var>
                <var>NonElectric_Tool</var>
```

```
            </disj>
        </disj>
        <var>GT_WristRotate</var>
    </imp>
</rule>
<rule>
    <imp>
        <var>EF_Cautery_Hook</var>
        <not>
            <var>GT_EffectorBend</var>
        </not>
    </imp>
</rule>
<rule>
    <imp>
        <disj>
            <var>EF_Cautery_Tissue_Grasper</var>
            <disj>
                <var>EF_Cautery_Shears</var>
                <var>NonElectric_Tool</var>
            </disj>
        </disj>
        <var>GT_EffectorBend</var>
    </imp>
</rule>
```

```
<rule>
    <imp>
        <var>MC_AS_Left</var>
        <not>
            <var>HP_InvertXYZInput</var>
        </not>
    </imp>
</rule>
<rule>
    <imp>
        <var>MC_AS_Right</var>
        <var>HP_InvertXYZInput</var>
    </imp>
</rule>
<rule>
    <imp>
        <conj>
            <var>MC_AS_Left</var>
            <var>IKSolver</var>
        </conj>
        <not>
            <var>IK_InvertXYZInput</var>
        </not>
    </imp>
</rule>
```

```
        <rule>
            <imp>
                <conj>
                    <var>MC_AS_Right</var>
                    <var>IKSolver</var>
                </conj>
                <var>IK_InvertXYZInput</var>
            </imp>
        </rule>
        <rule>
            <not>
                <conj>
                    <var>BI_ScaleFunction</var>
                    <var>BI_CauteryFunction</var>
                </conj>
            </not>
        </rule>
    </constraints>
    <calculations Auto="true" Constraints="true" Features="true"
    Redundant="true" Tautology="true"/>
    <comments/>
    <featureOrder userDefined="false"/>
</extendedFeatureModel>
```