A Framework to Support Automatic Certification for Self-Adaptive Systems

Ioannis Nearchou
Western Michigan University, INearch@aol.com

Follow this and additional works at: https://scholarworks.wmich.edu/masters_theses

Part of the Systems and Communications Commons, and the Systems Architecture Commons

Recommended Citation
https://scholarworks.wmich.edu/masters_theses/5171

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact wmu-scholarworks@wmich.edu.
A FRAMEWORK TO SUPPORT AUTOMATIC CERTIFICATION FOR SELF-ADAPTIVE SYSTEMS

by

Ioannis Nearchou

A thesis submitted to the Graduate College in partial fulfillment of the requirements for the degree of Master of Science
Computer Science
Western Michigan University
August 2020

Thesis Committee:

Wuwei Shen, Ph.D., Chair
Elise de Doncker, Ph.D.
Alvis Fong, Ph.D.
A FRAMEWORK TO SUPPORT AUTOMATIC CERTIFICATION FOR SELF-ADAPTIVE SYSTEMS

Ioannis Nearchou, M.S.

Western Michigan University, 2020

Presently, cyber-physical systems are increasingly being integrated into societies, from the economic sector to the nuclear energy sector. Cyber-physical systems are systems that combine physical, digital, human, and other components, which operate through physical means and software. When system errors occur, the consequences of malfunction could negatively impact human life. Academic studies have relied on the MAPE-K feedback loop model to develop various system components to satisfy the self-adaptive features, such that violation of the safety requirements can be minimized. Assurance of system requirement satisfaction is argued through an industrial standard form, called an assurance case, which is usually applied at design time. I propose a novel framework to approximate a human certifier’s analysis of a cyber-physical system’s assurance case. In this framework, the Dempster-Shafer theory is integrated into the MAPE-K model as a measure of an assurance case denoting trustworthiness of a cyber physical system with self-adaptive features. Two case studies are presented, inspired by the ENTRUST methodology, to evaluate the framework based on randomized evidence scores which support the arguments of each case study.
ACKNOWLEDGEMENTS

I would like to acknowledge Dr. Simos Gerasimou, with whom I discussed insights regarding the ENTRUST methodology and how it is implemented to provide support for facilitating self-adaptation of automated software systems. The work of he and his colleagues, R. Calinescu, et al., influenced my goal to provide a concept for evaluating the assurance given by arguments about self-adapting software systems under any application that does so. Their methodology’s concepts and example cyber-physical system scenarios provided a basis for models developed by my predecessor, Dr. Chung-Ling Lin, which were ultimately used for testing the application of assurance verification, with a version of system examples being modeled and included into this thesis.

I would also like to acknowledge my predecessor Dr. Lin, for his previous work in creating a framework that simulates dynamic assurance case changes at the end of the MAPE loop, under the direction and possession of my advisor Dr. Wuwei Shen for his legacy project of automated Assurance Case assessment. The source code packages provided in a base folder, his model files regarding combined systems involving MAPE automata, and the software systems on which they perform self-adaptation (including his interpretation of MAPE on the unmanned underwater vehicle example from the ENTRUST methodology) provided the foundation for developing a novel concept to integrate assurance analysis of self-adaptive software and are included in this thesis. Further, I want to thank Dr. Wuwei Shen for providing guidance, patience, the application for demonstrating dynamic Assurance Case changes which was used to show how my framework’s results can be applied at a system’s runtime, and advice as I formulated how to approach
implementing the concept as a method to further develop and hopefully to test integration feasibility into applications which implement the MAPE loop software self-adaptation. Last, I would like to acknowledge my parents for providing me the support, resources, and patience that allowed me to focus on my academic goals to perform well in my course work and to attempt to achieve higher learning.

Ioannis Nearchou
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ......................................................................................................................... ii

LIST OF FIGURES ........................................................................................................................................ vi

1. INTRODUCTION ........................................................................................................................................ 1

2. BACKGROUND ........................................................................................................................................... 7

   2.1 MAPE-K and Self-adapting Software Systems .................................................................................. 7

   2.2 Automated System Models and Instances ....................................................................................... 9

   2.3 Goal Structuring Notation and Assurance Cases ............................................................................. 10

   2.4 Dempster-Shafer Theory and Assurance Confidence Measurements ............................................. 12

3. IMPLEMENTATION ................................................................................................................................... 16

   3.1 Overview ............................................................................................................................................. 16

   3.2 Java AC Node and Scoring Implementation ...................................................................................... 18

   3.3 Weight Configuration Calculation .................................................................................................... 21

   3.4 Input Format and Parsing .................................................................................................................. 25

   3.5 Learning Approach ............................................................................................................................. 31

   3.6 Dynamic AC Evaluation .................................................................................................................... 34

4. SIMULATION AND RESULTS ............................................................................................................... 38

   4.1 Design-time and Runtime Setup ....................................................................................................... 38

   4.2 Case Study: ENTRUST UUV System ................................................................................................. 40
Table of Contents - Continued

4.3 Case Study: Modified Foreign Currency Exchange .......................................................... 46

5. DISCUSSION .......................................................................................................................... 51
  5.1 Conclusions .......................................................................................................................... 51
  5.2 Difficulties ............................................................................................................................. 52
  5.3 Future Work ......................................................................................................................... 53

APPENDIX ............................................................................................................................... 55

BIBLIOGRAPHY ....................................................................................................................... 60
LIST OF FIGURES

1. Diagram of example application of MAPE-K to an IoT system.................................8
2. GSN nodes and relationships of an Assurance Case...........................................11
3. Design flow of the dynamic AC analyzing framework........................................17
4. visitGoal algorithm of visitor implementation for D-S theory calculations.............20
5. Algorithm for visitSolution method.................................................................21
6. Algorithm for getAllRuleCombinations method..................................................23
7. Algorithm for getAllWeightCombinations method...............................................25
8. Algorithm for parseSysACPathInput....................................................................26
9. Algorithm for parseACs method...........................................................................29
10. Algorithm for getACConfigs..............................................................................30
11. Algorithm for findBestWeights..........................................................................32
12. Flow diagram of the dynamic AC loop phase......................................................35
13. An input UUV-MAPE AC arguing over nominal system requirement fulfillment.....41
14. Final tuple trust scores for UUV-MAPE configurations satisfying requirements R1-R3.43
15. Final trust computations for learning phase UUV-MAPE configurations satisfying requirement R4.44
16. Resulting AC of UUV R4 Configuration in dynamic AC loop..............................45
17. Initial AC argument structure for related to requirements R1-R3 of modified FX case study........47
18. AC argument structure related to requirement R4 of modified FX case study........47
19. Final AC trust scores for modified FX example configurations satisfying requirements R1-R3....48
20. Final AC trust scores for modified FX example configurations satisfying requirement R4........49
21. Resulting AC of FX R1-R3 Configuration in dynamic AC loop............................50
22. Designed abstracted version of a GSN metamodel..............................................55
List of Figures - Continued

23. uuv_Training_Configurations_10Entries.csv learning phase input data. ........................................... 55
24. uuv2_Training_Configs.csv input learning phase input data. .......................................................... 56
25. Input certification tuple file for dynamic UUV-MAPE AC. ............................................................. 56
26. Output dynamic UUV-MAPE AC certification with best weight configuration................................. 57
27. fx2_Training_Configs_10Entries.csv learning phase input data. ....................................................... 57
28. fx_Training_Configs.csv learning phase input data. ........................................................................ 58
29. Input certification tuple file for dynamic modified FX-MAPE AC..................................................... 58
30. Output dynamic modified FX-MAPE AC certification with best weight configuration..................... 59
31. Class diagram of automatic certification support framework. ......................................................... 59
CHAPTER 1

INTRODUCTION

Presently, from the economic sector to nuclear energy sector, cyber-physical systems (CPSes) are increasingly being integrated into most aspects of human life. A cyber-physical system is defined by a combination of physical components with digital, human, and/or other components, which act through hardware behaviors that are managed by software [1]. While system sensors are used to monitor cyber physical system attributes and processes, which are analyzed to verify whether they fulfill the system’s requirements [2], software deployed on them plays an important role. Sector by sector, more software is becoming self-adaptive since it should be reconfigured when external environmental changes require the software to ensure safety requirements are being satisfied. Therefore, how to designing and implement CPSes has become a crucial yet challenging issue in the CPS community.

The failure of a system can serious consequences. According to a Reuters report in May 2018, regarding a 5.3 million automobiles were recalled by Fiat Chrysler due to a cruise control defect [6]. On March 2nd of 2018, a complaint was filed to NHTSA regarding a car rented from Avis. The driver reported they travelled from Olathe, Kansas, moving 70 miles per hour. When cruise control was engaged, the wipers were described to have activated on their own volition, while the dashboard instrumentation dials reverted to zero, and the cruise control feature was unable to disengage [6]. The driver stated that after being able to break the automobile and maneuver to the side of the road, the engine still engaged to allow the vehicle to travel at 70 miles per hour and was resisting the breaks [6]. Engineers at Fiat Chrysler determined the software issue
causing the problem, recalling over 4.8 million vehicles from the United States and 490,000 vehicles from Canada of model years 2014-2019, according to regulators [6]. If an effective self-adaptive CPS had been deployed, the risk to lives due to such software behaviors would have been mitigated. This example motivates people to increase the reliability of a self-adaptive CPS from two perspectives. One is to invent some new reliable methodology to develop a self-adaptive CPS to avoid many potential errors. The other perspective is to provide certification for a CPS so its assurance can be validated.

The importance and ubiquity of CPSes influenced researches to develop a novel technique called Monitor-Analyzer-Planner-Executor, which execute according to a knowledge base (MAPE-K) to provide runtime self-adaptation of a software system in CPSes [2, 3, 4]. MAPE-K consists of the four following steps. The Monitor step collects environmental data from a system’s sensors and sends it to the second step, i.e. the Analyzer step [4]. The Analyzer step takes the received data, stored system configuration data from a repository, and then determines if the system violates requirements placed on how it should function, based on stochastic system models created at design time and analysis performed by a problem verification engine at runtime [4]. If so, it determines if a new system configuration satisfies the violated system requirements, or that a failsafe configuration is required if none can be found. The Planner step determines step-by-step actions to cause the system to enter this new configuration [4]. A complete list of final actions to perform are then passed to the final step of the feedback loop, the Executor step, which executes the system changes required in the step-by-step actions list, to change the system’s configuration [4]. Each MAPE step is generally designed via automatons that model the behavior for each step and the relationships between steps [4, 5].
To provide a measure of system assurance regarding any MAPE-K implementing system that adapts an automated software system from the certification perspective, the researchers argue the importance of using an industrial argumentation structure known as an assurance case, abbreviated as AC, to provide a level of assurance regarding the state of a functioning adaptive software system [4]. An AC is defined in [7] as a set of claims that are verifiable and are supported by evidence, which support the argument that a software system satisfies its requirements. Generally, any software or physical system, ACs are generated at design time, posing arguments about how the system functions safely, or assertions about how the system should fulfill requirements related to the environment in which it is deployed. The ACs usually include results from the verification and testing activities as evidence to support their claims. Quality of Service (QoS) engineers, also called certifiers, use these ACs to evaluate the assurance asserted by them about their systems. If some part of the argument does not match a certifier’s prior experience, the certifier can disapprove a software system via assigning a failing score to its AC.

But, the authors of [8] noted an AC developed at design-time encompasses limited operational conditions that a CPS may encounter and then proposed a dynamic AC that operates on system knowledge gained during deployment. Generation of ACs at runtime can support the dynamic certification especially when a CPS increasingly exhibits some emergent behaviors by integrating the learning and adapting capability. A more important reason given by [8] is that dynamic ACs provides a mechanism of continually assessing and evolving the assurance reasoning, which is called through-life safety assurance, for a CPS. Given that a self-adaptive CPS can change physical configurations to uphold its system requirements, [4] considers partial instantiation of an AC for a system at design time, omitting complete system descriptions due to environmental unknowns to be encountered when an instance of the system is deployed in the
field. When the system is deployed, the remaining portion of the AC is completed, according to the environmental factors that become known. Due to these factors being variables, the argument substructures that argue over them must change as they do. Thus, the notion of a dynamic AC is considered. A dynamic AC is important because evaluation of a CPS’ assurance can be performed at runtime, so a tool is necessary to perform the AC evaluations and edits.

Manual certification of an AC is not only erroneous, but also time consuming. Thus, various mathematical models have been proposed to approximate domain specific knowledge in a certifier’s mind. In the context of an automated system that is being adapted by another software system, it is impossible to acquire a human certifier’s perspective about an AC that would change as the system it describes changes. Past studies such as [9] propose use of some aspect of Bayesian belief networks (BBNs) to predict the safety assurance of an AC that argues over the safety of a software system. According to [9], a BBN is a directed acyclic graph combined with conditional probability weights on edges, under assumptions of conditional independence, to perform probability computations. Due to the acyclic graphical structure of an AC, an AC can be translated into a related BBN graph that considers dependencies between subclaims and has probability weights assigned to the graph edges. Such predictions are based on the strength of causal structure connections within an AC. But the evidence provided for a given subclaim has the possibility of not being believable to a satisfactory degree. Instead, the competing approach Dempster-Shafer (D-S) theory builds upon Bayesian probability, but includes consideration of the degrees of belief about evidence. Such an approach is studied in application of ACs, in [10, 11]. From the point of a piece of evidence, an upper bound can be given about the confidence a QoS engineer holds about the evidence. This confidence then relates to the amount of knowledge possessed about the correctness of claim it supports, as well as a bound on the ignorance about the claim. When
evaluating parent claims in an argument, the uncertainty of these bounds in subclaims, as well as the ignorance and knowledge of the subclaims contribute to the ignorance and knowledge regarding the parent. Each relation from child to parent is weighted, such that one subclaim may contribute more to the knowledge and ignorance of its parent, than its siblings. The evaluations are performed until the bounds of knowledge and ignorance about the argument itself are calculated, giving a final confidence value for the AC. This approach also requires a tool that provides effective computations to evaluate ACs at runtime.

The goal of this thesis is to propose a novel approach to automating a human certifier’s assessment of self-adapted software systems, which provides an approximation to a human certifier onboard on automated physical systems. The tool that demonstrates this approach is broken into two stages, both demonstrated at runtime on a small data set. The first stage is considered a learning stage which utilizes AC argument templates that refer to the level of system requirement satisfaction for some CPS that undergoes self-adaptation, and associated physical configuration names for each AC, which are accompanied by initial evidence confidence values for D-S theory calculations. For each AC template, all possible subclaim weight configurations are produced. As each weight configuration is applied, the tool performs D-S theory calculations to find the best weight configuration for an AC template. This portion provides the knowledge base for D-S theory calculations to be applied during a MAPE feedback loop. In the second stage, the end stage of MAPE-K is simulated, where input CPS model, CPS AC, and model instance information is input and evaluated to see if a physical configuration change for the CPS instance is required in each iteration. When a configuration change is required, the input AC is rewritten to account for CPS configuration change. The results from the first stage are applied to user input for initial evidence confidence values of the new configuration, such that D-S theory is used to find
the confidence of the new AC, according to the best weight configuration for that argument’s structure.

The rest of the sections of this paper are ordered as follows. In chapter 2, the foundational information about the techniques implemented to perform design-time evaluations of example self-adaptive system software configurations, which inform how to calculate confidence of a dynamic assurance case about the adaptive system are discussed. In chapter 3, the implementations of the learning technique and dynamic AC change loop are described. Chapter 4, the results of the learning technique’ application to case studies. Finally, in chapter 5, implications regarding the results and difficulties with technique application are discussed. If viable, application of scoring adaptations at runtime may be useful for gauging the quality of system adaptations when a human certifier cannot be present and lead to more reliable adaptations to be made on automated software systems.
CHAPTER 2

BACKGROUND

2.1 MAPE-K and Self-adapting Software Systems

MAPE-K is a novel approach that facilitates self-adaptation of a CPS, to maintain function according to system specifications as external conditions evolve. Due to the academic and industrial interest in MAPE, the authors of [5] formulated formal general templates of the MAPE-K automata, to reduce the time required to design domain-specific states and transitions. Timed and event triggered MAPE automata can be designed by specifying triggers for state changes and what tasks are performed by the automata, according to the CPS application. Further the authors of [4] give templates for implementing CPS specific MAPE automata related to their methodology, providing a means of creating domains that are verified and supported by their ENTRUST tool. Further usage of verification methods chosen by a CPS project can apply a knowledge base, by which any such tool that supports self-adaptation can be informed to change a CPS’ configuration according to the design-time generated requirements they must satisfy. Such CPS-MAPE projects can range from self-driving cars to some system that is connected into the Internet of Things (IoT), as displayed in Figure 1.
For the example provided, the MAPE automata and knowledge base are interwoven with management software and the IoT system, where the Monitor automaton reads sensor information related to information sent by trust agents, then sends notifications of changes in the system to the Analyzer. Analysis results are sent to the Planner, where procedures for system change are planned, then sent to the Executor. The Executor interacts with system actuators to perform the planned system configuration changes. Additionally, as considered by [4], an AC argument about the correctness of a system’s performance can be added to the knowledge base of the system. Engagement of the feedback loop can adjust the behavior of a self-adaptive system to closely follow its expected arguments based on this knowledge base, however, such arguments will have to accordingly change, and thus require another evaluation. The means and reasons for doing so shall be discussed later in this section and in the following section.
2.2 Automated System Models and Instances

Unified Modeling Language (UML) is a standard language used in the design of a general system’s domain model through metaclass concepts such as Classes, Attributes, Relationships, Generalizations, and other structural components that can define what data should be held and how the system components interact. To understand how to score the decision a MAPE-K implementing application makes to change the physical configuration of self-adaptive system in order to satisfy system requirements, a UML model is required to define combined artifacts of the application and the system on which it is applied. Classes are used for representing self-adaptive system components, MAPE components, MAPE process results, system requirements, computational models used to evaluate fulfillment of system requirements, and a system’s physical configuration. Attributes hold data relevant to the MAPE-K and self-adaptive system components, such as a name to identify an instance of a class, sensor measurements about the state of the external environment, and other key data that are necessary for MAPE to interact with a CPS or necessary for the specifications of CPS operation. Further, UML concepts can be extended via metaclasses called stereotypes. Stereotypes are defined as extensions of base metaclasses or specializations of stereotypes that already exist [11]. Stereotypes can be used to define extra metaclass information that can hold special meanings that are domain specific.

However, UML models cannot be used to validate the properties of a given instance of the combined system that it describes. The Eclipse Model Framework (EMF) provides API for unifying model generation via Java interfaces, XML Schema, and UML diagrams [12]. Models of systems designed with EMF can be abstracted further into metamodels in Ecore, which itself is an EMF model [12]. Ecore, EMF, and a UML editor can be used together to create the domain model, convert the domain model into an Ecore model, then generate xml files containing persisting
information for instances of classes in the domain model. The mechanism allows model instance objects to be created from a domain model, then populated with information for each attribute, including associations between instance files. When the associations are completed, a model instance is established. In relation to analyzing the combination of physical, software, and other components of a CPS (including an application of MAPE-K), the conversion of a UML file describing the system into an Ecore model, which can produce instances of components, further refining the potential for analysis of a specific CPS model.

2.3 Goal Structuring Notation and Assurance Cases

Goal Structuring Notation (GSN) is a style of notation in industry, used to formally structure ACs over the ability of software systems or parts of software systems in some environment, to fulfill their requirements. In GSN, argument claims, such as the statement that a requirement of a system being argued is satisfied, are represented by rectangular boxes. Contextual information about goal statements is represented by a capsule, evidence collected from lab tests or the field (solutions) are represented by circles, logical step strategies are represented by parallelograms, assumptions made about the argued system at a goal statement are represented by ovals so are justifications for goal statements. An argument’s overall structure is a tree-like graph, where the primary goal of the argument is the root, goals may have contexts, justifications, assumptions, other goals, strategies, and solution nodes as children. Strategies may only have goals as children, but the other types of nodes are terminal. These nodes are connected by two types of arrows: called SupportedBy and InContextOf. InContextOf arrows, are hollow arrows that point from a goal to an assumption, a goal to a justification, or a goal to a context. The SupportedBy arrows are solid and point from a goal to a strategy, a goal to another goal, or a goal to a solution node. There are further representations that exist for these components, which allow abstraction of
an argument’s form. An example of such an abstraction is the case of a solid circle being exhibited near the arrowheads of the two relationship types. The solid circle represents multiplicity of the connection from a parent node to a child node, meaning that when a concrete argument is made, there can be multiple instances of the child node. However, the statement of each child should be related to different system artifacts. By the above rules, an argument is assembled from these components into a tree structure, where the first goal is the root. Figure 2 below illustrates the described nodes and relationships of GSN.

![Figure 2 GSN nodes and relationships of an Assurance Case.](image)

An AC is an instance of one GSN tree, that describes an assurance argument over a software or physical system. ACs are concrete arguments, so should not have multiplicities or other abstract components. Further, ACs have specific labels according to the assertion of their arguments. For instance, ACs that argue over a system being safe are called safety cases [7]. In an editor platform such as Astah, stored GSN elements of an AC XMI file have defined attributes to connect the nodes. Elements carry unique id strings to identify them, allowing connection ends to possess the id values of the nodes they connect. Under the formatting of this platform, the ends of
relationships are called source and target. The target attribute holds the id of a parent GSN node, while source possesses the id of a child node. In the case of a Strategy, they are not considered parent or child nodes. Instead, they contain an attribute describedInferences, which holds space-separated id strings for each relationship they mediate a connection between a parent and child. With this serialization of GSN elements, a useful guideline exists for programmatically reproducing an AC through OOP to automate a method of AC analysis.

2.4 Dempster-Shafer Theory and Assurance Confidence Measurements

Dempster-Shafer theory is a generalization of Bayesian theory of subjective probability, formulated first by Arthur Dempster, then by Glenn Shafer [13]. Rather than emphasizing a single probability, Dempster considered lower and upper bounds on a combination of independent evidence instances with objective probabilities [14]. Then, Shafer reinterpreted the lower bounds of probabilities as degrees of belief in a new subjective probability context and created axioms to be fulfilled in the form of belief functions [14]. These belief functions were constructed to measure belief in evidence instances by including what is known about them, and the degree of what is unknown.

The authors of [10] considered a novel framework to apply these belief functions to assess confidence about safety arguments, given what is known and reasoning what is unknown about the evidence that supports them. The functions they formulate can be presented with the following example regarding claims. Let there be a claim A about the system. By the formulations in [10], there is a frame of discernment \( \Omega_A \) that is the superset of all possible values of A, \( \{ A, \bar{A} \} \) where \( \bar{A} \) is the negation of the truth of A’s claim. Then, a mass function \( m(A) = bel_A \), calculates the belief in the hypothesis that truth lies in claim A. The disbelief, or rather ignorance, of A can be quantified by \( m(\bar{A}) = dis_A \). The total probability of claim A must be one, so the uncertainty in the assessment
of the claim is \(1 - bel_A - dis_A\). In the formulation of D-S theory by [10], the results of the functions are combined in a three-tuple trust score \((bel, uncer, dis)\), which is applied to each goal node of an AC. The formulation further requires a two-tuple \((dec, conf)\) for certification purposes. It contains a certifier’s decision about a claim and the certifier’s confidence about their decision, respectively [10]. The authors of [10] give a procedure for converting this two-tuple to the above three tuple via equation (1), considering claim \(A\). Additionally, a final trust score is considered in [10] to be this two-tuple, so a procedure formulated as equation (2) is created by the authors to convert for claim \(A\).

\[
\begin{align*}
bel_A &= conf_A \times dec_A \\
\text{dis}_A &= conf_A \times (1 - dec_A) \\
uncer_A &= 1 - bel_A - dis_A
\end{align*}
\]

\[
\begin{align*}
conf_A &= bel_A + dec_A \\
\text{dec}_A &= bel_A / (bel_A + dis_A), \text{ if } bel_A + dis_A \neq 0 \\
\text{dec}_A &= 0, \text{ if } bel_A + dis_A = 0
\end{align*}
\]

Further, types of AC arguments called dependent and redundant are considered in [10], due to the differences in how calculations are performed. An example such as that in [10] can be taken illustrate differences. Suppose two subclaims \(B\) and \(C\) support \(A\). According to [10], a dependent argument is one in which for the contribution of \(C\) to \(A\)’s trust score is reliant upon the contribution of \(B\). A redundant argument is one in which there is some overlap of contributions between \(B\) and \(C\) to \(A\), such that there is no dependency between them. Let \(A\) have the claim, “The system X is correctly designed.” Then let \(B\) have the claim “The system design tolerances are sufficiently tested” and \(C\) have “The system design tolerances are sufficiently verified.” The supporting claims have similar claims about how the design is evaluated, but involve two forms of evaluation, so an overlap exist between the subclaims but are distinct from each other. These
types also define how weights are distributed onto the relationships between $A$ and its subclaims. In [10], a parameter $c_A$ called the degree of correspondence (for support of subclaims to claim $A$) is defined as $c_A = 1 - w_B - w_C$ for the weight of $B$’s contribution being $w_B$ and that of $C$’s is $w_C$. For the cases of a fully-dependent or fully-redundant arguments, the degree of correspondence is zero, so the sum of the weights is zero [10]. For the first case, $B$ cannot contribute to $A$ without $C$ and in the second case, either subclaim can contribute totally to $A$ [10].

Regarding arguments about CPSes self-adapting with MAPE-K, it can be assumed the argument is a dependent type. This is can be assumed because it is required that MAPE is implemented and tested well enough to accurately perform its four steps to maintain a system configuration that adheres to the requirements it must fulfill. If the Analyzer does not perform properly, the incorrect physical configuration of the system may be engaged, violating the assurance MAPE is supposed to provide for a self-adaptive system. However, [11] makes the case it is not possible to infer complete trustworthiness of a claim based on its children according to their relationship weights, so one other factor is applied, $v$. The factor $v$ is called the discounting factor, which represents the uncertainty in all subclaims [11]. Further, the degree of correspondence, is now called $co$ in [11], provides the same concept as in [10]. The equation (3) depicts general dependent argument child claim contribution aggregations to claim $A$ in [11], which expands on the aggregation rules in [10].

\[
\begin{align*}
\text{bel}(A) &= v[(co) \prod_{i=1}^{n} g_i + \sum_{i=1}^{n} g_i w_i] = g_A \\
\text{dis}(A) &= v[(co)[1 - \prod_{i=1}^{n} (1 - f_i)] + \sum_{i=1}^{n} f_i w_i] = f_A \\
\text{uncer}(A) &= 1 - g_A - f_A
\end{align*}
\]
Here, the authors of [11] use a subclaim index $i$ that ranges from 1 to $n$, depict the disbelief of the $i$th subclaim as $f_i$, depict the weight contribution of the $i$th subclaim as $w_i$, and depict the belief of the $i$th subclaim as $g_i$. The formulation is intended by the authors for application of calculations of ACs with disjoint weights applied to relationships. The focus on disjoint weights, quantifying the uncertainty in trust calculations, and consideration of dependent argument types, the combination of formulations of [10, 11] provide a basis for analyzing the best method to calculate trustworthiness of CPS ACs in the field, while attempting to be more mindful of the ignorance that can exist about a CPS.
CHAPTER 3

IMPLEMENTATION

3.1 Overview

The framework of this thesis emphasizes a design-time analysis of arguments made over different possible physical configurations of a self-adaptive CPS with any MAPE-K feedback loop application. The framework is implemented in Java, to leverage the power of object-oriented programming. Figure 3 below shows the flow of the project, which is split into two phases. The first phase is a learning phase, which is the primary focus of the project. The goal of the learning phase is to apply a technique that finds the best argument weights to measure confidence of an AC in field, via D-S theory. The first activity takes as input AC files that represent unique argument structures that argue satisfaction of CPS requirements, ranked CPS physical configuration files with initial evidence certifications that are related to the input ACS, and a CSV file that maps the entries of these files to each other. This mapping of many configurations to one AC was chosen, since many configurations can relate to a mutual argument structure, so it is not necessary to have multiple representative ACs for the same argument structure. Thus, memory space is saved, and redundancies can be avoided.

The phase also takes as input from the user’s console the values of the constants $\nu$ and $co$ from D-S theory, as well as a step value that is used to find all weight configurations for an input AC. The parsing activity parses the files related to this input, then stores their data in Java Map and List data structures, to be used in the phase’s calculation step. Specifically, for AC files, they are read as Document Object Model (DOM) trees, via an imported API. The elements of the files create Java AC instances. Ranked CPS physical configurations with initial evidence scores are
parsed via a Java Scanner instance, then the configuration evidences are mapped to their AC. The best weight configuration calculator activity takes these data structures output from the parsing activity as inputs, then finds the best weight configuration for each Java AC instance. The output of this activity is an internally saved map, which maps ACs to their best weight configurations, as well as CSV files that contain D-S theory calculations for each physical configuration, weight configuration, and AC. The best weight configuration maps become input to D-S theory calculations in the dynamic AC loop phase, as it is executed.

Figure 3 Design flow of the dynamic AC analyzing framework.

The dynamic AC phase is executed in a loop. The dynamic AC editing activity it takes as input from the user a CPS domain model file, model instance of the domain, a model instance artifact file that could cause physical configuration change, and an initial AC that argues the input model instance. If the artifact change requires a CPS configuration change, a new configuration artifact is also supplied. The activity then restructures the AC file according to related artifacts.
with the new configuration. The AC file is changed by replacing old artifacts with the new ones associated with the new CPS configuration file. The output of this activity is a new AC. Then D-S theory calculation activity takes the new AC as input, as well as the best weight configuration for that AC’s structure and an input evidence score file. The activity performs a D-S theory calculation on the new AC with this input. The activity outputs a CSV file containing an AC D-S theory calculation result, then the user is prompted either continue these activities or end the loop.

3.2 Java AC Node and Scoring Implementation

Since the D-S theory formulations in [10, 11] rely on three of the GSN class structures described in [7], an abstracted version (AGSN) of the metamodel is necessary. The model of AGSN is given in Figure 22 of the Appendix. Not only are the abstracted versions of GSN elements designed, but additionally, classes that represent the tuples of [10, 11], a visitor class implementations for performing D-S theory calculation of AC, a class representing a rule for inferring sub argument calculations from a parent goal, and a factory method pattern implementation for creating AGSN AC node and relationship instances. The SupportedBy relationships are given a weight attribute, which references a Wrapper class for double floating-point values. This design feature was chosen to allow a visitXXX or accept method involving the visitor and visitable nodes to return and Object type result, allowing flexibility of the information returned being either a TwoTuple, ThreeTuple, or a Double, which are used to calculate the three-tuple of a parent goal.

A visitor implementation performs the D-S theory trust calculation according to the formulation in [10]. There are only three methods implemented in the ConfidenceVisitor_v2 class, due to the three key GSN structures for calculating AC trust are Goals, Solutions, and SupportedBy relationships. The visitor class possesses two attributes to hold the $v$ and $co$ parameters from the
D-S theory formulation of [11]. The visitGoal algorithm is shown below in Figure 4. When calculating trust, the visitor is accepted by the root goal of the Java AC tree. As a goal is visited, three conditions are considered. At the line 10, the first if-then condition checks for an incomplete AC branch, to mitigate issues due to incorrect AC construction. If so, failsafe default input is stored in the result variable, which references new ThreeTuple instance. Here, it is assumed an input AC has one solution node per leaf goal. In the next block, if a goal is a leaf goal, the outgoing relationship between the parent goal and child solution is first visited, then the child solution. The result variable receives a converted three-tuple score from the return of accept(). If goal is an internal goal, then a for-loop executes at line 18, for each outgoing relationship from it. In the loop, the weight of the current relationship is first extracted, then the source of the relationship accepts the visitor, traversing the AC branch and passing its returned tuple to childTuple. The variables belSum, belProd, disSum, and disProd accumulate the summations and products of equation (3). Following the for-loop, the dis, bel, and uncer of result are set according to the computations in
equation (2) as sourced from [10]. The result reference is associated to goal, then returned.

```python
1: procedure visitGoal(goal)
2:   result ← new ThreeTuple instance
3:   childTuple ← NULL
4:   weight ← NULL
5:   belSum ← 0.0
6:   belProd ← 1.0
7:   disSum ← 0.0
8:   disProd ← 1.0
9:   //check if goal has no children
10:  if goal.targetOf.size = 0 then
11:     result.bel ← 0.0
12:     result.dis ← 1.0
13:     result.uncer ← 0.0
14:   else if goal.targetOf.size = 1 and goal.targetOf[0] is a Solution then
15:     //let goal.targetOf[0] accept visitor
16:     result ← returned value from goal.targetOf[0].source accepting visitor
17:   else
18:     for link ∈ goal.targetOf do
19:       weight ← returned value from link accepting visitor
20:       childTuple ← returned value from link.source accepting visitor
21:       belSum ← belSum + (weight.value * childTuple.bel)
22:       belProd ← belProd * childTuple.bel
23:       disSum ← disSum + (weight.value * childTuple.dis)
24:       disProd ← disProd * (1 - childTuple.dis)
25:     end for
26:     result.dis ← v * ((co * (1 - disProd)) + disSum)
27:     result.bel ← v * (co * belProd + belSum)
28:     result.uncer ← 1 - (result.bel + result.dis)
29:   end if
30:   goal.confidence ← result
31:   result.assignment ← goal
32: return result
```

Figure 4 visitGoal algorithm of visitor implementation for D-S theory calculations.

The visitSupportedBy() method is simply implemented. Departing from the common implementation of a visitor pattern, where child nodes tend to be visited from the perspective of a parent node, the value of weight is returned and the source is not traversed with an accept() call. But, visitSolution() is implemented to perform a two-tuple conversion calculation according to equation (1), as it was provided in [10]. The algorithm for the method is shown below in Figure 5.

As a solution node of a Java AC instance is visited, a new ThreeTuple instance is created. The solution’s initial two-tuple’s conf and dec attributes are used to set the values of the new three-tuple’s bel, uncer, and dis attributes, by implementing equation (1), how it was formulated in [10]. Then, the resulting three-tuple is returned by the method. The returned three-tuple will be
associated to the parent goal of the solution, as discussed in the \textit{visitGoal} implementation.

\begin{algorithm}
\begin{algorithmic}[1]
  \Procedure{visitSolution}{solution}
  \State \texttt{convertedTuple} $\leftarrow$ \texttt{new Tuple instance}
  \State \texttt{convertedTuple.bel} $\leftarrow$ \texttt{solution.confidence.conf} * \texttt{solution.confidence.dec}
  \State \texttt{convertedTuple.dis} $\leftarrow$ \texttt{solution.confidence.conf} * (1 - \texttt{solution.confidence.dec})
  \State \texttt{convertedTuple.uncert} $\leftarrow$ 1 - (\texttt{solution.confidence.conf} + \texttt{solution.confidence.dec})
  \State \Return \texttt{convertedTuple}
\EndProcedure
\end{algorithmic}
\end{algorithm}

\textit{Figure 5 Algorithm for visitSolution method.}

3.3 Weight Configuration Calculation

A class called \textit{WeightCombination} is implemented for finding all weight configurations of a given AC during the learning phase. It has two crucial functions for doing so, called \texttt{getAllRuleCombinations()} and \texttt{getAllWeightCombinations()}. The algorithms for the methods are shown in Figure 6 and Figure 7. In the first method, all possible combinations for each \textit{InferenceRule} of an AC, then return return. The method takes four inputs, called \texttt{start}, \texttt{end}, \texttt{step}, and \texttt{remaining}. The parameter \texttt{start} refers to the current starting index to reference on the \texttt{weights} attribute of \textit{WeightCombination}. The \texttt{end} parameter refers to the last index of the current list referenced by \texttt{weights}. The value \texttt{step}, is an increment value which is applied to each weight in \texttt{weights}, to produce a valid disjoint weight configuration according to the restrictions [10]. The \texttt{step} parameter is a real number inclusively \([0, 1]\), bounded in size by the number of fractional values that can be generated for a \textit{double} data type, so it is considered a fixed value for this framework. Then, the \texttt{remaining} parameter refers to the remaining fraction of the initial total to
which all weights of a configuration must sum. When the method first executes, the value of \texttt{remaining} should be \((1-co)\).

A base case is given on line 11, where \texttt{start} is zero, when the first element of a weight configuration has been reached. A new list \texttt{Double} list is instantiated and referenced by \texttt{list}. Then, the current value stored in \texttt{remaining} is added to \texttt{weights} element at the index \texttt{end}. The weight at \texttt{end} + 1 is removed from \texttt{weights}. At line 15, a for-loop executes to add all current weights in \texttt{weights} to \texttt{list}. The \texttt{list} is added to \texttt{coefficients_lists} before \texttt{coefficients_lists} is returned at line 19. In the else-block starting at line 8, a new 2-D \texttt{Double} list is created and referenced by \texttt{combined_lists}. Line 22, a for-loop executes for index \texttt{i} from the value of \texttt{step} to \texttt{remaining}, being incremented by \texttt{step} with each iteration. The value of \texttt{i} is added to the end of \texttt{weights}, then the element to its right is removed. The method is called recursively for arguments \texttt{start}, \texttt{end} – 1, \texttt{step}, and \texttt{remaining} -\texttt{i}, passing the result of the call to \texttt{partial_lists}. Next, a for-loop encompassing lines 26-44 is iterated for each list in \texttt{partial_lists}.

The loop starting at line 27 iterates for each weight value in the jth list in \texttt{partial_lists}. Inside the inner loop, at line 28, if a weight is determined to be close to 0.0, the inner loop is exited prematurely. Otherwise, the local variable \texttt{combinedWeight} accumulates the current weight value. After the inner loop exits, an if-statement checks the condition for the sum of weights being \((1-co)\), at line 35. The value of \texttt{combinedWeight} is multiplied by ten, then rounded to the nearest integer. The same is performed for the quantity \((1-co)\). If, the values are equal, then the local flag \texttt{isLegalCombo} is set to true. At line 38, another if-statement checks if the weight configuration is legal. If so, the list of weights at the jth index of \texttt{partial_lists} are added to \texttt{combined_lists}. The accumulator \texttt{combinedWeight} and flag \texttt{isLegalCombo} are then reset for the next outer loop iteration. After the end of the outer loop, the variable \texttt{combined_lists} should reference a listing of
all legal weight configurations for each inference rule in the scope of the current execution of the method. The variable is then returned as the method ends. After the return from the first call of the method, all weight configurations for each inference rule of an AC should be computed. These separate groups of weight configurations facilitate the process of computing all weight configurations of an AC, with the second method.

```
1: procedure GETALLRULECOMBINATIONS(start, end, step, remaining)
2:   if start > end or start < 0 then
3:     return NULL
4:   end if
5:   coefficients lists ← new list instance
6:   partial_lists ← NULL
7:   combined_lists ← NULL
8:   list ← NULL
9:   combinedWeight ← 0.0
10:  isLegalCombo ← false
11:  if start = 0 then
12:     list ← new list of double floating-point values
13:     weightst ← remaining
14:     // Remove weightst+1
15:     for i := 0 to weights.size - 1 do
16:       Add weightst to list
17:     end for
18:     // Add list to coefficients lists
19:     return coefficients.lists
20:   else
21:     combined_lists ← new list of list of floating-point values
22:     for i := step to remaining do
23:       weightst+1 ← i
24:       // Remove weightst+1
25:       partial_lists ← GETALLRULECOMBINATIONS(start, end − 1, step, remaining − 1)
26:       for j := 0 to partial_lists.size − 1 do
27:         if linkWeight ∈ partial_lists, do
28:           if linkWeight < 0.000001 then  // end loop prematurely
29:             end if
30:           combinedWeight ← combinedWeight + linkWeight
31:         end for
32:       end for
33:       // Round (10*combinedWeight) and 10*(1−co)) to nearest
34:       // integers then compare
35:       if 10*combinedWeight = 10*(1−co) then
36:         isLegalCombo ← true
37:       end if
38:       if isLegalCombo = true then
39:         // Add partial_lists to combined_lists
40:       end if
41:       combinedWeight ← 0.0
42:       isLegalCombo ← false
43:     end for
44:   end if
45: end if
46: return combined_lists
```

Figure 6 Algorithm for getAllRuleCombinations method.

In the second method, the weight configurations from the first method are combined to find all the configurations for the current AC tree. They are passed as the method argument represented by parameter vectorCombLists, along with integer indices for start and end. These integer indices represent a starting index and ending index to the input combined weight configuration lists,
respectively. In the if-statement at line 4, if the last and start indices are equivalent, the last entry of $vectorCombLists$ is returned, which is the listing of all weight configurations for the last $InferenceRule$ from the previous method. In the else statement that begins at line 6, the method is called recursively, with the $start$ argument being the incremented value of $start$ in the scope of this method call. The result, a list of partially combined weight configurations between two adjacent inference rules, is returned to $partial_lists$. From lines 9-15, nested for-loops are executed to combine partially filled weight configurations lists together. The outer loop executes for the number of weight configurations referenced at index $start$ of $vectorCombLists$.

The inner loop executes for the length of the $partial_lists$ list. Inside, an empty list is first added to the $combined_lists$ variable. Next, $combined_lists$ merges with the weight configuration at the $(start, i)$ coordinate of $vectorCombLists$. This merge serves to preserve $vectorCombLists$, while adding its weight configuration at the $(start, i)$ index to $combined_lists$ to prepare for appending with the weight configuration possessed by $partial_lists$ at the jth index. Finally, the weight configuration of the jth list of $partial_lists$ is appended to the current ending list of $combined_lists$. The result is a longer ending entry of $combined_lists$, possessing weights in order from the current configuration of the weights at $start$ index with weights at $start + 1$ index. When both loops end, $combined_lists$ should possess lists for each combination of weight configurations merged between all weight configurations of $vectorCombLists$ at $start$ index and all weight configurations from $partial_lists$. Line 17 ends the method, returning $combined_lists$. When
returning from the first method call, combined_lists should possess all combinations of weight configurations combined between each inference rule of the AC.

```
1: procedure GET_ALL_WEIGHT_COMBINATIONS(vectorCombLists, start, end)
2:       partial_lists ← NULL
3:       combined_lists ← NULL
4:     if start = end then
5:         return vectorCombLists[end]
6:     else
7:       partial_lists ← get_all_weight_combinations(vectorCombLists, start+1, end)
8:       combined_lists ← new list of list of double floating-point values
9:   for i := 0 to vectorCombLists[start].size - 1 do
10:       for j := 0 to partial_lists.size - 1 do
11:         // Add new list of floating-point values to combined_lists
12:         // Merge list of vectorCombLists[start, i] with
13:           combined_lists[combined_lists.size - 1]
14:           combined_lists[combined_lists.size - 1]
15:     end for
16:   end for
17: return combined_lists
```

Figure 7 Algorithm for get_all_weight_combinations method.

3.4 Input Format and Parsing

To read the file inputs, a Java interface ISysConfigAndACP parser is given to provide three method specifications for parsing input: 1) parseSysACPPathInput(), 2) parseACs(), and 3) getAC_configs(). The format of input files required at this stage are required to be comma separated value (CSV) files, so an implementing class CSVConfigAndACP parser is created to implement these methods. Further classes may be created to allow the parsing of equivalent information from other file formats. The first implemented method enforces CSV input is used for supplying pairs of file paths, one for a possible argument structure representative of the input self-adaptive system
and one for an associated file containing ranked and scored configurations for the learning phase. The method algorithm is described below in Figure 8.

The method takes a Java File object reference as input, to be parsed with a Java Scanner instance stream that is connected to the file. Inside the method specification, a simple while-loop is used to parse the file, split each entry by the comm delimiter, then add the AC file path of the entry into a Java List<String> class attribute called sysTrainACPaths while the configuration input file path is added to a parallel List<String> attribute called sysTrainACConfigsPaths. These attributes will be useful in the dynamic AC loop phase, when recalling the correct weight configuration to apply a trust calculation after the input system instance undergoes a physical configuration change.

```
1: procedure parseSysACPathInput(acAndConfigPathsFile)
2:   stream ← NULL
3:   inputFile ← NULL
4:   metadata ← NULL
5:   inputEls ← NULL
6:   stream ← input stream to acAndConfigPathsFile
7:   inputFile ← next line from stream
8:  //split fileInput string by commas and pass array to metadata
9:   sysTrainACPaths ← new list instance
10:  sysTrainACConfigsPaths ← new list instance
11:  while stream has a file line to read do
12:    inputFile ← next input line from stream
13:  //split fileInput string by commas and pass array to inputEls
14:  //Add inputEls0 to sysTrainACPaths
15:  //Add inputEls1 to sysTrainACConfigsPaths
16:  end while
17:  //close input stream connection of stream
```

Figure 8 Algorithm for parseSysACPathInput.

The next method parseACs() takes a list of AC file paths as input, then creates Java implemented AGSN AC trees for each path. It returns the reference to the list of the tree roots.
However, since the formulations of D-S theory in [10, 11] do not consider contributions from strategy nodes, contexts, justifications, or assumptions, the generated trees do not require them. An AGSN tree, which does not contain these structures, is generated for each input instead. An algorithm for this method is given in Figure 9. As an input list of AC path names *acFilePaths* is passed into the method, Java *DocumentBuilder*, *Document*, and *Element* variables are declared for parsing each XMI file. At line 15, a for-loop iterates for each file path in *acFilePaths*. Inside the for-loop, a DOM tree of the file is built with the current path string and passed to the variable *doc*, then the root element of the file is referenced by the local variable *docRoot*. The GSN AC elements stored in the document are extracted from *docRoot* by the element tag name “argumentElement,” as formatted by the Astah editor. The list of extracted elements is saved to the local variable *acElements*. Next, temporary storage variables *rootId* and *acRoot* are initialized to null values, in preparation of the task for finding the root goal of the input AC.

At line 23, a for-loop iterates for each element in the GSN XMI element list from *docRoot*. Inside the for-loop, the next element of the list is referenced by the variable *temp*. The information of *temp* is used to construct a Java AGSN node instance that is passed to the local *agsnNode* variable if the element refers to a GSN node that exists in the abstracted metamodel, or pass a null value to *agsnNode* if otherwise. At lines 26-35, nested if-blocks check if *agsnNode* is the root goal. First, *agsnNode* is checked if it is not null. If so, *agsnNode*’s type is examined to determine if it is a *Goal* type. If it is a goal, another if-statement checks if *rootId* is null. If so, *rootId* is set with the *id* attribute of *agsnNode* and *acRoot* is set with *agsnNode*. If not, the id of *agsnNode* is compared to *rootId*. At line 31, if *agsnNode*’s id lexicographically precedes *rootId*, then *rootId* is set with the id of *agsnNode* and *acRoot* is set with *agsnNode*. 
After the outer if-block ends at line 35, \textit{agsnNode} is mapped to the element id string in \textit{temp}, in \textit{agsnNodeMap}. The last if-block of the loop at line 37 checks if \textit{agsnNode} is a SupportedBy type. If so, \textit{temp} is stored into a list of relationship elements, called \textit{relationshipEls}. This is done to connect all relationship instances to goal and solution nodes, forming the AC tree in the next step. As the end of the inner for-loop at line 41, a for-loop executes for each element \textit{rel} in \textit{relationshipEls}, For each relationship, the Java instance is retrieved from \textit{agsnNodeMap}, then the \textit{source} and \textit{target} attributes of the Java instance are associated to the AGSN nodes mapped to the \textit{source} and \textit{target} id strings of its XMI file element counterpart. When the inner for-loop at line 47 ends, \textit{acRoot} is saved to local root goal list \textit{acRoots}. Last, \textit{acRoot} is mapped to the current
AC file path in the class attribute pathGoalMap. After the outer for-loop ends, acRoots is returned.

```java
1: procedure parseACs(acFilePaths)
2: acRoots ← new list instance
3: agnNodeMap ← NULL
4: relationshipElts ← NULL
5: pathGoalMap ← new map instance
6: acRoot ← NULL
7: agnNode ← NULL
8: temp ← NULL
9: db ← NULL
10: doc ← NULL
11: docRoot ← NULL
12: acElements ← NULL
13: rootId ← NULL
14: //set db with a new DOM builder
15: for acFilePath ∈ acFilePaths do
16:   //set doc with a connection to acFilePath with db
17:   docRoot ← doc.documentElement
18:   //set acElements with all elements in docRoot with tag name "argumentElement"
19:   acRoot ← NULL
20:   rootId ← NULL
21:   agnNodeMap ← new map instance
22:   relationshipElts ← new list instance
23:   for i ← 0 to acElements.length − 1 do
24:     temp ← acElements.item(i)
25:     //create AGSN instance of temp and give to agnNode
26:     if agnNode ≠ NULL then
27:       if rootId = NULL then
28:         rootId ← agnNode.id
29:         acRoot ← agnNode
30:       else
31:         if agnNode.id < rootId lexicographically then
32:           rootId ← agnNode.id
33:           acRoot ← agnNode
34:     end if
35:   end if
36:   //map xml:id string of temp to agnNode in agnNodeMap
37:   if agnNode is type SupportedBy then
38:     //Add temp to relationshipElts
39:   end if
40: end if
41: end for
42: end if
43: for rel ∈ relationshipElts do
44:   //connect all relationships by xml:ids of source and target nodes
45:   //Add rel's relationship instance to sourceOf list attribute to its source node
46:   //Add rel's relationship instance to hasContain list attribute to its target (expected as goal)
47: end for
48: //order all branches of acRoot by lexicographical order of sources
49: //Add acRoot to acRoots
50: //Add map entry acFilePath, acRoot to pathGoalMap
51: end for
52: return acRoots
```

Figure 9 Algorithm for parseACs method.

The final method takes a list of Java AGSN root goals as input, which will be mapped to input physical configuration trust tuples during input physical configuration parsing. A Configuration class is implemented to hold relevant information about a physical configuration, such as the string representing it, or whether the instance is deemed initially acceptable by a certifier, to track these features when later analyzing initial tuple scores. The algorithm for the method is given in Figure 10. At line 13, the internally stored list of input physical configuration file paths, called sysTrainACConfigPaths, is iterated in a for-loop for the number of list elements. Just after the loop header, the local file stream variable configTupleStream is connected to the file
of the ith path name in sysTrainACConfigPaths. The first line of the file, which contains the metadata, is read by the stream, then passes the file metadata to variable metadata.

```java
1: procedure GETACConfigs(acList)
2: acConfigs ← new map instance
3: fileLineEls ← NULL
4: metaData ← NULL
5: configTupleStream ← NULL
6: configFileInput ← NULL
7: tupleString ← NULL
8: tempConfig ← NULL
9: configString ← NULL
10: configTup ← NULL
11: initConfigRankMap ← new map instance
12: configInitTupleMap ← new map instance
13: for i := 0 to sysTrainACConfigPaths.length − 1 do
14:   // connect file with path sysTrainACConfigPaths[i] to configTupleStream
15:   configFileInput ← next line from configTupleStream
16:   // split configFileInput by commas and give to metadata
17:   // Add map entry acList, new list instance to acConfigs
18: while a file line can be read with configTupleStream do
19:   // split configFileInput by commas and give to fileLineEls
20:   tempConfig ← new Configuration instance
21:   // fileLineEls[0]
22:   configString ← fileLineEls[0]
23:   tempConfig.configString ← configString
24:   // Add map entry tempConfig, new map instance to configInitTupleMap
25:   for j := 1 to fileLineEls.length − 3 do
26:     configTup ← new TwoTuple instance
27:     tupleString ← fileLineEls[j]
28:     // remove parentheses from tupleString
29:     // split tupleString by semicolon delimiter, set
30:     // configTup.dec to floating-point conversion of first split
31:     // element, and set configTup.conf to floating-point
32:     // conversion of second split element
33:     // Add map entry metaData[j], configTup, key to tempConfig in configInitTupleMap
34:   end for
35:   tempConfig.acceptable ← boolean conversion of fileLineEls[fileLineEls.length − 2]
36:   // Add mapping of tempConfig to integer conversion of fileLineEls[fileLineEls.length − 1] in initConfigRankMap
37: end while
38: end for
39: acConfigsMap ← acConfigs
40: return acConfigs
```

Figure 10 Algorithm for getACConfigs.

At line 18, a while-loop causes the current file stream to be read while there is still an entry to read. Inside the while-loop two lines below, the entry is split by the comma delimiter into an array of strings, then passed to the variable fileLineEls. The first string of the split is expected to be the physical configuration string of the entry and is stored in a new Configuration instance called tempConfig. Then at line 25, a for-loop iterates from the second element to the second-to-last element of the fileLineEls, which are expected to be evidence two-tuple scores delimited by semicolons. For each two-tuple string, a new TwoTuple instance is created, the tuple is split, then the
dec and conf string values are converted to floating-point values and stored in the TwoTuple instance’s dec and conf attributes. After the for-loop ends at line 34, a certifier’s determination of the physical configuration being acceptable is saved to the Configuration instance. The last value of the entry of fileLineEls is the observing certifier’s ideal rank about the produced configuration, based on their scoring judgement for its argument’s evidence. It is mapped to the tempConfig in the class attribute initConfigRankMap. In the final line of the while-loop, tempConfig is added to a list of configuration instances, which is later mapped to the AGSN AC related to the current configuration file, in the variable acConfigs. After the while-loop and for-loop exit, acConfigs is stored in a class attribute acConfigsMap, to be referenced later during the learning phase calculations.

3.5 Learning Approach

The main motivation of the thesis requires careful consideration about how to evaluate results of D-S theory calculations on initial CPS scenario data, to inform how the same CPS should be certified in related scenarios during deployment. When generally used in conjunction with a CPS project, this framework will first parse initial CPS configuration data that is associated with a configuration verification step at the design-time of the CPS. The configurations output from a verification step will allow better input evaluation accuracy, and in turn, increased accuracy of AC trust and confidence during deployment. Under this assumption, a certifier can examine the results of the verification step, such that they can assign trust (dec, conf) scores for each evidence node of the AC arguing over the CPS, with each physical configuration. A calculator class called TrainingACCalculator2 is implemented to find the best weight configuration for each input AC. This calculator contains a crucial method findBestWeights(), to perform the best weight configuration search. This method takes no arguments, due to required information being stored
in the parsing class described above. The algorithm for `findBestWeights()` is shown below in Figure 11.

```java
1: procedure findBestWeights()
2:  //initialize local variables and other attributes
3:  for agsnAC in trainingParser.acConfigsMap key list do
4:      solnList ← all solutions in agsnAC
5:      for j := 0 to acTreeRules.size − 1 do
6:         weights ← new list instance
7:         //Add 0.0 to weights for each relationship in acTreeRules
8:         ruleCombs ← getAllRuleCombinations(0, acTreeRules, contain.size−1, this.step, 0, 0)
9:      end for
10:     combinations ← getAllWeightCombinations(ruleCombs, 0, ruleCombs.size−1)
11:    for j := 0 to combinations.size−1 do
12:       //Add map entry to rankedTrainingScores
13:       for k := 0 to combinations.size−1 do
14:          //Add relationships, weight ← combinations[jk]
15:       end for
16:      end for
17:     for config related to agsnAC in trainingParser.acConfigsMap do
18:        for soln ∈ solnList do
19:           soln.confidence ← TwoTuple stored in trainingParser.confInitTupleMap
20:           rootTup ← result of agsnAC accepting confVisitor
21:           convertedTup ← new TwoTuple instance
22:           if (rootTup.lex + rootTup.size) = 0.0 then
23:            convertedTup.conf ← 0.0
24:           else
25:             convertedTup.conf ← rootTup.lex + rootTup.size
26:             convertedTup.conf ← rootTup.conf/(rootTup.lex + rootTup.size)
27:           end if
28:        end for
29:       //Add to rankedTrainingScores
30:      end for
31:  end for
32:  //rank all calculated tuples related to combinations in rankedTrainingScores
33:  for i := 0 to size of list mapped to combinations in rankedTrainingScores do
34:     if initialRank ≠ (i + 1) then
35:        misses ← misses + 1
36:     end if
37:   end for
38:  if misses < f penaMisses then
39:     f penaMisses ← misses
40:     bestWeights ← combinations
41: end if
42: //write configuration, dec, conf, and rank output for each ranked configuration
43:  misses ← 0
44:  //Clear map entries from trainingACConfInstances
45:  //Clear map entries from rankedTrainingScores
46: end for
```

Figure 11 Algorithm for findBestWeights.

The method relies on three main loop scopes, as shown in lines 3-47, with other loops that provide additional tasks to facilitate these three loops exist within these scopes. The outermost loop at line 3 iterates for each representative argument `agnsAC` supplied in the parsing step, instantiated as a Java AGSN tree. For each iteration, a new output CSV file is created and the metadata for expected output is written in the first row, through the output stream `trainingOutput`. Next in lines 5-9, the total weight configurations for each sub argument are computed. At line 10,
the total weight configuration assignments that could be given to the current tree structure are computed, then given to combinations. The second main loop at line 11 iterates for each weight configuration computed.

At line 16, the inner main loop iterates for each parsed input CPS configuration related to the current tree. In this loop, the D-S theory score is calculated with a visitor pattern instance confVisitor, which produces a root three-tuple at the end of visiting the current AGSN tree. The result is converted into a (dec, conf) two-tuple, then stored into a list. After the end of the innermost main loop at line 31, the list of all physical CPS configuration computations for this weight configuration is sorted according to dec as the priority value, then conf. The stored rankings for each physical configuration from the parser are compared to their orderings in the current score list, from lines 33-38. Any point in which an ideal rank from the input is not matched with the current physical configuration score rank, the misses counter is incremented. The number of missed ideal scores are compared to the fewest misses found so far. If the number of misses for this weight configuration is less than the current fewest, the fewestMisses counter is updated and the current weight configuration is considered the best. The number of misses is reset to zero in preparation for the next middle loop iteration. After the middle main loop ends at line 20, the currently considered best weight configuration is mapped to the current tree structure, to be used later in the dynamic AC phase. The best weight configuration reference and fewestMisses are reset, to begin the process again for the next argument structure. Once the outer loop ends, the method ends execution. The result should be stored mappings of the best weight configuration results mapped to their respective argument representatives.
3.6 Dynamic AC Evaluation

The dynamic AC evaluation process is described below in the flow diagram Figure 12. In the main class level of the project, the phase consists of a do-while loop, requesting an input domain model of a CPS, a related model instance, an artifact whose value is changed, an input AC file describing the model instance, and an output file path for the AC that saves the dynamic change of the input AC. These inputs are saved to attributes of an IMaintenance implementing object. The interface is an addition made long after the original framework implementation by Dr. Lin. The original framework of the legacy project contained two important classes Maintenance and AssuranceCaseParser which were used to change features of an AC when artifact changes cause a change to the AC structure and to generate an AC instance from a file. However, the original implementations would have to be modified to allow D-S theory calculations after AC changes occur. Due to this issue, additional interfaces IMaintenance and IAssuranceCaseParser were created, to allow different implementations of Maintenance and AssuranceCaseParser.

So, additional classes MaintenanceExtended and AssuranceCaseParserExtended were created to add features, without modifying the dynamic AC framework’s original classes. It should be noted that in these classes, novel source code was preserved from the original class counterparts. The extending features are the modifications necessary to perform D-S theory trust calculation after a physical configuration change occurs and to build an AGSN AC for such a calculation. The actions provided by these classes are described below by the flow diagram in Figure 12. In the loop, the user must provide on console the domain model path of the CPS whose ACs were used to perform the learning phase trust calculations. Then, the user must supply the path of an initial system model instance that is subject to changes due to environmental conditions. Next, the path first artifact whose value changes is given. In the case of a CPS that uses sensors to collect some
environmental data, any monitored attribute of a sensor would be related to such an artifact. Last, the path of the AC arguing the current physical makeup of the system instance is supplied, as well as an output path name used to generate an output file containing AC changes.

Once the input is accepted, the files of these inputs are loaded. The first changed artifact has its monitored value changed according to the value of the input artifact file, then the user is asked if this change affects other artifact of the system instance file. If not, the user is asked if they want to continue the dynamic AC loop. If the user chooses not to continue, the loop and program end. Otherwise the loop begins with model and model instance collection from the user. If an artifact affects other artifacts, the next step of the phase searches for all affected artifacts. Affected artifacts are determined by stereotypes applied to relationships with other affected artifacts, according to artifacts designed by Dr. Lin. For the classes of primary affected artifacts, they have a stereotype MonitoredAttribute applied to their value attributes. Then, classes with this stereotype

Figure 12 Flow diagram of the dynamic AC loop phase.
have a stereotype called *Conditional* applied to them. Other classes associated to these may have stereotypes applied to associations ends. The stereotypes *To* and *From* are added to directed associations ends between classes that should be affected by artifact changes. Further, the *Owner* stereotype is applied in this same context, but is applied to the owning end of a composition association.

When checking for affected artifacts, the properties of *To* and *Owner* are used to define secondary affected artifacts, according to Dr. Lin’s design of the base framework applied to this project. Due to this stipulation, input UML domain diagrams must have the above described stereotypes for correctly performing the affected artifact search. Through iterating all associations of the model instance artifact files, the complete listing of artifacts is stored. Next, the user must supply a new configuration artifact file, that will be newly associated into the instance model, replacing the old configuration artifact. The new configuration should have associations to other artifact files, to completely replace the old artifacts, otherwise changes to the AC structure may cause branches to be incorrectly edited. Once the old artifacts are replaced with the new artifacts, changes to the system’s AC structure are performed. The applications of node and relationship changes were implemented by Dr. Ling according to the strategy described in [8], where AC nodes contain in some pare of their data the names of artifacts to which they relate. The attribute *description* is what was chosen by Dr. Lin to possess a related artifact’s name.

So, the AC nodes related to the old artifacts are removed from the AC file, along with connected relationships. Then, by tracking the structure of the removed sub arguments, replacement arguments are added, but with new nodes and associations according to the addition of the new artifacts. The similarly structured replacement branches are added into the AC, which is then saved to the output file, whose path name was given at the beginning of the loop. Following
the saving of the new AC, the applied modification to the extension of the base framework is executed, where the AGSN Java instance of the new AC structure is generated and stored. Next, the user is prompted to supply a file containing solution id-input tuple entries for each solution of the new tree and the name of the supplied input representative AC from the learning phase, which shares the same argument structure as the new AC. The input learning phase AC path is required, to allow the program to know which best weight configuration to apply during D-S theory calculation. It should be noted the modifications expects a text file input format for this file but is implemented in such a manner to allow parsing of other file formats.

After the file entries are parsed, the D-S theory trust calculation of the new AC is performed. The visitor from the learning phase is associated into this phase, so the choice of $v$ and $co$ from the learning phase are applied here. The input AC path from the tuple file is used to retrieve the best weight configuration for the new AC, then the weights are applied to the correct relationships. The tuples are applied to their correct solutions in this step as well. Once set up, the AGSN instance of the new AC then accepts the visitor to retrieve the final three-tuple of the root. The three-tuple is converted to a two-tuple, then written to an output CSV file, which is named after the file containing the new AC. The end of the loop occurs, and the user is asked if they wish to continue the loop. If so, the process begins again. Otherwise, the loop ends, and program exits.
CHAPTER 4

SIMULATION AND RESULTS

4.1 Design-time and Runtime Setup

The approach is executed with the Eclipse integrated development environment, with a Java Runtime Environment 1.8.2. Two case studies were inspired by the ENTRUST methodology case examples in [4]. One example CPS is an unmanned underwater vehicle (UUV) to be deployed for conducting marine environmental analysis research. The UUV example was designed to mimic the scenario in [4] to have some number n of onboard sensors that measure external environment conditions, for some defined rate of accurate measurements for every ten meters. Further, each sensor has a rate of energy consumption when operating and the UUV expends energy when turning on a sensor or turning off a sensor. As the UUV operates, it travels at one of several speeds determined at design-time during verification analysis. A physical configuration of a UUV is defined in [4] to be a tuple \((x_1, \ldots, x_n, sp)\), where \(sp\) is the current UUV speed and \(x_i\) for \(i = 1,\ldots,n\) is either zero or one, referring to whether the \(i\)th sensor is inactive or not, respectively.

The example is given three nominal requirements (R1-R3) that the UUV must satisfy while operating and a failsafe requirement (R4), which must be satisfied if no physical configuration of a UUV can satisfy the nominal requirements. The requirement R1 in the UUV example requires no less than twenty accurate sensor measurements are taken for every ten meters scanned, while R2 requires no more than one-hundred twenty joules of energy are consumed by a UUV for every ten meters scanned, and R3 requires that for several sensor-speed physical configurations satisfying requirements R1 and R2, the configuration should be chosen is one which minimizes a cost calculation applied to each configuration and which has the fastest speed [4]. Regarding the
failsafe requirement R4, if no configuration satisfying requirements R1-R3 is found, all sensors are turned off and the UUV is slowed to a full stop, until a configuration exists which satisfies those requirements [4].

The instance models derived by Dr. Chung-Ling Lin consider three-sensor systems, that follow the scenario behaviors defined in the UUV example described in [4]. Stereotypes labeled MonitoredAttribute are applied to attributes in the UUV-MAPE domain model to add metadata to value attributes of Configuration and MeasurementRate classes, due to these concepts being the triggers for changes in argument structure of an AC that describes the system. This technique follows a strategy considered in [8], where model artifacts are suggested for use, in tandem with metadata in ACs. ACs were designed by Dr. Lin to support arguments stating their UUV instances were well designed and correct, according to the current system configuration captured in them. Each AC node contains a model artifact name in the description attribute of its XMI file entry, relating the node to a model class, such that successful coordination of dynamic AC entry changes can be performed.

The second case study is inspired by the FX case study of [4], which is a foreign currency exchange system. The system considered is also an FX model, but differs from the example model in [4]. The modified model created by Dr. Lin considers two sensors, one that measures the exchange rate of the peso to the American dollar and another that measures the exchange rate of the euro to the American dollar. Unlike the UUV-MAPE model, the domain model of the modified FX-MAPE system does not consider sensor models. It is simplified domain example that considers the action of AC change through the same Configuration and MeasurementRate classes applied to the UUV-MAPE case, so the same stereotype MonitoredAttribute is applied to their value attributes, to trigger the changes to an AC’s structure. Further, this case study was not given
defined requirement descriptions, but rather enforces having three generic nominal requirements R1-R3, that must all be fulfilled simultaneously, or else a generic failsafe R4 is satisfied until a determined modified FX-MAPE configuration fulfills R1-R3 again. Here, several AC instances were designed to support a model instance, supporting satisfaction of the generic requirements R1-R3, or R4.

Regarding the case studies, randomly assigned input data values for system configuration certifications were provided in CSV files under two assumptions. The first assumption asserts that a physical configuration for a CPS can possibly, but not guaranteed, be encountered multiple times. The assumption is justified based on the reported analysis of the Foreign Currency Exchange (FX) model in [4]. There are six services provided in a physical configuration given by [4], which can be considered a tuple. Each service in the tuple has two modes of operation, so there are sixty-four possible configurations [4]. The finiteness of the configuration scale allows the possibility that a physical configuration could be encountered multiple times. The second assumption is when a physical configuration is encountered multiple times, each successive configuration should have higher ratings for its evidence. The justification for this assumption in under the argument that from a human perspective, if a certain configuration is frequently chosen for the system to best fulfill its requirements, then it must be a better configuration. Thus, an engineer can be more confident in the evidence for that configuration if it is favorable over the others. As a final note, case study learning phase configuration score input and dynamic AC output for each case study are in the Appendix for additional context to the results presented.

4.2 Case Study: ENTRUST UUV System

The first demonstration of the AC evaluation approach is investigated by applying the example UUV-MAPE interaction of [4], due to a preexisting domain model and several model
instance artifacts relating to the self-adapting CPS, written by Dr. Lin. Under the Figure 13 shows the initial UUV AC argument developed by Dr. Lin, which argues over satisfaction of the three nominal system requirements for an example physical configuration, reported in [4]. Any argument encountered during the dynamic AC phase that argues over satisfaction of these same requirements is expected to have the same argument tree shape, regarding relationships between its claims and subclaims. In the AC, initial evidence scores are considered for the initial trust of the system, its sensor measurements, design-time sensor models used to calculate the probability of a sensor configuration during the verifications steps in [4], satisfaction of the three system requirements designed for a UUV system in [4], and reports that are generated during the Planner and Executor steps of the self-adaptive system. This file was also used as an argument representative structure to demonstrate the technique applied during the learning phase.

![Figure 13 An input UUV-MAPE AC arguing over nominal system requirement fulfillment.](image)

To find all weight configurations, a \( \text{step} = 0.1 \) is chosen, due to no argument subtree of either argument structure having more than four subclaims and to produce more weight configurations in an attempt to increase the resolution of a best weight configuration choice.
Further, the values $v = 0.7$ and $co = 0.1$ are initially chosen for performing the learning phase and dynamic AC phase D-S theory calculations, to investigate calculations of the system from a pessimistic perspective. These parameter values are chosen as if there is a large degree of ignorance about the system and that subclaims are loosely dependent upon each other to contribute to a parent claim. Investigation of different values for these parameters and trust scoring performance was restricted due to the computational intensity of the approach for finding a best weight configuration for dynamic ACs during the design-time of a self-adaptive system project. Two CSV files were filled with initial example UUV physical configurations. The file uuv_Training_Configurations_10Entries.csv contains ten configurations, scores, and ranks associated with fulfillment of system requirements R1-R3 of UUV. The file uuv2_Training_Configurations.csv is associated with two entries of the failsafe requirement configuration. Fewer input configurations are chosen due to computational intensity as well. It should be noted that when learning technique is applied to a self-adaptation methodology that uses MAPE-K, the input information should be sourced from results of the methodology’s verification.
step. Figure 14 below shows a sample of the final \((\text{dec}, \text{conf})\) tuple results for this AC.

![Plot showing final tuple trust scores for UUV-MAPE configurations satisfying requirements R1-R3.](image)

**Figure 14** Final tuple trust scores for UUV-MAPE configurations satisfying requirements R1-R3.

Trust scores are produced with a general spread, ranging in decision values near 0.6 to near 0.85 and confidences from approximately 0.35 to near 0.5. When examining the results of the two configuration entries for satisfaction of R4, the behavior exhibits more fine-grained clustering. Figure 15 below shows the two-tuple calculations for this case. When comparing the learning
phase calculations between entries that support satisfaction of requirement R4 to those that satisfy R1-R3, the clustering behavior is defined, with apparent regular distances between clusters.

![Final tuple trust computations for satisfaction of requirement R4 in UUV-MAPE](image)

*Figure 15 Final trust computations for learning phase UUV-MAPE configurations satisfying requirement R4.*

The dynamic AC loop phase was executed with one iteration, to perform simple demonstration of the application of a best weight configuration after the learning phase. The UUV-MAPE domain model UUV.uml designed by Dr. Lin of [11] was provided as input, as well as an initial model instance uuv.xmi, and its argument uuv_AC.xmi. An affected artifact rateS2.xmi was supplied, to cause a change in measurement rate of the sensor S2. It was chosen that the rate change violates R1-R3, such that a physical configuration change was necessary. The configuration config.xmi was supplied, which contains the physical configuration (0,0,0,0), resulting from the requirement R4. Due to the configuration change, the *MaintenanceExtended* class collected affected model instance artifacts due to changes from rateS2.xmi and config.xmi. Instance files of the requirements, the original configuration of UUV, and other related files were replaced in
associations to the model instance files associated with config.xmi, which includes a requirement file instance R4.xmi.

During dynamic AC editing, the AC nodes and branches containing names of artifacts to replace in their *description* attributes were removed, then replacement nodes and branches were added to the AC file, which is saved to uuv_R4-config_newAC.xmi. The resulting file of the dynamic AC change is shown below in Figure 16. The trustworthiness calculated following the file save, with an input file GSNData_UUV_config_tuples.txt. The result was exported to uuv_R4-config_AC.csv, where the final tuple based on the best weight configuration for this argument structure and the initial values of $v$ and $co$ is $(0.68427, 0.31939)$. By these subjective probabilities, assuming 0.5 is considered a measure of indifference, the resulting AC possesses a plausible decision potential. However, the confidence in such a decision is insignificant.

*Figure 16 Resulting AC of UUV R4 Configuration in dynamic AC loop.*
4.3 Case Study: Modified Foreign Currency Exchange

The demonstration is performed in the second round with the modified domain model of the FX-MAPE interaction of [4]. Figure 17 and Figure 18 below give initial assurance arguments designed by Dr. Chung-Ling Lin, arguing the satisfaction of generic requirements R1-R3 or R4, in the attempt of a system instance facilitating exchange interactions for the euro and peso. The parameters step, v, and co were given the same values to perform best weight configuration analysis as with the previous case study, during the learning phase and to further perform D-S theory calculations in the dynamic AC loop. The selection of step = 0.1 for finding all weight configurations to this argument structure is considered reasonable, due to the smaller tree size.

Two CSV files were filled with initial example modified FX physical configurations, as well. The file fx2_Training_Configs_10Entries.csv contains ten configurations, scores, and ranks related to satisfaction assurance of requirements R1-R3 of the modified FX model, while fx_Training_Configs.csv is contains two entries of R4 configurations. Just as with the UUV-MAPE ACs, four strategies provide logical steps between supporting claims and evidence to the root claim 0G1.1 arguing over the correctness of implementation and design of the system, one for each automaton of MAPE. Strategy 0S1.1 connects currency exchange rate data for the euro-peso exchange example, which is read by sensors.
As the learning phase is executed, all trust scores are computed for all input physical configurations satisfying R1-R3 and R4. Figure 19 below shows the total trust calculations for the modified FX argument structure for satisfaction of requirements R1-R3, while Figure 20 shows
the resulting trust score computations for physical configurations that satisfy R4. Like the nominal requirement satisfaction results of the UUV-MAPE case study, no discernable clustering is found. A clear spread of trust calculations range in confidence scores between 0.25 to just over 0.5 and decision scores between 0.35 to near 0.85. However, clustering is noticeable regarding the trust calculations for both R4-satisfying configurations, with a lower confidence range. The defined features may be partially explained by their being very few entries which satisfy R4. Fewer calculations can allow the behavior of results to become more apparent. Further, the confidence range relating to satisfaction of R1-R3 is somewhat broader than that for R4 results. This can be attributed to the scoring differences between pieces evidence supporting configurations satisfying the nominal system requirements compared to those related to the failsafe. It is expected that initial scores for evidence that relates to nominal system function should be decided with greater consideration by a certifier over evidence supporting an AC satisfying the failsafe requirement.

![Figure 19 Final AC trust scores for modified FX example configurations satisfying requirements R1-R3.](image)

48
Figure 20 Final AC trust scores for modified FX example configurations satisfying requirement R4.

The dynamic AC loop phase is run with a two-sensor modified FX-MAPE system instance that satisfies a generic requirement R4. A change is applied via rateR1.xmi, which refers to the rate measured for the euro conversion, by the first sensor. After supplying a new artifact Config.xmi, generic requirements R1-R3 are fulfilled. Initial randomized data about a certifier’s confidence in the system is given in the file GSNDData_fx_R1-R3_config_tuples.txt. The best calculated weight configuration was applied to the new modified FX-MAPE domain AC arguing over generic requirements R1-R3. The resulting AC of the system is given in Figure 21. The final tuple and best weights used to calculate the tuple are given in the output file fx_R1-R3_config_newAC.csv in the TestModels/FX/ directory. The final tuple (0.76181, 0.43042), has a slightly higher decision value compared to that of the first case study and a higher level of
confidence in the decision as well, but to a marginal degree.

Figure 21 Resulting AC of FX R1-R3 Configuration in dynamic AC loop.
CHAPTER 5
DISCUSSION

5.1 Conclusions

The framework has been demonstrated the ability to find the best weight configuration of a CPS’s AC relationships through D-S theory trust calculations and simulated how the results may be applied to the CPS as it is deployed. Comparing the learning phase results between the first case study and the second in consideration of this strategy, there are apparent differences in the calculations between the R1-R3 calculation results. One reason to consider is there are fewer relationships between the assurance cases that satisfy the nominal requirements of these two cases. Specifically for the second case study, the fewer relationships possessed can result in more weight being distributed across the arguments, since there are less supporting evidence instances that can contribute to the overall argument, compared to an AC of the first case study. Further, the number of contributing evidence instances between case studies may have an impact again. Due to having more evidence to apply to its argument structure, the UUV-MAPE AC can contribute more belief from the bottom of the AC, upward, causing differences in behavior from an AC argument structure that would have less evidence.

However, the accuracy of such certifications is subject to the choices of the parameters of $v$ and $co$, as well as the input evidence applied to a configuration. Further calculations varying $v$ and $co$ are required to determine a best overall manner of applying D-S theory calculations, but assumptions about level of ignorance or level of correspondence between subclaims in a subargument can only contribute so much to the inaccuracy of calculations. The primary source of inaccurate calculations will be the input two-tuple scores applied to evidence for a CPS.
configuration. Further, it is ultimately a system’s certifier’s decision to determine the value of these parameters. Given an environment in which a CPS is deployed, only so much information can be obtained during design to inform how much ignorance over all claims can be given.

Another issue concerns \textit{step} for computing all weight configurations to an AC. In the pure mathematical sense, \textit{step} is a real number. So, since \textit{step} ranges between [0,1], it potentially has an infinite number of values to hold. An approximation had to be performed, fixing the value of \textit{step} to give a set of weight configurations. Another strategy for computing all weight configurations for an AC could be more efficient or give better certifications during design-time. Following this note, the condition for the best weight configuration choice may also be improved. Since the first weight configuration with the fewest missed ideal input CPS configuration ranks is chosen, any weight configuration with the same number of misses is not considered. Having additional criteria such as comparing how far missed ranks are distanced from the ideal ranks could strengthen the approach as a reasonable strategy.

5.2 Difficulties

While designing the framework and collecting evidence, several ideal implementations and testing scenarios could not be achieved due to the computational intensity of the learning phase, or the unavailability of data. In an ideal case, further study into the learning phase by varying parameters \(v\) and \(co\) would be preferable. However, the ability to do so was found to be restricted by the size of an input AC. Noting the first case study, the number of weight configurations that could be applied to analyze either argumentation structure significantly increased the runtime of the learning phase. For the case of ten configuration entries, most of the seven minutes of calculation time taken by the learning phase was due to measuring the final trust scores for each configuration, with each weight configuration. Given that both \(v\) and \(co\) can have values
inclusively from zero to one and further restricting incrementing of values by 0.1, there are one-hundred twenty-one combinations of parameter values to explore in analysis. For cases of AC structures such as those found in the second case study, such analysis would be tenable.

Further, difficulties were encountered when attempting to apply input data more realistically found during design-time verification steps CPSes. For the case of the verification engine found in [4], a virtual machine is implemented to use Markov models to produce system configurations based on probability of the model state transitions and calculate whether nominal system requirements are satisfied. Due to time constraints on the ability to extract data from a verification engine, random trust scoring was applied to configurations inspired by the analyses done in [4]. Thus, the veracity of the calculations performed by the framework was tested under the quality of input.

5.3 Future Work

Given the difficulties and conclusions above, there are several avenues to expand this work. First, the variation of $co$ can be performed on the case studies considered for this framework. Using the results from this expansion, the most effective value of $co$ could provide better insight into stronger ties between subclaims in the arguments of these CPSes. Then, the formulation of a better strategy for finding the best disjoint weight configurations is required, to provide less resource intensive searches. After this formulation, a realistic analysis on this framework can be provided by examining verification engines of existing CPS studies, focusing on input configuration data based on models that consider deployment environment factors that could influence a certifier’s judgement regarding initial AC evidence scoring. Once enough data has been collected to consider the improvements reasonable, the ultimate goal is to test the design-time support of this framework.
with MAPE-K based self-adaptation tools, and facilitate dynamic AC analysis by integrating the D-S theory computation approach into the implemented MAPE feedback loop.
Figure 22 Designed abstracted version of a GSN metamodel.

Figure 23 uav_Training_Configurations_10Entries.csv learning phase input data.
Figure 24: uuv2_Training_Configs.csv input learning phase input data.

Figure 25: Input certification tuple file for dynamic UUV-MAPE AC.
Figure 26 Output dynamic UUV-MAPE AC certification with best weight configuration.

Figure 27 fx2_Training_Configs_10Entries.csv learning phase input data.
Figure 28 fx_Training_Configs.csv learning phase input data.

Figure 29 Input certification tuple file for dynamic modified FX-MAPE AC.
Figure 30 Output dynamic modified FX-MAPE AC certification with best weight configuration.

Figure 31 Class diagram of automatic certification support framework.


