8-2018

A Decision Support System for Selecting Between Designs for Dynamic Software Product Lines

Ethan Travis McGee
Clemson University, bulletshot60@gmail.com

A Decision Support System for Selecting Between Designs for Dynamic Software Product Lines

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Ethan Travis McGee
August 2018

Accepted by:
Dr. John D McGregor, Committee Chair
Dr. Brian Malloy, Committee Co-Chair
Dr. Murali Sitaraman
Dr. Amy Apon

Abstract

When commissioning a system, a myriad of potential designs can successfully fulfill the system's goals. Deciding among the candidate designs requires an understanding of how the design affects the system's quality attributes and how much effort is needed to realize the design. The difficulty of the process compounds if the system to be designed includes dynamic run-time self-adaptivity, the ability for the system to self-modify its architecture at run-time in response to either external or internal stimuli, as the type and location of the dynamic self-adaptivity within the architecture must be co-decided.

In this proposal, we introduce a Decision Support System, which contains a new Dynamic Software Product Line-centric cost / effort estimation technique, the Structured Intuitive Model for Dynamic Adaptive System Economics (SIMDASE), that will allow system designers / architects to select the most appropriate design for systems where the candidates can be structured as a Dynamic Software Product Line. We will focus on using the Decision Support System to select designs for a system where at least one component of the system is a low-level embedded system for use within the Internet of Things (IoT), particularly embedded systems whose purpose is to exist as "things" (either intelligent sensors or actuators).

The Decision Support System we introduce is a multi-step process that begins with a high-level system architecture generated from the system requirements and goals. Candidate designs that can fulfill all goals / requirements of the high-level architecture are selected. Each design is then annotated using SIMDASE so that the effort, risk, cost and return on investment that can be expected from the realization of the design(s) can be compared in order to select the best design for a given organization.

Dedication

"To the only God our Savior, through Jesus Christ our Lord, be glory, majesty, power, and authority before all time, now and forever. Amen." (Jude 1:25 [HCSB])

"Who is a God like You, removing iniquity and passing over rebellion for the remnant of His inheritance? He does not hold on to His anger forever, because He delights in faithful love. He will again have compassion on us; He will vanquish our iniquities. You will cast all our sins into the depths of the sea." (Micah 7:18-19 [HCSB])

Acknowledgments

Throughout this academic journey, Dr. John D. McGregor has provided countless hours of advice, proofreading, ideas and encouragement. I thank him greatly for his help throughout every phase of my Ph.D. career. In addition I would like to thank my fellow Ph.D. colleagues, Yates Monteith, Fryad Rashid and Wanda Moses, for their willingness to listen and offer constructive feedback concerning my research ideas. Finally, I would like to thank my dissertation defense committee for both their willingness to serve and for the constructive criticism offered throughout the defense process.

I would also like to thank everyone who has offered encouragement and support throughout my entire graduate career. This includes my parents, my church family, my former co-workers at Transplace, my professors at both Bob Jones University / Clemson University and my close friends.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Dynamic Software Product Lines (DSPLs) [28] are a variation of Software Product Lines (SPLs) [13] that can be used to represent Dynamic Adaptive Systems (DASs) [18, 38], systems that are able to monitor the environment in which they reside and self-adapt in response to changes in that environment [5]. Even though they are a variant of SPLs and share many of the same concepts, DSPLs have several unique facets that differentiate them from SPLs. First, DSPLs focus on a local environment and respond to changes in the environment rather than changing in accordance with market desires [28]. Second, in SPLs, designers are not typically concerned with run-time variations whereas in DSPLs, designers are typically not concerned with pre-run-time variations. However, both areas acknowledge that mixtures of variation types are sometimes necessary [28]. Finally, the central tasks of each are different. In SPLs, the primary focus is to provide as wide a variety of products as the market demands reusing as much as possible. In DSPLs, monitoring the environment and, either manually or automatically, self-adapting into a new configuration are the central tasks [28].

Consider an example DAS that monitors select properties of its environment, in this case signal strengths of various wireless transmission protocols, and can reason about the measured values of those properties in order to determine what behaviors the system should adopt in order to best operate in the detected environment. Such a system is a classic example of a DAS [45].

Our example DAS will be placed into a manufacturing facility and will be used to monitor forklift drivers to ensure that 1) they are following the posted speed limits with the facility, 2) they are completing a route circuit in a timely manner, and 3) they are completing the required number

of circuits during a work shift. To do this, the DAS will periodically transmit location, speed and direction to a central server that will produce reports for management. The manufacturing facility, due to sensitive robotic machinery, has placed strict limits on which protocols are allowed within certain parts of the facility. They have also placed strict limits on which protocols are allowed overall. These accepted protocols are Bluetooth Low Energy (BLE) [26], Standard 2.4 Gigahertz Wireless, and Zigbee [33]. The DAS will be required to monitor the location of the forklift so that certain protocols can be disabled within specified zones. It will also be required to monitor signal strengths of the various protocols so that it can select the best protocol, from those currently available, for transmission.

The inclusion of dynamic self-adaptivity, or the decision to use a DSPL versus a SPL, is a decision not to be made rashly. Its inclusion can lead to partially specified requirements [25], increased complexity of the design [31], increased effort required to implement and increased time needed to test [36]. However, if included, dynamic self-adaptivity has several benefits. It can increase the reliability / availability of systems that are forced to operate in rapidly changing conditions [46], and it can also decrease the effort required for deployment as the system can be made capable of selecting the correct configuration rather than having the correct configuration pre-selected.

In our example, including dynamic adaptivity will lead to increased hardware costs. In order to accommodate real-time monitoring, it will be necessary to use a more powerful chip than what might have otherwise been used. It is also likely that a custom board will have to be fabricated rather than using commodity hardware due to using three protocols. Testing the DAS to ensure that it does disable protocols in restricted zones properly will also take additional time.

The decision of whether to include dynamic self-adaptivity into a software system or product line of software systems is also compounded by the fact that rarely does only a single design fulfill the goals of a project. Often, a myriad of candidate designs will satisfy the goals of the project with each design trading off a different set of quality attributes. It is also possible for several of the candidate designs to offer the opportunity for incorporating dynamic self-adaptivity. Deciding among the candidates, and to what extent, if any, to include dynamic self-adaptivity, requires weighing the benefits offered by each design against the effort required to realize the design understanding the trade-offs made to achieve the benefits.

Software cost, or effort, estimation techniques, which provide trade-off / risk analysis and return on investment analysis [6], provide potential methods of comparing designs. It is also common

for Decision Support Systems (DSSs), interactive computer based systems which help decision makers utilize data and models to solve unstructured problems [47], to include cost estimation to be used in situations where one or many of the decisions could have long-lasting detrimental effects on the ability of the organization to carry out its goals. To the best of our knowledge, no DSS exists for selecting between designs of a DSPL.

1.1 Problem Statement

In this dissertation, we address the lack of an existing DSS for DSPLs. As few system development projects are "greenfield," our method will also guide existing design modification. The DSS will include three main components: a method of estimating the cost / effort (which leverages architectural information) required of a DSPL design, guidance concerning common system attributes degraded by self-adaptivity, and an improved method of validating DSPL architectures.

1.2 Research Approach

We will begin by identifying the quality attributes that are commonly degraded / improved by the addition of dynamic self-adaptivity. These attributes will be identified through both the conduction of a survey of existing literature as well as an analysis comparing DASs with functionally equivalent non-adaptive systems. These attributes will be provided as part of the DSS as a suggestion of which quality attributes should be closely monitored during design / implementation.

We will then produce a method of estimating the cost, or effort, of a DSPL design. This method will leverage architectural models of the design, which increases accuracy in calculating the cost of DSPL designs. For this research, our method will use the Architecture Analysis & Design Language (AADL) exclusively as we are focused on embedded Cyber-Physical Systems (CPSs) [3, 52].

Finally, we will produce an improved method of validating DSPL architectures that accounts for the fact that many DSPL designs use inheritance. Once again, for this research, our method will use AADL exclusively.

1.3 Contributions

The primary contributions of the dissertation will be

- The introduction of an improved method of verifying a DSPL architecture modeled using AADL, especially those that include inheritance.
- The identification of quality attributes which, in general, improve / degrade when dynamic adaptivity is introduced.
- The introduction of a cost, or effort, estimation technique tailored for use with DSPLs which leverages architectural knowledge of the DSPL to produce a more accurate estimate of cost.
- The introduction of a DSS, utilizing the aforementioned contributions, that allows designers to select the best design for a given organization from a set of candidate DSPL designs.

1.4 Dissertation Statement

In this dissertation, we assert:

- That the inclusion of dynamic self-adaptivity, even when designed, implemented and tested successfully, has definite impacts on system design.
- That the identification of the aspects of a system's design which can degrade in the presence of dynamic self-adaptivity will enable designers to make better decisions concerning if dynamic self-adaptivity should be included in the system's design.
- That a DSS can be provided to allow a designer to select the best DSPL design for an organization from a set of candidate designs.

1.5 Dissertation Organization

We provide an overview of Dynamic Adaptive Systems, the Architecture Analysis & Design Language, Dynamic Software Product Lines, and the SIMPLE framework in section 2. In section 3, we provide an overview our an extension we made to the AADL AGREE annex to support AADL's inheritance features, and we provide an overview of our proposed framework in section 4 including

an introduction to our cost / effort estimation technique. In section 5, we provide an evaluation of the cost / effort estimation technique introduced as well as an evaluation of the improvements to the AGREE annex. We provide a discussion of the broader impacts of both our cost / effort estimation technique as well as the impacts of our DSS in section 6. Finally, we discuss related work in section 7, and section 8 introduces plans for future enhancements.

Chapter 2

Background

In this section, we present relevant background material for understanding the work reported in this dissertation. This section will provide an overview of Dynamic Adaptive Systems (DASs), Dynamic Software Product Lines (DSPLs), the Architecture Analysis & Design Language (AADL), relevant AADL annexes / plugins and the Structured Intuitive Model for Product Line Economics (SIMPLE) framework.

2.1 Dynamic Adaptive Systems

Dynamic Adaptive Systems (DASs) are systems which are capable of monitoring themselves and their environment, and, when the requirements or environment (context) changes, self-modifying (reconfiguring) to optimize their execution in the new context [5]. A DAS has multiple adaptation points throughout its architecture, and each adaptation point has a set of associated adaptations that can be swapped at run-time as contextual changes dictate. The adaptivity is made possible through the use of one or more MAPE (Monitor, Analyze, Plan, Execute) loops. These monitor both the system / the system's context and analyze the feeds from monitoring. After analysis, the loop plans which variation points require reconfiguration as well as the most appropriate variant to be swapped into the variation point. Finally, the loop executes the formulated plans on the system. An example architectural diagram of a simple DAS is shown in figure 2.1.

DAS offer several benefits each taking advantage of the ability to self-modify at run-time. However, the benefits offered are not without trade-offs, which will be discussed in section 4. Some of

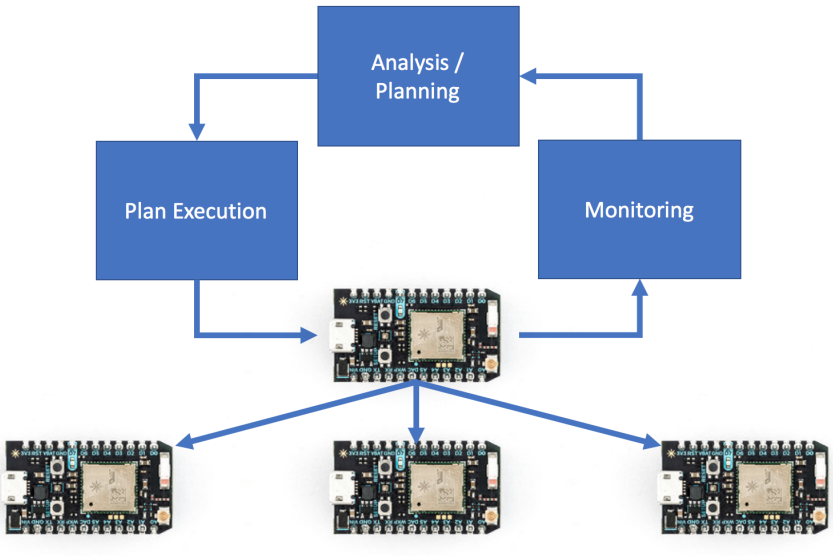

Figure 2.1: DAS Example

these benefits, depending on the dynamic adaptivity framework / design pattern used, are enhanced reliability, increased ability to meet goals and greater portability. We will discuss each of these, in turn, to answer why an architect might choose to include dynamic adaptivity into a new or existing system.

Several DAS frameworks offer the ability to self-modify in the face of faults, increasing the reliability of the system (see [16, 48, 23]). Frameworks leveraging dynamic adaptivity to increase fault tolerance or reliability fall into one of two categories: prevention and recovery. The first, prevention, extends a DAS's reconfiguration process adding a testing phase to ensure that the about-to-be-enabled module answers within tolerance under the restrictions imposed by the new context. This process could perform simple tests to check common behavior of configurations, or it could perform extensive tests that determine which features are available in the newly selected variant. The second set of frameworks, recovery, are systems that use dynamic adaptivity to recover from a fault after the fact. These systems detect the failure of a configuration and reconfigure accordingly. The reconfiguration could be guided in that, an appropriate configuration is known and can be selected, or the system could be allowed to explore the configuration space, randomly selecting configurations until it learns which configurations work best under various detected conditions.

Other frameworks allow DASs to alter their execution profile in response to either external

/ internal stimuli to better meet goals [55, 34, 44]. For example, a DAS might monitor core usage to determine if it is possible to spread a calculation on one core across other, under-utilized cores to increase efficiency. The system might also recognize the need to scale back multi-core execution to allow other, more urgent computations to execute. The idea of DASs self-modifying to increase efficiency is not limited to just multi-core or high-performance computing. It has also been explored in the domain of energy harvesting systems. Dynamic adaptivity in this domain allows the systems to tailor their cycles to the amount of energy currently available. This can increase the time which the device can remain active and the battery lifetime by using the battery only when an absolute minimum level of energy cannot be harvested from the environment.

A final reason for choosing to build a dynamic adaptive system is their increased portability. Because DASs can switch between configurations on the fly in response to detected environmental stimulus, it is possible to build a single system that can operate in a wide range of environments. However, this does not mean that a DAS can be deployed to an unknown / untested environment and operate successfully. Provided that the new environment is similar to a previously known environment, it may be possible, but it is not guaranteed.

2.2 Dynamic Software Product Lines

A Software Product Line (SPL) is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment / mission and are developed from a common set of core assets in a prescribed way [13]. SPLs have achieved remarkable benefits including productivity gains, increased agility, increased product quality and mass customization [12].

The systems, or products, derived from a SPL are the result of instantiating variation points with appropriate variants. Each product shares a common architecture, known as a family architecture, with choices of variants for a set of variation points which are the only differences among the various products. This reuse of a common core permits shortened development schedules and increased product quality.

Using figure 2.2 as an example, consider that we have a system that we plan to instantiate. Inside are two variation points, sub-components to be replaced with variants at instantiation time. Each variation point has several variants which are options; for example, the variants *A'* and *A"*

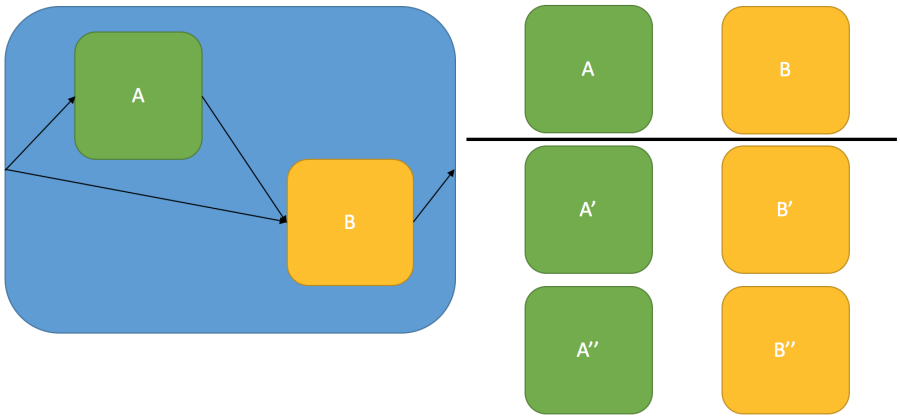

Figure 2.2: SPL Example

can be used for the variation point *A* and the variants *B'* and *B"* can be used for the variation point *B*. It is common in SPLs to place restrictions on the combination of variants that can be selected. For example, *A'* and *B'* may not be compatible with one another and instantiating a product with that combination may lead to erratic behavior. Thus a restriction stating that *A'* and *B'* can never be in the same product would be created.

In a SPL, variation points are usually instantiated before run-time, either during the design, implementation or deployment phases. It is possible, however, for a SPL to instantiate a variation point while the system is operating. SPLs that bind a variation point at run-time and allow that variation point to be changed without restarting the system are known as Dynamic Software Product Lines (DSPLs), an accepted method of modeling DASs. Variants (which could be the DAS's adaptations) and variation points (which could be the DAS's adaptation points) are also used by DPSL, where instantiation of a variation point does not necessarily lead to a new product but could lead to a new run-time configuration.

Variation points of both SPLs and DSPLs possess two primary properties: state and variant addition state [49]. We add to this list the variation point's type [17]. The state of a variant can be one of three mutually exclusive values:

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

points as the project progresses.

- **Designed** - These are variation points where the design decision has been deliberately left unanswered.
- **Bound** - A variation point that has been replaced by a variant.

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

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

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

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

[17]

2.3 Architecture Analysis & Design Language

The Architecture Analysis & Design Language (AADL) is an architecture modeling language that uses a common description language to bind together both hardware and software elements into a unified model [21]. The language is a standard of the Society of Automotive Engineers (SAE) and is actively maintained, at the time of this writing being in its second revision. AADL features a robust type system, and the language can be extended through the use of annexes, each of which can have separate syntax / functionality and each of which is standardized independently.

2.3.1 Language Overview

AADL is a description language used to represent architectures. The language has keyword constructs for representing hardware (*device*, *processor*, *bus*, *memory*), software (*process*, *thread*, *subprogram*) and the integration of the two (*system*). The language is particularly suited for a Cyber-Physical System (CPS), systems where the hardware is directly controlled and monitored via software. At its highest level, AADL allows for the intermixing of hardware and software elements within the *system* construct. It is also possible to create *abstract* components that can be refined into either hardware or software at a later time. This ability to incrementally increase specificity is beneficial in the system engineering process where the engineer knows that a specific component is needed, but he is unsure whether this component will be implemented and maintained as a hardware or software artifact. The component and its functionality can be declared and used in the architecture as an abstraction. Then, once a decision is made, it can be refined into either a hardware or software component. An example system in AADL can be seen in listings 2.1 and 2.2.

In listings 2.1 and 2.2, we have a description / representation of an ITS (Intelligent Transportation System) CPS that posts an event if the distance between the vehicle to which the system is attached and another vehicle falls below some threshold. Such a device might be used on a vehicle to determine if the vehicle is following another vehicle too closely. The system described has one output, an alert named *too_close_alert*. The system has two sub-components, a distance sensor named *sensor* and a distance evaluator named *evaluator*. The *connections* section of the system defines how the sub-components are connected. In this example, the distance value of the sensor is connected to the input of the evaluator. The system's alert value is also directly connected to the alert value of the evaluator.

AADL components are split into two separate definitions. The first is a specification of the features, inputs and outputs, that a component consumes and produces respectively. The complete set of features is referred to as a contract. An example contract is shown in listing 2.1.

```
1 system proximity_alert_system
2     features
3         too_close_alert: out event port;
4 end proximity_alert_system;
```

Listing 2.1: AADL Contract Snippet

The second definition is an implementation of the contract. Contract implementations are represented by using the *implementation* keyword as well as providing a name for the implementation which is comprised of two pieces. The first piece is the name of the contract being implemented, and the second piece is a unique identifier for the implementation. The two are concatenated using a single dot in the form *contract name.unique identifier*. The implementation of the contract listing 2.1 is shown in listing 2.2. It is possible and common for contracts to have more than one implementation. It is also common for the internal components of each implementation to vary drastically from one another.

```
1 system implementation proximity_alert_system.impl
2     subcomponents
3         sensor: device distance_sensor.impl;
4         evaluator: process distance_evaluator.impl;
5     connections
6         sensor_to_processor: port sensor.distance -> evaluator.distance;
7         processor_out: port evaluator.too_close_alert -> too_close_alert;
8 end proximity_alert_system;
```

Listing 2.2: AADL Contract Implementation Snippet

AADL's functionality is augmented through the language's provision for extensions, or annexes. An annex in AADL provides additional functionality not native to the core language. They are usually maintained by outside groups that have a vested interest in the AADL language, although some annexes are published and maintained by the language maintainers.

2.3.2 Behavior Annex

The behavior annex [54] allows users to specify how a component reacts under normal circumstances. Let's return to our example of a distance sensor. From our previous AADL snippet, the behavior of the distance evaluator is unknown. We know that it will receive an input value, *distance*. If that input value should fall below a specific threshold, an event should be fired. In listing 2.3, we show the description for this behavior along with the associated behavior specification.

```
1 process implementation distance_evaluator.impl
2     annex behavior_specification {**
3         variables
```

```
4             min_distance: int;
5         states
6             start: initial state;
7             ready: complete state;
8             calculating: state;
9         transitions
10            start -[]-> ready { min_distance := 5 };
11            ready -[on dispatch]-> calculating;
12            calculating -[distance < min_distance]-> ready { too_close_alert!
                  };
13            calculating -[distance >= min_distance]-> ready;
14    **};
15 end distance_evaluator.impl;
```

Listing 2.3: AADL Behavior Annex Snippet

The behavior annex defines a component's reactions to inputs via a finite state machine syntax. The user defines a set of internal variables which represent the data that describe the component. These variables maintain their value between transitions, and unless they are deliberately modified, they do not change their value.

The user also defines a set of states. A state defined in the state machine of the behavior annex has four possible attributes: initial, complete, final and "normal". Each state is defined to have one or more of these attributes. The initial state is analogous to the start state of a finite state machine. It is the state into which the machine is initially placed when the component is instantiated. A complete state can be thought of as a yield state; the component has completed its work for the time being and is suspended until an event or input causes it to begin execution anew. A final state is a state at which execution has completed, and the component "turns off". Normal states are simply intermediary states between any of the other three primary state types. A behavior description may contain many states with the complete and final attributes, but only one state may be the initial state. Let us consider listing 2.3 as an example. From the *start* state to the *ready* state, there is an unconditional transition, meaning as soon as the system enters the *start* state, it immediately transitions to the *ready* state. However, this transition also has a side effect indicated by the expression in curly braces to the right of the transition. In this case, the *min_distance* variable

is set to 5. From the *ready* state, there is a single conditional transition. The keywords *on dispatch* here mean on input received, and the system transitions from *ready* to *calculating* whenever input is received. The final two transitions are also conditional, but they are based on the value received from the distance sensor. If the value from the distance sensor is less than the minimum distance, the system transitions back to the *ready* state, but, as a side effect, fires the *too_close_alert* event. If the value is equal or greater, the system simply transitions back to the *ready* state with no side effects.

2.3.3 Error Annex

The error annex [19] of AADL allows the specification of how components react under abnormal conditions. It also allows specification of which types of errors are anticipated, and if those errors are handled within the component or if they are propagated to connected components. The error annex ships with a structured set of pre-defined errors types (referred to as an ontology), shown in appendix C. These existing types can be aliased or extended to create user-defined error types. The large number of error types in the ontology provides a rich set of considerations when evaluating a system for potential hazards. For example, given a system, an engineer could walk throughout the ontology asking if a specific error or error type from the ontology could be encountered by the system. Each error that could be encountered should be listed in the error annex so that the behavior of the system when encountering that error can be specified.

Let us once again consider the distance evaluator. We will assume that the sensor which sends us a value can encounter two types of errors. It could become stuck, perhaps hitting a floor or a ceiling, never being able to update its value, or, it could send us a value that is out of the range of possible or expected values, say a negative value. In the case of a negative value, the component will "fail", but it can recover as soon as the value is updated. In the case of a stuck value, the component will need to permanently fail because something has happened from which it cannot recover. If this were to occur, it is likely that the sensor will need to be cleaned or replaced for normal operations to resume. In either case, the system will be powered down and put through a hard reset, so it will fail and not provide an option for recovery. A snippet of the AADL error annex, EMV2, is shown in listing 2.4 which models the error behavior described above.

```
1 process implementation distance_evaluator.impl
2     annex EMV2 {**
```

```
3         use types ErrorLibrary;
4         use behavior EvaluatorBehavior;
5         error propagations
6             distance: in propagation {OutOfRange, StuckValue};
7             too_close_alert: out propagation {StuckValue};
8         end propagations;
9         component error behavior
10            transitions
11                normal -[distance{OutOfRange}]-> transient_failure;
12                normal -[distance{StuckValue}]-> full_failure;
13            propagations
14                full_failure -[]-> too_close_alert;
15        end component;
16    **};
17 end distance_evaluator.impl;
```

Listing 2.4: AADL Error Annex Snippet

Listing 2.4 references an error type library which defines the standard ontology including the two specific errors we mentioned previously, *StuckValue* and *OutOfRange*. The annex description also defines which errors are propagated to the component, given by the in propagation. In the distance evaluator, both of the error types that can be encountered are propagated to the component by the distance sensor. The distance evaluator can handle an *OutOfRange* error so it does not propagate it, but it cannot handle the *StuckValue* error, so it is propagated. The transitions section of the error annex is similar to the specification of the behavior annex in that it is also a finite state machine. However, this is only a partial definition of the state machine. The rest of the machine is imported by the *use behavior* statement and is shown in listing 2.5. The primary machine is defined outside of the component so that it can be reused in other components. Notice, however, that components can introduce individual transitions. In our case, the distance evaluator adds two, one for the occurrence of an *OutOfRange* error and one for the occurrence of a *StuckValue* error.

```
1 annex EMV2 {**
2     error behavior EvaluatorBehavior
3     events
```

```
4           failure: error event;
5           self_recover: recover event;
6       states
7           normal: initial state;
8           transient_failure: state;
9           full_failure: state;
10      transitions
11          normal -[failure]-> (transient_failure with 0.9, full_failure with
                0.1);
12          transient_failure -[self_recover]-> normal;
13      end behavior;
14  **};
```

Listing 2.5: AADL Error Annex Behavior Snippet

2.3.4 AGREE

Even with specifying the normal behavior and abnormal behavior of the model, it is still necessary to perform verification and validation activities. AADL facilitates these activities in multiple ways. Its strong syntax and typing have allowed for the creation of multiple simulators that can be used to confirm the architecture. Several formal verification tools have also been constructed, one of which is AGREE.

AGREE (Assume Guarantee REasoning Environment) is a compositional verification tool that can be used to confirm the behavior of a component [39]. It follows the popular assume-guarantee reasoning model, which states that provided the assumptions about a component's inputs are met the component can provide certain guarantees about its output. AGREE works by using the model to construct a state machine that is fed into a Satisfiability Modulo Theorem (SMT) prover which confirms that the state machine, given the assumptions the contract makes, can produce the values guaranteed by the contract. As it is a compositional verification tool, it can also be used to ensure that all sub-components of a component correctly contribute towards the parent component's goal. The annex utilizes the fact that AADL components are split between a contract and contract implementation. A component's assume-guarantee contracts are attached to the component contract. The component's AGREE model is then placed in the contract implementation.

It is necessary to specify the AGREE model even if the behavior annex is used as AGREE and the behavior annex are not compatible. It is also necessary to install a third-party SMT solver as one is not currently, at the time of this writing, prepackaged with the tool.

As an example, let's consider the distance sensor device. The particular sensor that we are using has a maximum range of 25 meters. Any readings beyond that distance are likely to be erroneous. Physical measurements such as distance cannot be negative. We will, therefore, specify a guarantee that our device will return a value between 0 and 25. A snippet of AGREE that matches this description is shown in listing 2.6. In this case, the error model could handle an *OutOfRange* error by ignoring it because we will have a sufficient gap at that point.

```
1  device distance_sensor
2      features
3          distance: out data port Base_Types::Integer;
4      annex agree {**
5          guarantee "the output distance is between 0 and 25": distance >= 0
               and distance <= 25;
6      **};
7  end distance_sensor;
8
9  device implementation distance_sensor.impl
10     annex agree {**
11         assert distance =
12             if distance < 0 then
13                 0
14             else
15                 if distance > 25 then
16                     25
17                 else
18                     distance;
19     **};
20 end distance_sensor.impl;
```

Listing 2.6: AADL AGREE Annex Snippet

2.4 Structured Intuitive Model for Product Line Economics

The Structured Intuitive Model for Product Line Economics (SIMPLE) is a method of predicting the cost and return on investment of developing a SPL. It is capable of modeling a range of real-world scenarios and allows product line decision makers to determine whether or not a product line is an appropriate option for development [14].

The primary idea of SIMPLE is that a company has m product lines and it wants to transition to having m' product lines. The existing product lines could be standalone products, or they could be product lines each having several products. Along the way, the company plans to add and / or delete products from its lineup. SIMPLE can be used to compare one or more product line approaches to determine which is the best fit for an organization, based on the cost required to adopt the product line vs. the benefit it brings.

SIMPLE is structured as a set of cost / benefit functions that allow a decision maker to determine how much benefit and cost savings adopting a product line approach will yield. The framework is comprised of four primary cost functions:

- $C_{org}()$ - This function returns the cost the organization incurs by adopting a product line strategy. Some example costs are reorganization, training, process improvement, etc.

- $C_{cab}()$ - This function returns the cost of developing the common base shared by all the products. An extensive amount of analysis work is usually required to determine which components are common to all, or a subset, of the products and which components are unique to individual products.

- $C_{unique}()$ - This function returns the cost of developing unique components of a single product. This might result in a complete product, or it could result in an intermediate product (or a common sub-base) shared by a subset of products in the product line.

- $C_{reuse}()$ - This function returns the cost of building a product using the core assets. This includes checking out the shared models, configuration, insertion of unique components and testing.

[14] These four primary functions can also be paired with two other functions which are used to model the cost of evolving the product line.

- $C_{cabu}()$ - This function returns the cost of updating the base of the product line. These changes could introduce new variation points, or it could be bug fixes to correct discovered issues.
- $C_{uniqueu}()$ - This function returns the cost of updating a single product. This function accounts for the fact that not all products will require updating, and represents only the cost of updating the unique components of the product.

[14]

Combining the above functions, we can create a "super" function which provides a definition of the cost of constructing a product line. The equation $C_{prod}()$ is written

$$C_{prod}(t) = C_{org}(t) + C_{cab}(t) + \sum_{i=1}^{n} (C_{unique}(product_i, t) + C_{reuse}(product_i, t)) \tag{2.1}$$

.

Chapter 3

Dynamic Software Product Line Model Validation

As stated in section 1.3, the first goal of this research is

- The introduction of an improved method of verifying a DSPL architecture modeled using AADL, especially those that include inheritance.

The traditional version of AGREE, AADL's verification / validation tool, does not support AADL's inheritance features [1], thus, as part of the work for this dissertation, we have created an extended version of AGREE, XAGREE, with additional behavior and keywords that support AADL's inheritance features. This allows models of DSPLs to be verified compositionally reusing verification assets to a greater extent reducing duplication and complexity.

We now cover the modified / additional statements added to the XAGREE language to facilitate inheritance. A short overview of the statements that have been added or modified is presented in table 3.1. Each statement will be discussed and an example of its use provided.

3.1 Assume / Guarantee Statements

The *assume* and *guarantee* statements of the AGREE language are analogous to the pre-condition / post-condition concepts of other verification tools. With traditional AGREE, the assumptions and guarantees of parent components are not inherited by their children even though

Table 3.1: XAGREE Syntax Overview

Keyword	**Description**
assume	declare that the system expects input to conform to the following statement
do not inherit	explicitly disable inheritance
eq	declare a concrete variable or override an abstract variable
eq abstract	declare an abstract variable
guarantee	declare that output of the system will conform to the following statement
inherit	explicitly state that inheritance from a parent occurs

children will often use the same inputs, outputs, assumptions and guarantees as their parents. eXtended AGREE (XAGREE) allows for such inheritance. We also recognize that it is sometimes necessary to tweak the assumptions or guarantees of your parent, particularly if the child introduces new inputs or outputs that the parent does not have. This overriding of parent assume / guarantees is also possible with XAGREE.

```
1  system parent
2      features
3          ip: in data port Base_Types::Integer;
4          op: out data port Base_Types::Integer;
5      annex xagree {**
6          assume "input req": ip >= 0;
7          guarantee "output req": op >= 1;
8      **};
9  end parent;
10
11 system implementation parent.impl
12     annex xagree {**
13         assert op = ip + 1;
14     **};
15 end parent.impl;
```

Listing 3.1: AADL Child Guarantee / Assume Example

```
1  system child extends parent
2      features
3          --ip: in data port Base_Types::Integer;
4          --op: out data port Base_Types::Integer;
```

```
5          op2: out data port Base_Types::Integer;
6      annex xagree {**
7          --assume "input req": ip >= 0;
8          guarantee "output req": op >= 1 and op2 >= 1;
9      **};
10 end child;
11
12 system implementation child.impl extends parent.impl
13    annex xagree {**
14        --assert op = ip + 1;
15        assert op2 = ip + 1;
16    **};
17 end child.impl;
```

Listing 3.2: AADL Child Guarantee / Assume Example

An example of XAGREE's assume and guarantee statements are shown in figures 3.1 and 3.2. Also shown in these figures is a demonstration of how XAGREE permits verification assets to be reused across different components of the model hierarchy as well as how assumptions and guarantees can be overridden by children if necessary. The parent component, introduced in figure 3.1 has two features, a single input and output, and the XAGREE annex assumes that the input will be greater than or equal to 0 while guaranteeing that the output will be greater than or equal to 1. The behavior of the parent is simply to take the input value and set the output to the input plus 1. The child, shown in figure 3.2, adds a second output. Note that in figure 3.2 all inherited statements are shown using comments (denoted by a double dash in AGREE / XAGREE and AADL). The guarantees of the child have to be modified or amended to account for this extra output. The override is driven by the descriptor string, or, children who have an assumption or guarantee with a descriptor that matches a parent assumption / guarantee's descriptor will override the parent's matching descriptor.

3.2 Eq / Eq Abstract Statements

In traditional AGREE, the *eq* statement allows for the declaration of a single variable. In introducing inheritance, we modified the *eq* statement to either introduce a new variable or to

override an existing variable if the variable in the child has the same name and type, or a related polymorphic subtype [20] of the parent's type, as a variable in the parent. We also introduced an *eq abstract* statement that provides a way to define a variable without providing an implementation for that variable. Abstract variables in XAGREE are much like abstract variables in Java or C++. They can be used in calculations and statements just like any other variable but their implementation is left for children, or extenders, to provide. We also introduce the concept of an abstract implementation, a contract implementation that contains an XAGREE annex which introduces or inherits an abstract variable. For an implementation specification to be non-abstract, or concrete, it must override and provide an implementation for all inherited abstract variables without introducing any new abstract variables.

```
1  system parent
2     features
3        type: out data port Base_Types::String;
4     annex xagree {**
5        guarantee "output req": type = null;
6     **};
7  end parent;
8
9  system implementation parent.impl
10    annex xagree {**
11       eq abstract myType: string;
12       assert type = myType;
13    **};
14 end parent.impl;
```

Listing 3.3: AADL Eq Example

```
1  system child extends parent
2     --features
3        --type: out data port Base_Types::String;
4     annex xagree {**
5        guarantee "output req": type = "child";
6     **};
7  end child;
```

```
8
9  system implementation child.impl extends parent.impl
10     annex xagree {**
11         eq myType: string = "child";
12         --assert type = myType;
13     **};
14 end child.impl;
```

Listing 3.4: AADL Child Eq Example

An example of eq / eq abstract statements and how they are used in inheritance is shown in figures 3.3 and 3.4. Once again comment lines (those starting with a double dash) represent statements that have been inherited. The parent figure, shown in figure 3.3, has one output, a string representing the type. In the parent figure, the type produced by the component is guaranteed to be null. This is reflected in the parent's implementation as *myType* has been declared abstract and not provided with an implementation. The child figure, shown in figure 3.4, overrides the parent's guarantee and asserts that the component will declare its type as "child". The child, however, does not have a full implementation, only a provision of a definition for the inherited abstract variable. The assert that ties the abstract variable to the output is inherited and does not require re-specification.

3.3 Inherit / Do Not Inherit Statements

The *inherit* and *do not inherit* statements are unique to XAGREE. The *do not inherit* statement allows inheritance to be explicitly disabled reverting to the behavior of the AGREE. This statement was introduced to provide a means of backward compatibility. The *inherit* statement is similar to the *do not inherit* statement in that it allows a developer to state explicitly that inheritance does occur.

```
1 system parent
2     features
3         ip: in data port Base_Types::Integer;
4         op: out data port Base_Types::Integer;
5     annex xagree {**
```

```
6           assume "input req": ip >= 0;
7           guarantee "output req": op >= 1;
8       **};
9   end parent;
10
11  system implementation parent.impl
12      annex xagree {**
13          assert op = ip + 1;
14      **};
15  end parent.impl;
```

Listing 3.5: AADL Inherit Example

```
1   system child extends parent
2       features
3           --ip: in data port Base_Types::Integer;
4           --op: out data port Base_Types::Integer;
5           op2: out data port Base_Types::Integer;
6       annex xagree {**
7           do not inherit;
8           assume "input req": ip < 1;
9           guarantee "output req": op <= 0 and op2 <= 0;
10      **};
11  end child;
12
13  system implementation child.impl extends parent.impl
14     annex xagree {**
15         do not inherit;
16         assert op = ip - 1;
17         assert op2 = ip - 1;
18     **};
19  end child.impl;
```

Listing 3.6: AADL Child Inherit Example

Finally, we provide an example where inheritance is controlled using the *inherit* and *do not inherit* statements. This example is shown in figures 3.5 and 3.6 and can be seen in the implementation's XAGREE annexes. Note that the child annex does not inherit the assert of the parent due to the child specifying that inheritance should not be used. Note, however, that the *do not inherit* statement does not affect extends. The child will still inherit the features of the parent even though the AGREE annex will not inherit any attributes of the parent.

Chapter 4

Dynamic Software Product Line Model Comparison

As stated in section 1.3, the remaining goals of this research are

- The identification of quality attributes which, in general, improve / degrade when dynamic adaptivity is introduced.
- The introduction of a cost, or effort, estimation technique tailored for use with DSPLs which leverages architectural knowledge of the DSPL to produce a more accurate estimate of cost.
- The introduction of a DSS, utilizing the aforementioned contributions, that allows designers to select the best design for a given organization from a set of candidate DSPL designs.

Each of these goals are discussed, in turn, throughout the remainder of this section. In section 4.1, we discuss the effects of dynamic adaptivity on quality attributes. In section 4.2, we introduce a cost / effort estimation framework tailored for DSPL models. Finally, in section 4.3, we introduce a DSS framework for comparing designs of a DSPL model.

4.1 Dynamic Adaptivity Effects on System Quality Attributes

The benefits offered by DASs are not free, and several trade-offs must be taken into account when deciding whether or not to incorporate dynamic adaptivity. However, not all trade-offs occur

in every situation. DASs, like many other software engineering decisions, exist along a continuum of possible choices. In some situations, incorporating dynamic adaptivity changes only how the quality attributes are measured. In other situations, dynamic adaptivity affects the quality attribute itself. However, this depends on why and how dynamic adaptivity was incorporated.

An example of dynamic adaptivity changing how a quality attribute is measured would be dynamic adaptivity's affect on fault recoverability. Traditionally, recoverability focuses on the component's ability to handle a particular fault. For a dynamic adaptive system, the design question is whether a component can switch to another configuration which handles that particular fault.

An example of dynamic adaptivity changing a quality attribute would be dynamic adaptivity's affect on testability. Dynamic adaptive systems have extra run-time configurations that must be covered as well as an adaptation mechanism that must be tested to ensure that it works as expected. The method of measuring testability, based on the number of system states, remains unchanged, but dynamic adaptivity adds additional states increasing the effort required to satisfy the same level of test coverage.

To determine which quality attributes are affected, we conducted a systematic literature review of existing DAS / DSPL literature. The survey revealed several common quality attributes which receive general improvement with the addition of dynamic adaptivity, and some literature results listed the trade-offs made to achieve the improvements in other attributes. However, a number of quality attributes were not listed in the literature, and no information was provided to indicate how dynamic adaptivity affected them. To determine the effects on these attributes, we took the architectures of several systems and introduced dynamic adaptivity into the architecture, analyzing the systems before and after to determine how dynamic adaptivity affected the system.

We now provide some guidelines, from the DAS literature and an analysis of system architectures, for attributes that can be improved or degraded. The existing body of DAS literature also provides some mitigation tactics that can be used to offset or lessen the effects of dynamic adaptivity on some quality attributes. For purposes of discussion, we limit ourselves to the quality attributes given in ISO (International Standards Organization) 25010 [2], also listed in appendix B.

4.1.1 Functionality

With the addition of dynamic adaptivity, changes to characteristics related to functionality are, in general, mixed. Suitability changes seem to be largely dependent on whether or not extra

processing power is available for performing the computational work for dynamic adaptivity. If extra processing power is available, then suitability will largely remain unchanged. However, if extra processing power or extra hardware components are needed, then suitability can be negatively affected [37, 36]. Accuracy can either improve or degrade. If the dynamic adaptivity permits the system to adhere more completely to its goals by avoiding failure, then accuracy is likely to increase. However, if the system is forced to spend long periods of time searching for the best configuration or dynamic adaptivity allows the system to respond in an incorrect manner for the context, then accuracy will be negatively affected. Compliance, likewise, can either improve or degrade depending on the function dynamic adaptivity serves. If dynamic adaptivity allows for fault recovery, then compliance is likely to improve, however, for other purposes, how compliance is affected will depend on whether or not the dynamic adaptivity allows for configurations that are not compatible with the context. Security has been shown, in general, to improve with the addition of dynamic adaptivity.

4.1.2 Reliability

As a sizeable portion of dynamic adaptivity literature focuses on fault recovery and avoidance, reliability appears to be one of the primary motives for introducing dynamic adaptivity (see [16, 48, 23]). Therefore, it should come as no surprise that, in general, dynamic adaptivity improves the characteristics related to reliability. However, there are cases where reliability can suffer.

Maturity can be improved by dynamic adaptivity due to the increased fault tolerance and recoverability. Fault tolerance and recoverability both assume that there exists a reconfiguration which handles the error or context in which the software currently exists. Provided that uncertainty and requirements have been appropriately managed, there is little reason that this should not be the case. However, some of the approaches used to enhance fault tolerance involve run-time testing of about-to-be-deployed configurations to ensure that they will not fail. Other approaches involve solution space searches to find a configuration that will not fail in the current context. There exist several domains where the offline time necessary for these tests and searches does not exist unless extra processing capabilities are added. Thus, these techniques are not suitable for each such domain, such as real-time safety-critical CPS [36].

4.1.3 Usability

How characteristics related to usability are affected depends on whether or not dynamic adaptivity changes the interfaces used to interact with other software components. If this is the case, then both learnability and operability will both degrade, however, if this is not the case, then both sub-characteristics will remain unchanged.

Regardless of whether or not the interface is modified as part of dynamic adaptivity, the understandability of the software will almost always be negatively affected. Dynamic adaptivity is complex, and explaining to end users how the software arrives at a decision without human intervention through the monitoring of sensors and stimulus is, in our experience, difficult and easily misunderstood.

4.1.4 Efficiency

Characteristics related to efficiency are affected with mixed results when dynamic adaptivity is introduced. Dynamic adaptivity can allow the system to better utilize existing resources [55, 34]. This is especially true for dynamic adaptive software that can consolidate computation onto a minimal number of cores saving energy. This is also true for systems which use dynamic adaptivity to get better total lifetime out of batteries, like energy harvesting systems [44]. Time behavior can either improve or degrade. For systems that explore solution spaces or require pre-reconfiguration testing, the time behavior can be adversely affected, especially if reconfigurations occur frequently. However, time behavior can be improved if the dynamic adaptivity allows computation to continue when it would have otherwise not been able to do so.

4.1.5 Maintainability

Characteristics related to maintainability are, in general, negatively affected by the incorporation of dynamic adaptivity; however, some sub-characteristics do see general improvement. Analyzability of the software is negatively affected by dynamic adaptivity. This is due, in part, to the fact that much of the behavior of the system is not static and occurs largely at run-time. Changeability, provided that the software remains modular, can be improved by dynamic adaptivity. This is especially true if the change requires simply adding a configuration. Stability of software usually remains unaffected by dynamic adaptivity.

Testability of software is negatively affected by dynamic adaptivity [36], but it is also the area where the greatest amount of work has been completed to offset the trade-off (see [53, 40]). Many of these approaches rely on formal methods or run-time testing. As mentioned previously, run-time testing is not applicable to every domain, and more work on methods for fully modeling a dynamic system's context using formal methods are needed. However, most of the work in this field has shown promise at improving the testability of software systems including dynamic adaptivity despite the compounded number of states adaptivity introduces.

4.1.6 Portability

With the addition of dynamic adaptivity, portability sub-characteristics are generally improved or remain unchanged. Adaptability is usually improved as it is now possible for the software to respond to a greater number of contextual situations without additional changes. It still may not be possible for the software to respond correctly to unknown or unforeseen contexts, but steps can be taken to manage this uncertainty. Installability is usually enhanced; it is now usually possible to deploy the software to a new installation with little to no additional configuration. Conformance will, in general, remain unchanged. However, it is possible for it to degrade if it is possible for the software to choose the wrong configuration for a context or if the software can adversely affect the environment. Finally, replaceability will usually degrade. The software is required to manage not only a large number of configurations, but it is also responsible for switching between those configurations as the need arises. Any new piece of software that replaces the system will have to meet those requirements on the first deployment or find a way to mitigate them. However, adding additional features to a DAS can be easier, particularly if the addition requires only a new variant.

4.1.7 Summary

DAS are adopted to achieve specific gains with respect to a focused set of quality attributes, however, these gains come at the cost of other attributes which degrade. Care must be taken to ensure that, even though the gains desired were achieved, unacceptable losses were not incurred elsewhere. It must also be noted that not every quality attribute improves or degrades consistently in the presence of dynamic adaptivity; the scenario and reason for incorporating dynamic adaptivity both play a role.

4.2 Cost / Effort Estimation for Dynamic Software Product Lines

As DSPLs can be modeled with SPL techniques, it follows that it should also be possible to estimate the instantiation cost / effort in a fashion similar to SPLs. SIMPLE is one such cost estimation technique that provides a method for comparing SPL designs.

However, directly applying SIMPLE to a DSPL architecture yields sub-optimal results since the reasons for adopting a SPL strategy differ from the reason for adopting a DSPL strategy. SPLs are selected because they allow an organization to produce better products more quickly and with less cost. This is accomplished by related products sharing a common base that is populated with variation points that each product instantiates. The variation points in a product are traditionally closed, bound and static, that is, once the product has been instantiated, it is no longer possible to change the variation point binding. Some product lines delay the instantiation of a product until run-time, but the result is still the same, once bound, the variation point cannot be rebound.

DSPL systems are adopted to account for run-time uncertainty, which takes many forms. For example, the environment in which the system will be deployed may not be completely determined. It might also be possible that the system will require complex tradeoffs due to the environment that will require it to evolve on the fly. Also, unlike traditional SPL architectures, the variation points of a DSPL are typically fluid. The system is free, at any time, to unbind and rebind another variant to the variation point. Due to this fluidity, DSPL architectures exact heavier trade-offs.

In addition to DSPL being adopted for different reasons than SPL, several differences exist between the costs incurred by DSPL adoption and the costs incurred by SPL adoption. First, when building a DSPL, the company is no longer building a set of products, rather the company is either building a single product which can operate in many configurations or a product line of products of which some can operate in many configurations. The ramifications of the difference touch many aspects of the software development life cycle. In addition to requirements which govern the family architecture shared by all configurations and the requirements that govern individual variants, requirements are also needed to govern the adaptation process. The cost of these extra requirements must be accounted for. Design costs could also be increased due to the possible addition of components to control adaptivity.

Second, each variation point within a DSPL can incur additional costs atypical of a variation

point in a traditional SPL. DSPL variation points typically have the ability to change on the fly without human intervention while SPL variation points typically do not. This ability can require extra hardware, and the decision planning and execution process must be supplied with all data needed in order to make a decision. This could compound the number of connections a variation point component needs increasing the cost of instantiation.

Following the above reasoning, more accurate effort / cost estimation results can be obtained if adaptive and non-adaptive components of the family architecture are considered separately. Splitting components allows the cost of each variation point to be tailored specifically to the adaptivity style being considered while enabling reuse of the effort / cost estimation for the non-adaptive components.

We now introduce the Structured Intuitive Model for Dynamic System Economics (SIMDASE). SIMDASE is a model-based estimation technique and, as such, requires an architectural model from which it can obtain additional information to produce more accurate estimates. SIMDASE, unlike SIMPLE, is therefore provided as an annotation for architecture description languages. At the time of this writing, tooling for SIMDASE exists only for AADL and its development environment OSATE. SIMDASE for AADL can be installed from the plugin update site, https://bitbucket.org/strategicsoftwareengineering/simdase-osate. Support for other architectural modeling languages is planned as part of future efforts, see section 8.

4.2.1 Overview

SIMDASE is based on SIMPLE, and shares some of SIMPLE's design philosophy as well as some of SIMPLE's core functions. However, in order to leverage the recursive nature of an architectural model, SIMDASE is split across 5 equations. Each is listed below, along with the definitions of the sub-functions of each.

$$C_{pl}(c) = C_{setup}() + C_{org}() + C_{comp}(c, 0) \tag{4.1}$$

C_{pl} represents the cost of constructing a product line. The parameter c is the top-most component of the architectural model / sub-model to be evaluated. C_{setup} represents the cost of developing the architectural model for evaluation within SIMDASE. C_{org} represents the cost the organization incurs by adopting a DSPL strategy. Some example organizational costs are reorgani-

zation / training / process improvement needed by the organization before a DSPL design can be adopted.

$$C_{comp}(c,i) \quad = \quad (C_{static}(c))^{s(i)} \quad + \quad \sum^{n(c)}(C_{nvp}(n,i \quad + \quad 1)) \quad + \quad \sum^{a(c)}(C_{avp}(a,i \quad + \quad 1))) \quad (4.2)$$

C_{comp} represents the cost of building a single component. The parameter c is the component being evaluated, and i is the depth at which c occurs in the architecture's hierarchy. C_{static} represents the cost of c's sub-components that do not contain variation points. $s(i)$ is the user-definable scaling factor. The scaling factors allow costs to be scaled based on depth in the architecture hierarchy as well as architectural style. For example, if you have a component that is reused across multiple architectures, but its cost(s) vary depending on its depth within the architecture, you can provide a scaling factor that will allow you account for the difference so that the component does not have to be forked and modified for each architecture. $n(c)$ returns all of the non-dynamic variation points within c, and $a(c)$ will return all of the dynamic variation points within c.

$$C_{nvp}(n,i) = \sum^{v(n)}(C_{var}(v,i)) \quad (4.3)$$

C_{nvp} represents the cost of building a non-dynamic variation point. The parameter n is the non-dynamic variation point under consideration, and i is the level at which n occurs in the architecture's hierarchy. The function $v(n)$ will return all of the variants for the non-dynamic variation point, n.

$$C_{avp}(a,i) = C_{adapt}(a)^{s(i)} + \sum^{v(a)}(C_{var}(v,i)) \quad (4.4)$$

C_{avp} represents the cost of building an adaptive variation point. The parameter a is the dynamic variation point under consideration, and i is the level at which a occurs in the architecture's hierarchy. C_{adapt} is the cost of the adaptation mechanisms for a. Note that this cost will depend heavily on which adaptation mechanism is chosen. $s(i)$ is the user-definable scaling factor. The function $v(a)$ will return all of the variants for the dynamic variation point, a.

$$C_{var}(v,i) \;=\; (C_{static}(v))^{s(i)} + C_{reuse}(v)^{s(i)} + \sum^{n(v)}(C_{nvp}(n,i+1)) + \sum^{a(v)}(C_{avp}(a,i+1))) \quad (4.5)$$

C_{var} represents the cost of building a variant. The parameter v is the variant under consideration, and and i is the level at which v occurs in the architecture's hierarchy. C_{static} represents the cost of v's components that do not contain variation points, somewhat equivalent to SIMPLE's C_{unique}. C_{reuse} represents the cost of building a variant by reusing the core assets, and $s(i)$ is the user-definable scaling factor. Finally, $n(v)$ will return all of the non-dynamic variation points within v, and $a(v)$ will return all of the dynamic variation points within v.

4.2.2 Use

SIMPLE defines 29 categories among which the development costs of a project can be divided (listed below) and SIMPLE relies on managerial knowledge of a company and teams to produce an estimate for each of these categories for a given function definition. This does not mean that the values are strictly integers; the values for these subareas could be function definitions, such as a sinusoidal function, if the user wishes.

- Software Engineering
  - Architecture Definition
  - Architecture Evaluation
  - Component Development
  - Mining Existing Assets
  - Requirements Engineering
  - Software System Integration
  - Testing
  - Understanding Relevant Domains
  - Using Externally Available Software
- Technical Management

  - Configuration Management
  - Make/Buy/Mine/Commission Analysis
  - Measurement and Tracking
  - Process Discipline
  - Scoping
  - Technical Planning
  - Technical Risk Management
  - Tool Support
- Organizational Management
  - Building a Business Case
  - Customer Interface Management
  - Developing an Acquisition Strategy
  - Funding
  - Launching and Institutionalizing
  - Market Analysis
  - Operations
  - Organizational Planning
  - Organizational Risk Management
  - Structuring the Organization
  - Technology Forecasting
  - Training

SIMDASE, derived from SIMPLE, uses these 29 categories as well, but it also adds the ability to define custom categories. These categories form the basis for the C_{setup}, C_{org}, C_{setup}, C_{static}, C_{adapt} and C_{reuse} functions. Managerial knowledge is also leveraged to produce a definition for the scaling factor function, $s(i)$, as not all projects will have a scaling factor and factors can differ between projects.

SIMDASE also uses a different representation of cost. Where as SIMPLE represents each category's cost as either an integer or a function returning an integer, SIMDASE represents cost as the number of employee's required to complete the tasks in a given category as well as how much time is required of each employee. However, SIMDASE does also allow fixed costs which cannot be assigned to a single employee or group of employees to also be returned.

An example of such a situation might be that a given task will require 5 developers. In addition to developer time, several software purchases will have to be made so that the developers can complete their assigned tasks. The fixed cost of the software will be returned along with the employee costs by the function to account for the fixed costs that are not assigned to a given developer.

The reason for choosing to make the cost estimation employee based rather than using solely numeric functions is that, in our experience, managers often think of costs in terms of how many employees will be assigned to a given task rather than in fixed numbers. Having SIMDASE represent cost as the number of employees reduces the cognitive load required to use the tool. In addition, raw numeric costs can be obtained by taking the total number of hours required for each employee and multiplying it by their estimated hourly salary.

4.2.3 Annex Example

To calculate an estimation of a project's cost, a user must first provide an architectural definition of the system. Note that this definition does not need to be complete, but it must contain enough information to loosely outline the system, its variation points, variation point types and the variants. Afterwards, a list of the various components needed for the system can be forwarded to a manager so that cost information can be supplied. The information provided can then be annotated into the model through the use of the SIMDASE annex.

After defining and annotating architecture, the SIMDASE evaluation engine can determine which specified entities are variation points as well as the variation point's type. For each variation point, the variants for the variation point must be specified. Also, for dynamic variation points, the costs of adaptivity must be specified. An example SIMDASE annex is given in listing 4.1.

In listing 4.1, an example SIMDASE annex is given in which component development will occur for 31 time periods. During the first time period, several employees will be used to plan development and assess risk on the project. 2 Business Leads will be used for 20 hours to do

organizational planning and 20 hours to do organizational risk management. Note that the variable t is available in all formulas within the annex and represents the current time period. Another variable i, which represents the annex's containing component's level within the architecture hierarchy, is also injected into all formulas.

```
annex simdase {**
 -- the systems construction will occur over 31 spaced time periods
 time_min => 0.0;
 time_max => 30.0;
 cost => {
  organizational_planning => {
   2.0 "Business Lead" for 30.0
  };
  scoping => {
   2.0 "Project Manager" for 40.0
  };
 };
**};
```

Listing 4.1: AADL Contract Snippet

The output from SIMDASE is an aggregated list of components in the architecture and the employees required to build them. Note that top level components in the architecture aggregate the costs of their sub-components as well as their own costs in the report. Reports also contain which components are under active development during a time period.

4.3 Decision Support System

As of the writing of this dissertation, the full DSS is not yet fully completed and there is more work to be completed as future work (discussed in section 8). The DSS's outline is shown in figure 4.1. The parts of the framework shown in green are mostly completed and have been discussed in previous sections (Cost Estimation [SIMDASE] in section 4.2, Architecture Validation & Verification [XAGREE] in section 3). We briefly discuss Quality Attribute Estimation in section 4.1, but work in this area is still ongoing. We now introduce how these various parts will ultimately fit

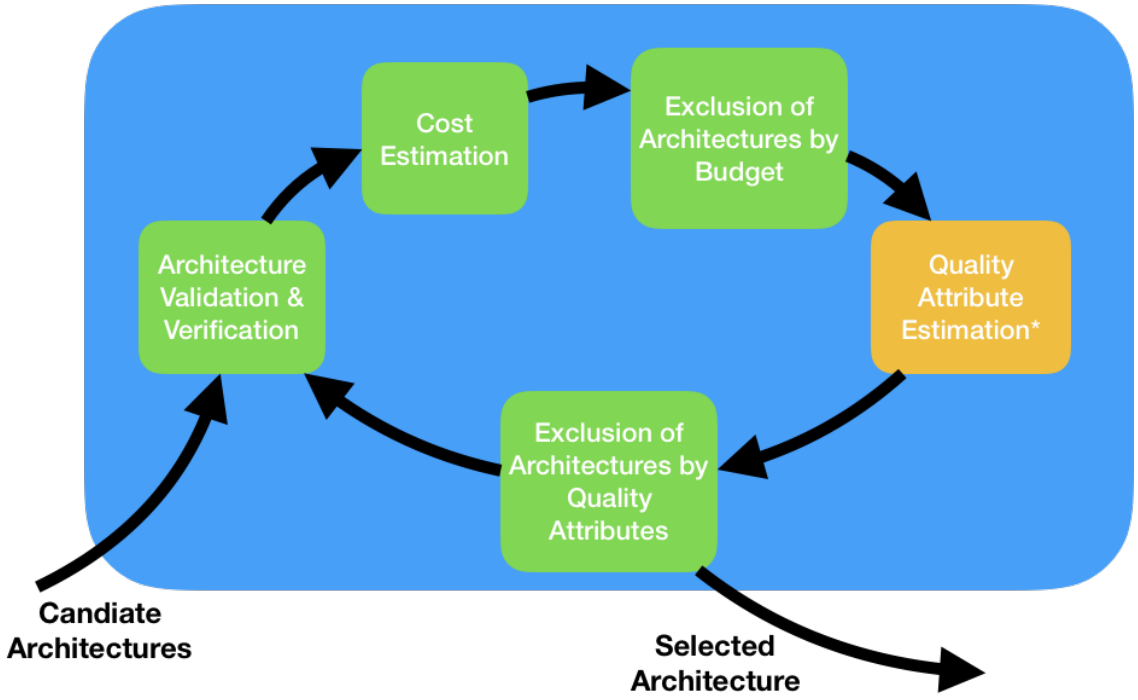

Figure 4.1: Decision Support System Outline

together to provide a complete DSS.

Unfortunately, comparison of DPSL designs on the basis of cost solely might lead to sub-optimal results. There are two primary reasons for this. First, the cost of testing a DSPL differs from the cost of testing of a traditional SPL. In a traditional SPL, each product can be individually tested as it is to be released. It is not necessary to test the entire line of products all at once. Since a DSPL can be a single product with many run-time configurations or a product line of products some with many run-time configurations, the amount of effort required to test a deployable project is usually greater since more testing must be done up front rather than in gradual increments.

Second, DSPLs make more significant trade-offs than traditional SPLs, and the measurement and accounting for each trade-off incurs additional cost. These trade-offs can drastically affect deployments of the system. Dynamic adaptivity is chosen for specific reasons to improve specific quality attributes, however, thought is not always given to the quality attributes that are traded-off. The trade-offs can make the system harder to maintain, more difficult to deploy, less secure or more prone to failure. Each trade-off incurs a cost to either mitigate the cost or to ensure that the trade-off is acceptable.

We now introduce our DSS for selecting between designs of a DSPL. The first phase requires that an architect select which verified, candidate designs they wish to compare against one another. The cost / effort to instantiate each design is then calculated so that any designs that are out of budget can be eliminated. The second phase involves analyzing the detailed architectural designs of candidates selected from the results of the first phase to estimate the design's quality attributes. The DSS we present is tool agnostic. Any framework, description language, cost estimation framework, validation tool, etc. can be used. Naturally, we recommend the tools created as part of this research, but we do not require them.

4.3.1 Phase 1

The first phase of our DSS involves the creation of several validated and verified DPSL designs for a given to-be-constructed system. Each design's high-level goals and requirements should be inline with the requirements of the to-be-constructed-system.

The cost / effort to construct each design will be calculated and a set of designs that meet the budget criteria established by the organization will be produced. Note that it is generally advised to set boundaries for the budget criteria. For example, perhaps the best design is 105% of the budget. In some organizations, spending the extra 5% to achieve the most optimal design is acceptable whereas in some other organizations it is not.

There are multiple cost / effort estimations frameworks that are available to be used in this step. The estimation framework presented in this research, SIMDASE, is an option as is SIMPLE. Other estimation frameworks related to these two frameworks, discussed in section 7, are also viable options.

4.3.2 Phase 2

As comparison on cost / effort alone is not enough, the second phase of our DSS involves the selection of one or more candidates from phase 1 whose efforts / cost are within the budgeted range, or within acceptable deviation of the budget, for the organization. The architectural models of the designs accepted in phase 1 will be used to estimate the current compliance of each design's quality attributes.

Two sets of quality attributes will be selected by the system architect. The first set of ISO

25010 quality attributes will be the set of attributes the architect seeks to improve by incorporating dynamic adaptivity. For example, the architect might choose to include dynamic adaptivity so that reliability is improved, specifically fault tolerance and recoverability.

The second set of quality attributes is the set for which a trade-off is acceptable to the system architect. This will leave a third, implied set of attributes which are not in the union of the two identified sets. These are attributes which the architect is not necessarily seeking to improve but which the architect is unsure if degradation is acceptable.

Once an architectural model is constructed, estimations of the first and third set of quality attributes should be taken. These estimations provide additional insight into the appropriateness of each candidate design for the organization. It can also alert the system architect to a design's need for reconsideration or modification to bring attributes back into compliance.

Note that it is not sometimes necessary to include the second set of attributes in the estimation process as well. This is true if any of the quality attributes of the second set have minimum thresholds. If this is true, those attributes having minimum thresholds must be measured in addition to the first and third set of attributes to ensure compliance.

4.3.3 Decision

Once the cost and quality attribute compliance of each design has been calculated, in general, the design with the lowest cost and highest compliance will be the best choice. However, this choice will ultimately be an organizational decision.

Occasionally, designs will be heavily modified during the second phase to bring the design into compliance. If this occurs, the DSS can be used in an iterative manner where the updated design has its cost / effort recalculated followed by an additional estimation of quality attribute compliance. This iteration can be done as many times as deemed necessary for as many of the candidate designs for which iteration is needed.

Chapter 5

Analysis & Evaluation

In this section, we evaluate the completed portions of our DSS for comparing designs of DSPLs. In this research, we focus exclusively on evaluating the cost / effort estimation framework, SIMDASE, and XAGREE, the improved verification / validation tool for DSPL architectures. The remaining portions of the DSS will be evaluated as part of future work (discussed in section 8.

We make two claims concerning SIMDASE.

- **Claim C1** SIMDASE is sufficiently flexible to support the estimating of cost / effort for a wide range of development strategies.
- **Claim C2** SIMDASE's effort estimation formula is robust with regard to input parameters, including user-defined input parameters that could return periodic or sinusoidal values.

We also make one claim concerning XAGREE.

- **Claim C3** XAGREE, in general, reduces the size of verification assets, reduces the complexity of verification assets, and reduces the amount of duplication across verification assets when compared to traditional AGREE.

Each claim is discussed individually in the remaining sections. Throughout the evaluation, we will use 3 primary examples. The first example is a forklift tracking system that originates from a project encountered during our research. The two remaining examples are testing exemplars proposed and accepted by the DSPL community as part of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). The first is a dynamic adaptive

Table 5.1: Evaluation Examples Overview

Example (Scenario)	Case Study Analysis	Sensitivity Analysis	Complexity Reduction Analysis
Tracking System	5.1.2		5.3.2
Tracking System (Waterfall)	5.1.2.1		
Tracking System (Agile)	5.1.2.2		
Tracking System (Undetermined)	5.1.2.3	5.2.2	
DAS - Nutrition	5.1.3		5.3.3
DAS - Nutrition (Waterfall)	5.1.3.1		
DAS - Nutrition (Components Same Periods)	5.1.3.2		
DAS - Nutrition (Purchased Component)	5.1.3.3		
DAS - Nutrition (Undetermined)	5.1.3.4	5.2.3	
DAS - Underwater	5.1.4		5.3.4
DAS - Underwater (Undetermined)	5.1.4.1	5.2.4	

nutrition system [4] and the second is a dynamic adaptive underwater vehicle [24]. Full models for each example along with additional examples can be found on the project's homepage: https://bitbucket.org/account/user/strategicsoftwareengineering/projects/SIMDASE. For each of our selected examples, we will present multiple scenarios, some of which will be reused across evaluations. The table below lists each scenario as well as which evaluations the scenario is associated with and in which section the evaluation using that particular scenario occurs. In addition to providing full models, the majority of the data plots in the evaluation are available as manipulable data plots. These plots can be accessed by navigating to the link associated with each graph. The full data set for each graph is also available via this link.

5.1 Demonstrating Cost / Effort Estimation Model Flexibility

Claim C1: SIMDASE is sufficiently flexible to support the estimating of cost / effort for a wide range of development strategies.

5.1.1 Evaluation Plan

To demonstrate claim C1, we will evaluate each of our selected examples. For each system, we will first introduce the system and its architecture. We will then produce an estimate of the cost of implementation for that system under various scenarios (3 scenarios for the first system, 4 scenarios for the second system and 1 scenario the last system). In both the introduction and scenario analysis, we will provide snippets of the AADL architecture as they relate to the discussion. Full models can be found online at
https://bitbucket.org/strategicsoftwareengineering/simdase-examples-2. With each scenario, we show that our model is capable of supporting the unique complexities introduced by that scenario.

5.1.2 Forklift Tracking System

The first system we introduce is a forklift tracking system for an industrial manufacturing facility. The goals of this system are to monitor the location and speed of forklifts as they make their rounds through the facility on a daily basis to determine if 1) the forklift drivers are obeying posted speed limits throughout the facility, 2) the forklift drivers are completing one circuit in an appropriate length of time (with acceptable standard deviation) and 3) that drivers are completing enough circuits during a given shift.

Due to manufacturing facility regulations, only three communication protocols are allowed: 2.4 GHz Wireless, Bluetooth Low Energy (BLE) [26] and Zigbee Wireless [33]. No single protocol is available throughout the entire facility due to assembly machinery sensitivity to certain radio frequencies, but at least one of the three protocols is available for use in all of the areas that forklifts travel. In addition, there are areas in the facility where certain protocols are banned due to the aforementioned machinery sensitivity. It is a necessity that the forklift tracker be able to determine that it is inside one of these zones so that the protocol can be disabled. In addition, the tracker must select the best protocol from the remaining protocols to transmit data to the central receiver.

In this example, we will focus exclusively on the tracker which transmits data to a receiver. The receiver will then transmit the data to a database for storage and reporting. The development team of the manufacturing facility claimed responsibility for reading and analyzing data contained in the database. We were responsible for building the system and ensuring data reliably reached the database.

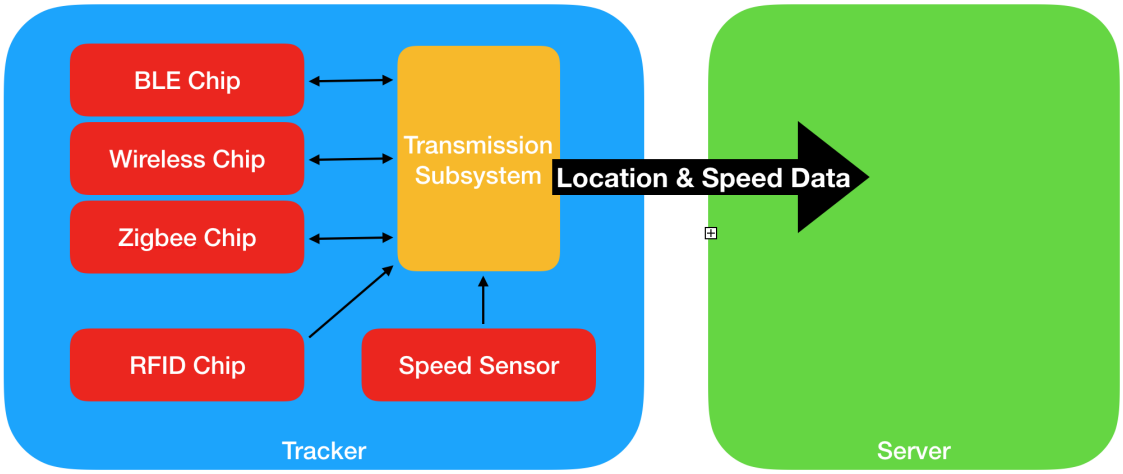

Figure 5.1: Tracking System Architecture

Determining how the forklift's location would be calculated proved to be a challenge. Global Positioning System (GPS) calculations were not possible due to interference as was triangulation due to sparsely located Wifi, BLE and Zigbee transmitters. It was eventually decided that passive Radio-Frequency IDentification (RFID) tags would be placed around the facility. They would be placed at the start and end of each "road" as well as at the entrance and exit of each "frequency-offline zone." The tracker would transmit the speed and last RFID tag read to the central receiver along with a timestamp.

A diagram overview of our tracking system is shown in figure 5.1. We will now provide and discuss the individual core components of the architecture.

```
1 device rfid_reader
2  features
3   x: out data port Base_Types::Float_64;
4   y: out data port Base_Types::Float_64;
5   tag_id: out data port Base_Types::String;
6 end rfid_reader;
```

Listing 5.1: Tracking System

In listing 5.1, we show the first primary component of the architecture: the interface for the RFID reader tag. This interface shows that the *rfid_reader* component produces 3 values. The first two are a x and a y value that represent the location of the RFID tag within the facility. The final

is the id of the tag itself.

```
abstract communication_module
 features
  power_on: in event port;
  power_off: in event port;
  signal_strength: out data port Base_Types::Integer;
  send: in event data port fork_lift_tracking_system::tracking_information;
  data_send_event: out event data port
      fork_lift_tracking_system::tracking_information;
end communication_module;
```

Listing 5.2: Tracking System

In listing 5.2, we have the abstract interface that each communication protocol implements. Each interface has two event ports, *power_on* and *power_off*, that allow the protocol to be enabled or disabled in software. The interface also allows each protocol, when powered on, to provide the current *signal_strength*. The interface also has two data events: *send* and *data_send_event*. *send* alerts the protocol to send a data packet and *data_send_event* performs the actual sending of data.

```
thread poll_positioning_tag
 features
  coordinates: requires data access fork_lift_tracking_system::coordinates;

  x: in data port Base_Types::Float_64;
  y: in data port Base_Types::Float_64;
  tag_id: in data port Base_Types::String;

  tracking_info: out event data port
      fork_lift_tracking_system::tracking_information;
end poll_positioning_tag;

thread implementation poll_positioning_tag.impl
 properties
  Dispatch_Protocol => Periodic;
  Period => 5000 ms;
```

```
16 annex behavior_specification {**
17  variables
18   previous_location: fork_lift_tracking_system::coordinates;
19   current_location: fork_lift_tracking_system::coordinates;
20   speed: Base_Types::Float_64;
21   tracking_information: fork_lift_tracking_system::tracking_information;
22  states
23   start: initial state;
24   offline: complete state;
25   online: state;
26  transitions
27   start -[]-> offline;
28   offline -[on dispatch]-> online {
29    previous_location := current_location;
30    coordinates.build_coordinates!(x, y, tag_id, current_location);
31    coordinates.compute_speed!(speed);
32    coordinates.build_tracking_information!(current_location, speed,
          tracking_information);
33    tracking_info!(tracking_information)
34   };
35   online -[]-> offline;
36 **};
37 end poll_positioning_tag.impl;
```

Listing 5.3: Tracking System

In listing 5.3, we have the thread which checks every 5 seconds for new RFID tag readings. Each reading is packaged and sent to the server via the active communication protocol, whatever that may be. The *behavior_specification* block of this component gives the behavior, in a state machine format, of the component. On each firing of the thread, a coordinate reading is taken, speed computed, and a tracking information data set built. All of this information is finally placed into the *tracking_info* event queue for sending.

```
1 thread monitor_signal_and_location
2  features
```

```
3   is_enabled_at_location_interface: requires data access
        fork_lift_tracking_system::is_enabled_at_location_interface;
4   tracking_information: requires data access
        fork_lift_tracking_system::tracking_information;
5
6   tracking_info_event: in event data port
        fork_lift_tracking_system::tracking_information;
7
8   wireless_power_on: out event port;
9   wireless_power_off: out event port;
10  wireless_signal_strength: in data port Base_Types::Integer;
11
12  bluetooth_power_on: out event port;
13  bluetooth_power_off: out event port;
14  bluetooth_signal_strength: in data port Base_Types::Integer;
15
16  radio_power_on: out event port;
17  radio_power_off: out event port;
18  radio_signal_strength: in data port Base_Types::Integer;
19
20  use_wifi: out event port;
21  use_ble: out event port;
22  use_radio: out event port;
23 end monitor_signal_and_location;
24
25 thread implementation monitor_signal_and_location.impl
26  annex behavior_specification {**
27   variables
28    tracking_info: fork_lift_tracking_system::tracking_information;
29    location: fork_lift_tracking_system::coordinates;
30    wireless_enabled: Base_Types::Boolean;
31    bluetooth_enabled: Base_Types::Boolean;
32    radio_enabled: Base_Types::Boolean;
```

```
33    wireless_rating: Base_Types::Integer;
34    bluetooth_rating: Base_Types::Integer;
35    radio_rating: Base_Types::Integer;
36   states
37    start: initial state;
38    offline: complete state;
39    online: state;
40   transitions
41    offline -[on dispatch tracking_info_event]-> online {
42      tracking_info_event?(tracking_info)
43    };
44    online -[]-> offline {
45     tracking_information.get_location!(tracking_info, location);
46     is_enabled_at_location_interface.is_wireless_enabled_at_location!(location,
           wireless_enabled);
47     is_enabled_at_location_interface.is_bluetooth_enabled_at_location!(location,
           bluetooth_enabled);
48     is_enabled_at_location_interface.is_radio_enabled_at_location!(location,
           radio_enabled);
49     if (wireless_enabled)
50      wireless_power_on!;
51      wireless_rating := wireless_signal_strength
52     else
53      wireless_power_off!;
54      wireless_rating := 0
55     end if;
56     if (bluetooth_enabled)
57      bluetooth_power_on!;
58      bluetooth_rating := bluetooth_signal_strength
59     else
60      bluetooth_power_off!;
61      bluetooth_rating := 0
62     end if;
```

```
63      if (radio_enabled)
64       radio_power_on!;
65       radio_rating := radio_signal_strength
66      else
67       radio_power_off!;
68       radio_rating := 0
69      end if;
70      if (wireless_rating > bluetooth_rating)
71       if (wireless_rating > radio_rating)
72        use_wifi!
73       end if
74      end if;
75      if (bluetooth_rating > wireless_rating)
76       if (bluetooth_rating > radio_rating)
77        use_ble!
78       end if
79      end if;
80      if (radio_rating > bluetooth_rating)
81       if (radio_rating > wireless_rating)
82        use_radio!
83       end if
84      end if
85     };
86   **};
87 end monitor_signal_and_location.impl;
```

Listing 5.4: Tracking System

In listing 5.4, we have the primary adaptive component of this architecture. This thread listens for new RFID tags to compute location, but it also uses these readings to determine if a communication protocol should be disabled. In addition to monitoring for RFID tag readings, this thread also monitors the signal strength of active protocols. After protocols have been enabled or disabled based on location, the protocol with the best signal strength is activated for use by the sending thread.

As previously mentioned, these components represent the core components of the architecture. There are additional components which "glue" the sending and adaptive threads together, however, in the following scenarios, we will focus exclusively on the above components along with any components that inherit from them (such as the *ble_module*, *radio_module* and *wifi_module* that extend the *communication_module*).

5.1.2.1 Forklift Tracking System Waterfall Scenario

In this scenario, we analyze the development of the tracking system using a waterfall based approach. The system will be developed over 10 time periods. Organizational training will occur during the first time period, and architecture / requirements will be co-generated during the second and third time periods continuing into the fourth time period. Development work will also begin in the fourth period and continue until the eighth period. Testing will start in the eighth period and continue until the tenth period. Deployment preparations will also occur in the tenth period. A visual overview of this scenario can be seen in figure 5.2.

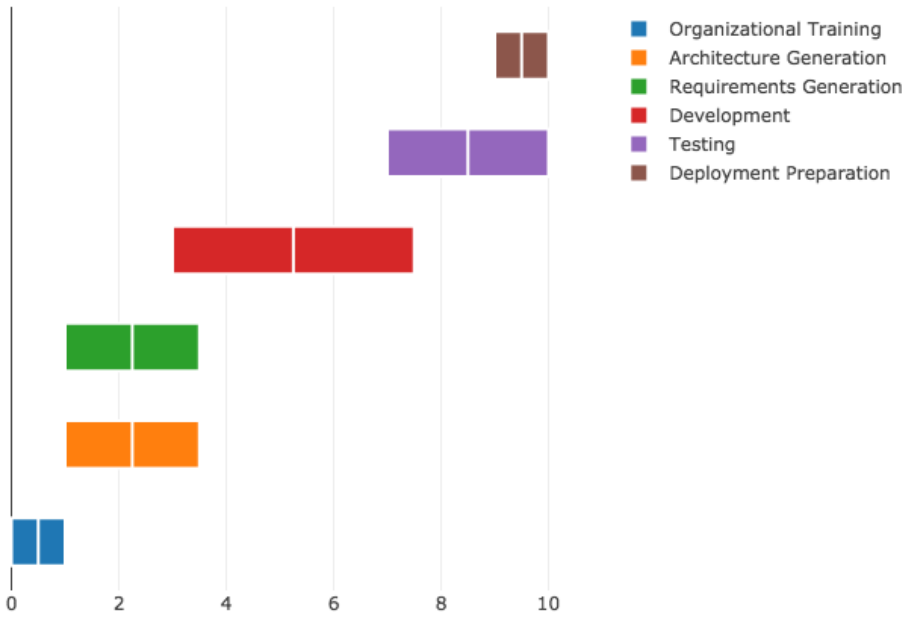

Figure 5.2: Example 1 Waterfall Timeline

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features,

subcomponents, behavior specification, etc.).

```
annex simdase {**
 time_min => 0.0;
 time_max => 9.0;
 variants => {
  "wifi_enabled":{
   "wireless_module","rfid_reader","send_tracking_info_via_wifi","protocol_management_process"
  };
  "ble_enabled":{
   "bluetooth_module","rfid_reader","send_tracking_info_via_ble","protocol_management_process"
  };
  "radio_enabled":{
   "radio_module","rfid_reader","send_tracking_info_via_radio","protocol_management_process"
  };
 };
 active_variants => { "fork_lift_tracker.impl": ["wifi_enabled",
     "ble_enabled", "radio_enabled"]; };
 cost => {
  organizational_planning => {
   2.0 "Project Manager" for if(t == 0.0) { 40.0 } else { 0.0 }
  };
  requirements_engineering => {
   2.0 "Business Analysts" for if(t == 1.0 || t == 2.0) { 40.0 } else { if(t
       == 3.0) { 20.0 } else { 0.0 } }
  };
  architecture_definition => {
   2.0 "Software Architects" for if(t == 1.0 || t == 2.0) { 30.0 } else {
       if(t == 3.0) { 20.0 } else { 0.0 } }
  };
  architecture_evaluation => {
   2.0 "Software Architects" for if(t == 1.0 || t == 2.0) { 10.0 } else {
       if(t == 3.0) { 5.0 } else { 0.0 } }
  };
```

```
29   component_development => {
30    3.0 "Software Developers" for if(t >= 3.0 && t <= 7.0) { 40.0 } else { 0.0
         }
31   };
32   testing => {
33    2.0 "Testers" for if(t >= 7.0 && t <= 9.0) { 40.0 } else { 0.0 }
34   };
35   software_system_integration => {
36    1.0 "System Administrator" for if(t == 9.0) { 40.0 } else { 0.0 }
37   };
38  };
39 **};
```

Listing 5.5: Tracking System Cost Scenario 1

In listing 5.5, we present the representation for specifying the hours required for the tracking system's first scenario. In this scenario, note that each task has the precise ranges specified during which the activity occurs. As activities do not reoccur on a regular basis, exact specification is acceptable. The output giving the total hours required for each time period can be found in figure 5.3. Note that once again, as activities do not reoccur, their costs appear only during the periods specified.

5.1.2.2 Forklift Tracking System Agile Scenario

In this scenario, we analyze the development of the tracking system using an agile based approach. The system will once again be developed over 10 time periods, however, it will be developed in 5 cycles of two periods each. The first cycle will consist of organizational training and restructuring. The remaining 4 cycles will consist of requirements / architecture generation (beginning of cycle), development (middle of cycle) and testing (end of cycle). A visual overview of this scenario can be seen in figure 5.4.

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features, subcomponents, behavior specification, etc.).

```
1 annex simdase {**
2  time_min => 0.0;
```

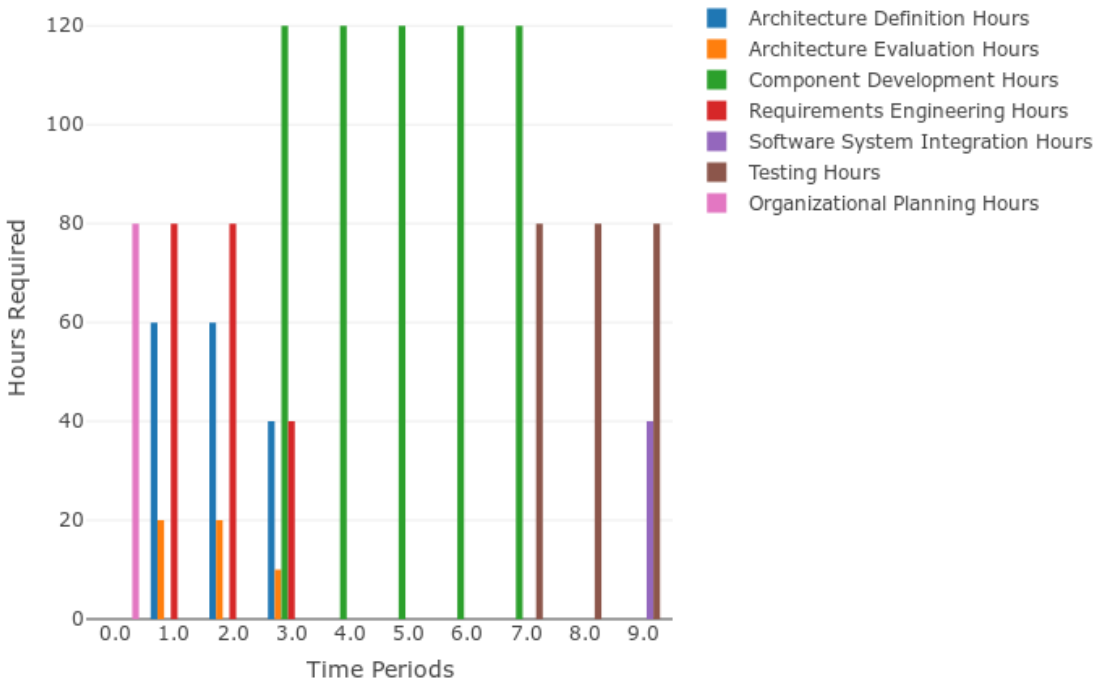

Figure 5.3: Tracking System Cost Report Scenario 1 (https://plot.ly/~bulletshot60/33)

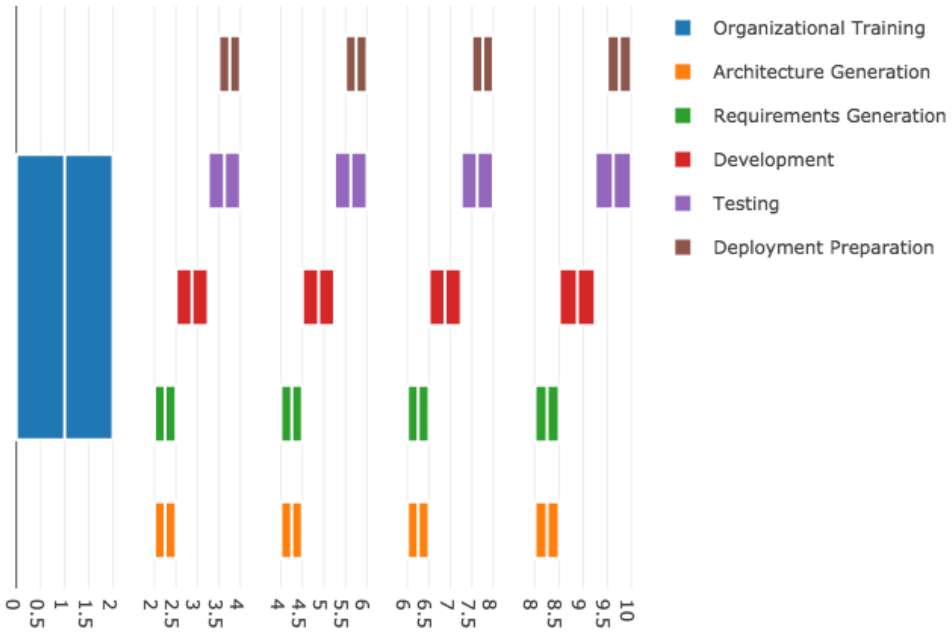

Figure 5.4: Example 1 Agile Timeline

```
time_max => 9.0;
variants => {
 "wifi_enabled":{
  "wireless_module","rfid_reader","send_tracking_info_via_wifi","protocol_management_process"
 };
 "ble_enabled":{
  "bluetooth_module","rfid_reader","send_tracking_info_via_ble","protocol_management_process"
 };
 "radio_enabled":{
  "radio_module","rfid_reader","send_tracking_info_via_radio","protocol_management_process"
 };
};
active_variants => { "fork_lift_tracker.impl": ["wifi_enabled",
    "ble_enabled", "radio_enabled"]; };
cost => {
 organizational_planning => {
  2.0 "Project Manager" for if(t == 0.0 || t == 1.0) { 40.0 } else { 0.0 }
 };
 requirements_engineering => {
  2.0 "Business Analysts" for if(t > 0.0 && t \% 2.0 == 0.0) { 20.0 } else {
      0.0 }
 };
 architecture_definition => {
  2.0 "Software Architects" for if(t > 0.0 && t \% 2.0 == 0.0) { 20.0 } else
      { 0.0 }
 };
 architecture_evaluation => {
  2.0 "Software Architects" for if(t > 0.0 && t \% 2.0 == 0.0) { 20.0 } else
      { 0.0 }
 };
 component_development => {
  3.0 "Software Developers" for if(t > 1.0) { if(t \% 2.0 == 0.0) { 20.0 }
      else { 10.0 } } else { 0.0 }
```

```
31    };
32    testing => {
33     2.0 "Testers" for if(t > 1.0 && t \% 2.0 == 1.0) { 30.0 } else { 0.0 }
34    };
35    software_system_integration => {
36     1.0 "System Administrator" for if(t > 1.0 && t \% 2.0 == 1.0) { 20.0 }
          else { 0.0 }
37    };
38   };
39 **};
```

Listing 5.6: Tracking System Cost Scenario 2

In listing 5.6, we present the representation for specifying the hours required for the tracking system's second scenario. In this scenario, costs that occur on a rotating basis (every other time period) are introduced. Note that such costs can be represented using the modulus operator on time. As will be shown in later examples, SIMDASE also supports more complex representations such as sinusoidal functions. The output giving the total hours required for each time period can be found in figure 5.5. Note that activities do reoccur now on a regular basis. This has the added benefit that the project's number of time periods can be extended without having to modify the annex's cost structure as long as the extension is a multiple of two.

5.1.2.3 Forklift Tracking System Undetermined Scenario

In this scenario, we analyze the development of the tracking system using an agile approach but when exact values for testing costs are undetermined. It has been decided that while it is possible to estimate some of the costs up front (such as organizational restructuring and training) other costs can only be estimated as a range. The system will still follow an agile timeline similar to that of scenarios 1 and 2.

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features, subcomponents, behavior specification, etc.).

```
1 annex simdase {**
2  time_min => 0.0;
```

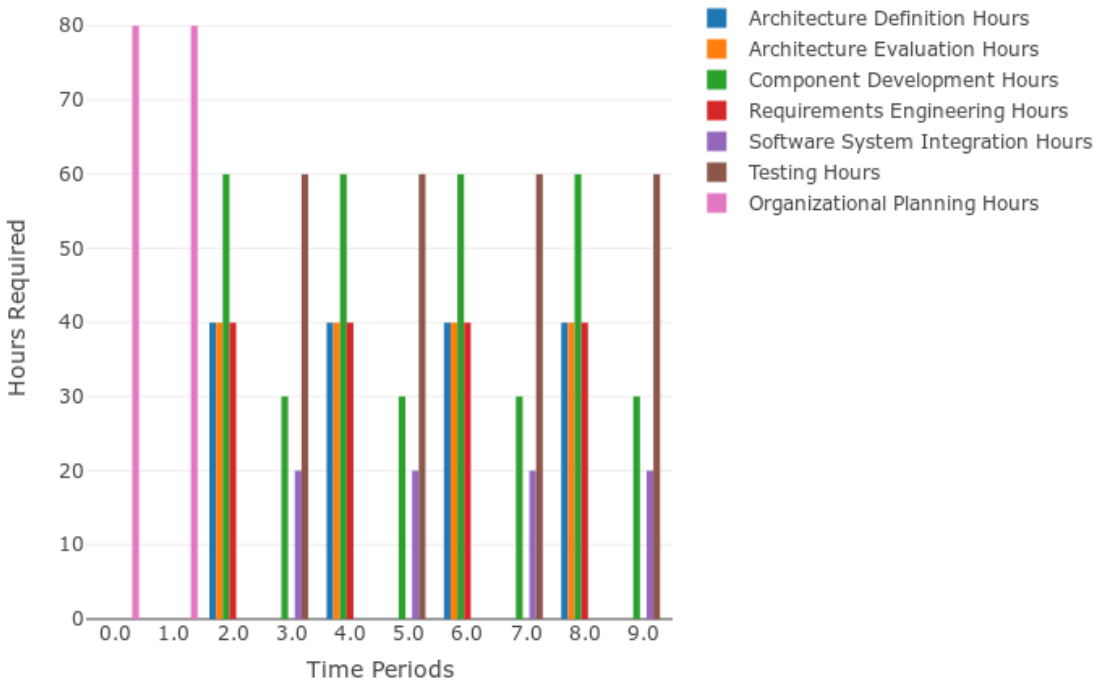

Figure 5.5: Tracking System Cost Report Scenario 2 (https://plot.ly/~bulletshot60/35)

```
time_max => 9.0;
variants => {
 "wifi_enabled":{
  "wireless_module","rfid_reader","send_tracking_info_via_wifi","protocol_management_process"
 };
 "ble_enabled":{
  "bluetooth_module","rfid_reader","send_tracking_info_via_ble","protocol_management_process"
 };
 "radio_enabled":{
  "radio_module","rfid_reader","send_tracking_info_via_radio","protocol_management_process"
 };
};
active_variants => { "fork_lift_tracker.impl": ["wifi_enabled",
    "ble_enabled", "radio_enabled"]; };
cost => {
 organizational_planning => {
```

```
18      2.0 "Project Manager" for if(t == 0.0 || t == 1.0) { 40.0 } else { 0.0 }
19    };
20    requirements_engineering => {
21      2.0 "Business Analysts" for if(t > 0.0 && t \% 2.0 == 0.0) { random() *
            40.0 } else { 0.0 }
22    };
23    architecture_definition => {
24      2.0 "Software Architects" for if(t > 0.0 && t \% 2.0 == 0.0) { random() *
            40.0 } else { 0.0 }
25    };
26    architecture_evaluation => {
27      2.0 "Software Architects" for if(t > 0.0 && t \% 2.0 == 0.0) { random() *
            40.0 } else { 0.0 }
28    };
29    component_development => {
30      3.0 "Software Developers" for if(t > 1.0) { if(t \% 2.0 == 0.0) { random()
            * 40.0 } else { random() * 20.0 } } else { 0.0 }
31    };
32    testing => {
33      2.0 "Testers" for if(t > 1.0 && t \% 2.0 == 1.0) { random() * 40.0 } else
            { 0.0 }
34    };
35    software_system_integration => {
36      1.0 "System Administrator" for if(t > 1.0 && t \% 2.0 == 1.0) { 20.0 }
            else { 0.0 }
37    };
38  };
39 **};
```

Listing 5.7: Tracking System Cost Scenario 3

In listing 5.7, we present the representation for specifying the hours required for the tracking system's final scenario. In this scenario, we continue with costs that occur on a rotating basis, but we introduce an unknown scaling factor (a randomly selected number between 0 and 1). This allows

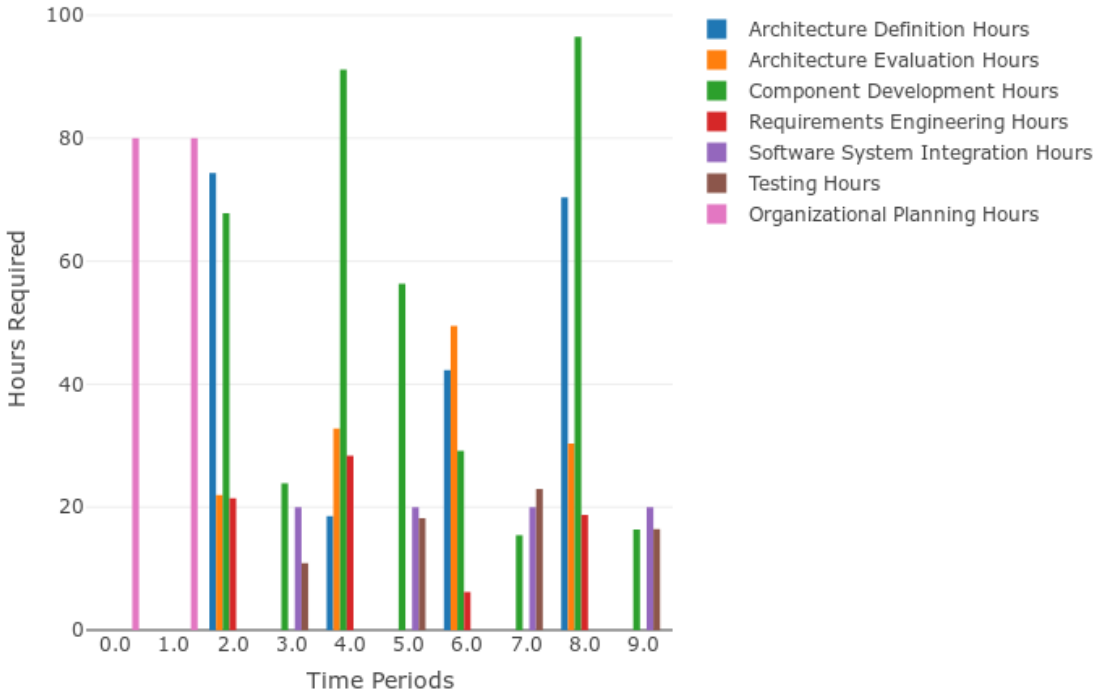

Figure 5.6: Tracking System Cost Report Scenario 3 (https://plot.ly/~bulletshot60/47)

us to represent costs for which we do not have a firm guess on the number of hours required. In this situation, the typical evaluation might include evaluating the annex repeatedly a sufficient number of types to determine the average, worst and best costs. (Such an analysis will be done as part of the evaluation for claim C2).

The output giving the total hours required for each time period can be found in figure 5.6. Note the increase in the precision of the costs introduced. However, with each run of the annex through the evaluator subsystem of SIMDASE, the values produced will change so the costs represented in the figure cannot be counted as exact or final; more analysis is needed. This is due to the use of the *random()* function in the annex which will pick a pseudo-random value between 0 and 1 inclusive with each call.

5.1.3 Dynamic Adaptive Nutrition System

The second system we introduce is a dynamic adaptive nutrition system [4]. The goal of this system is to monitor the total output of a country's farmers as well as the needs of each citizen. Through careful monitoring, the production of highly perishable goods, such as meat, dairy, bread

and vegetables, can be limited to only what is going to be used preventing spoilage.

An overview of the information flow of this system is shown in figure 5.7. The system has 7 major components: an individual caloric expenditure monitor, family meal planners, community needs monitors, national needs monitors, national food production monitors, national distributors and community distributors.

The architecture is divided up into four primary layers: the individual, the family, the community and the nation. Individuals have caloric needs based on their metabolic rate and activity levels. They also have food preferences and allergies. For each family, the caloric needs, allergies and preferences are aggregated and fused to produce a meal plan that satisfies the needs of the entire family without endangering them (causing contact with allergens). The meal plans of each family are then reported to the community monitor that aggregates the needs of all families within the community. This information is then reported to the national monitor which provides suggestions to farmers. Production from farmers is then passed into distributors which account for the needs of individual communities and families attempting to fulfill all needs and ensures fair distribution of goods. We will now describe each component while showing architecture snippets relating to that component.

We begin with the individual, shown in listing 5.8. The architecture model that we use to represent the system concerns itself only with the perishable goods previously mentioned: vegetables, bread, dairy and meat. Thus allergies to items not monitored are not considered nor is their production. In addition to allergies, the model also reports the individual's preference to each perishable item. This is used in the case of an individual placing himself on a specialized diet that excludes one of the four perishable food groups.

```
1 abstract individual
2  features
3   avg_caloric_needs: out data port Base_Types::Integer;
4   vegetable_allergy: out data port Base_Types::Boolean;
5   bread_allergy: out data port Base_Types::Boolean;
6   dairy_allergy: out data port Base_Types::Boolean;
7   meat_allergy: out data port Base_Types::Boolean;
8   no_vegetable_pref: out data port Base_Types::Boolean;
9   no_bread_pref: out data port Base_Types::Boolean;
```

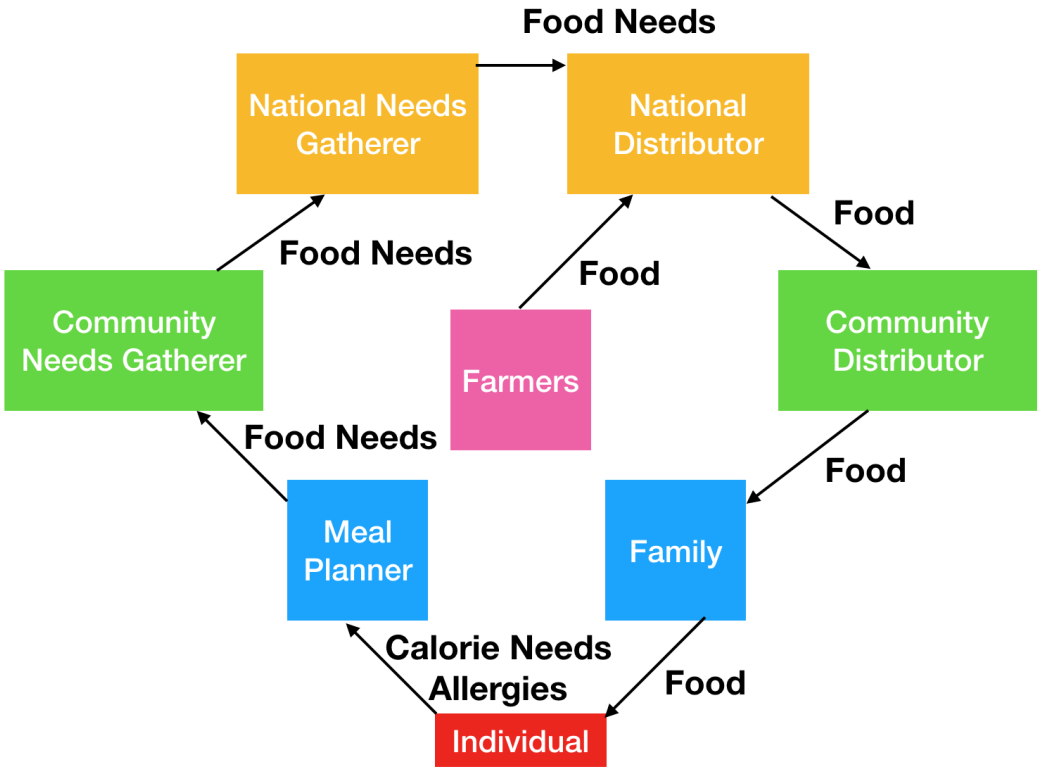

Figure 5.7: Nutrition System Information Flow

```
10    no_dairy_pref: out data port Base_Types::Boolean;
11    no_meat_pref: out data port Base_Types::Boolean;
12 end individual;
```

Listing 5.8: Nutrition System Individual

As previously mentioned each individual's calorie monitor reports to a dynamically adaptive meal planner, shown in listing 5.9. There are four types of meal planners: normal, allergy, preference and preference allergy. The meal planner for each family will dynamically adapt in response to family needs between one of these four.

The reasoning for dynamically adapting in the meal planner is due to the possibility of an individual having a new allergy diagnosed or gaining a new allergy due to medication. By dynamically adapting, the meal planner can alter the meals it generates accordingly. Because the allergy versions of the meal planner "forget" about the allergen, meaning that meals containing that

allergen are also forgotten, the allergy meal planner is safer than using if statements alone. Even if the allergy meal planner receives invalid input, it still cannot give meals containing the allergen.

```
abstract meal_planner_abstract
 features
  total_vegetables_needed: out data port Base_Types::Integer;
  total_bread_needed: out data port Base_Types::Integer;
  total_dairy_needed: out data port Base_Types::Integer;
  total_meat_needed: out data port Base_Types::Integer;
end meal_planner_abstract;

abstract implementation meal_planner_abstract.impl
 subcomponents
  member_1: abstract individual::individual.impl;
  member_2: abstract individual::individual.impl;
  member_3: abstract individual::individual.impl;
  member_4: abstract individual::individual.impl;
end meal_planner_abstract.impl;
```

Listing 5.9: Nutrition System Meal Planner

From the meal planner needs are aggregated to the community and national levels, both of which share the same architectural component description shown in listing 5.10. The gather takes the needs of each sub-unit (either communities or families) and anonymizes those needs so that they cannot be traced back to individual families or communities, protecting privacy and medical information.

```
abstract consumer_gatherer
 features
  total_vegetables_needed: out data port Base_Types::Integer;
  unit_1_vegetables_needed: out data port Base_Types::Integer;
  unit_2_vegetables_needed: out data port Base_Types::Integer;
  unit_3_vegetables_needed: out data port Base_Types::Integer;
  unit_4_vegetables_needed: out data port Base_Types::Integer;
  total_bread_needed: out data port Base_Types::Integer;
  unit_1_bread_needed: out data port Base_Types::Integer;
```

```
10    unit_2_bread_needed: out data port Base_Types::Integer;
11    unit_3_bread_needed: out data port Base_Types::Integer;
12    unit_4_bread_needed: out data port Base_Types::Integer;
13    total_dairy_needed: out data port Base_Types::Integer;
14    unit_1_dairy_needed: out data port Base_Types::Integer;
15    unit_2_dairy_needed: out data port Base_Types::Integer;
16    unit_3_dairy_needed: out data port Base_Types::Integer;
17    unit_4_dairy_needed: out data port Base_Types::Integer;
18    total_meat_needed: out data port Base_Types::Integer;
19    unit_1_meat_needed: out data port Base_Types::Integer;
20    unit_2_meat_needed: out data port Base_Types::Integer;
21    unit_3_meat_needed: out data port Base_Types::Integer;
22    unit_4_meat_needed: out data port Base_Types::Integer;
23  end consumer_gatherer;
24
25  abstract implementation consumer_gatherer.impl
26   subcomponents
27    common_1: abstract common_object;
28    common_2: abstract common_object;
29    common_3: abstract common_object;
30    common_4: abstract common_object;
31   connections
32    conn_1: port common_1.total_vegetables_needed -> unit_1_vegetables_needed;
33    conn_2: port common_2.total_vegetables_needed -> unit_2_vegetables_needed;
34    conn_3: port common_3.total_vegetables_needed -> unit_3_vegetables_needed;
35    conn_4: port common_4.total_vegetables_needed -> unit_4_vegetables_needed;
36    conn_5: port common_1.total_bread_needed -> unit_1_bread_needed;
37    conn_6: port common_2.total_bread_needed -> unit_2_bread_needed;
38    conn_7: port common_3.total_bread_needed -> unit_3_bread_needed;
39    conn_8: port common_4.total_bread_needed -> unit_4_bread_needed;
40    conn_9: port common_1.total_dairy_needed -> unit_1_dairy_needed;
41    conn_10: port common_2.total_dairy_needed -> unit_2_dairy_needed;
42    conn_11: port common_3.total_dairy_needed -> unit_3_dairy_needed;
```

```
43   conn_12: port common_4.total_dairy_needed -> unit_4_dairy_needed;
44   conn_13: port common_1.total_meat_needed -> unit_1_meat_needed;
45   conn_14: port common_2.total_meat_needed -> unit_2_meat_needed;
46   conn_15: port common_3.total_meat_needed -> unit_3_meat_needed;
47   conn_16: port common_4.total_meat_needed -> unit_4_meat_needed;
48 end consumer_gatherer.impl;
```

Listing 5.10: Nutrition System Gatherer

Finally, once needs have been announced and farmers have produced goods to fulfill them, the food units are passed through distributors that divide the goods up among markets from which families can purchase the goods needed to fill their meal plans. The distributor's definition is shown in listing 5.11.

There are two types of distributors in the architecture between which the system dynamically swaps: a normal distributor and an excess distributor. The excess distributor is used when food production generates more units than are needed. In this case, the additional units are stored temporarily (until their expiration date nears) in case of a food shortage. Both community distributors and national distributors have the ability to store.

The reasoning behind introducing dynamic adaptivity into the distributor is due to food production continuously changing. Droughts, environmental disasters, excess rain, etc can all affect harvests and food production. Dynamically adapting to ever changing variables is one of the hallmarks of a dynamic system.

```
1 abstract abstract_redistributor
2  features
3   total_vegetables_received: in data port Base_Types::Integer;
4   unit_1_vegetables_needed: in data port Base_Types::Integer;
5   unit_2_vegetables_needed: in data port Base_Types::Integer;
6   unit_3_vegetables_needed: in data port Base_Types::Integer;
7   unit_4_vegetables_needed: in data port Base_Types::Integer;
8   total_bread_received: in data port Base_Types::Integer;
9   unit_1_bread_needed: in data port Base_Types::Integer;
10  unit_2_bread_needed: in data port Base_Types::Integer;
11  unit_3_bread_needed: in data port Base_Types::Integer;
```

```
12   unit_4_bread_needed: in data port Base_Types::Integer;
13   total_dairy_received: in data port Base_Types::Integer;
14   unit_1_dairy_needed: in data port Base_Types::Integer;
15   unit_2_dairy_needed: in data port Base_Types::Integer;
16   unit_3_dairy_needed: in data port Base_Types::Integer;
17   unit_4_dairy_needed: in data port Base_Types::Integer;
18   total_meat_received: in data port Base_Types::Integer;
19   unit_1_meat_needed: in data port Base_Types::Integer;
20   unit_2_meat_needed: in data port Base_Types::Integer;
21   unit_3_meat_needed: in data port Base_Types::Integer;
22   unit_4_meat_needed: in data port Base_Types::Integer;
23 end abstract_redistributor;
24
25 abstract implementation abstract_redistributor.impl
26  subcomponents
27   common_1: abstract common_object;
28   common_2: abstract common_object;
29   common_3: abstract common_object;
30   common_4: abstract common_object;
31   veggie_divider: abstract divider.impl;
32   bread_divider: abstract divider.impl;
33   dairy_divider: abstract divider.impl;
34   meat_divider: abstract divider.impl;
35 end abstract_redistributor.impl;
```

Listing 5.11: Nutrition System Distributor

5.1.3.1 Dynamic Adaptive Nutrition System Waterfall Scenario

For example 2's first scenario, we begin with a traditional waterfall development scenario which will be modified in the remaining scenarios. In this scenario, whose timeline is shown in figure 5.8, development will consist of 166 time periods (each one week long) with 3 phases. Phase 1 will consist of the caloric expenditure sensor development followed by development of the meal planners. In the final phase the aggregators and distributors will be co-developed.

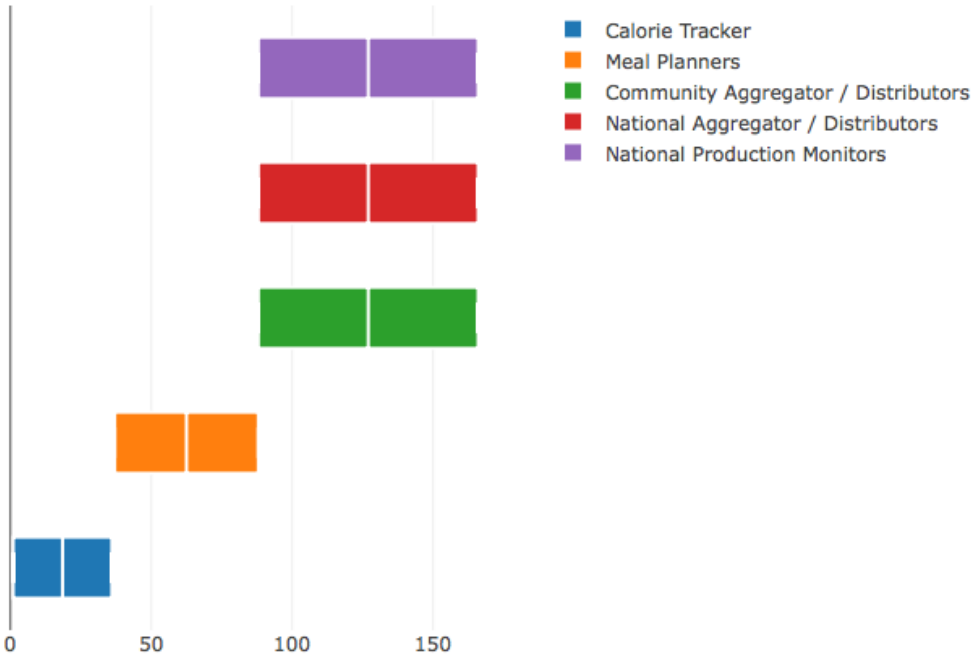

Figure 5.8: Nutrition System Scenario 1 Timeline

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features, subcomponents, behavior specification, etc.). The output from the evaluation of the SIMDASE instantiation is shown in figure 5.9. Note that the bulk of the costs, development of the redistributors / aggregators, are situated towards the end of development.

We begin with the SIMDASE instantiation for the individual, shown in listing 5.12. To simplify the annex instantiations, we will focus solely on building the components of the architecture not their planning. In the instantiation for the individual, we see that it will be developed for 36 time periods with requirements, architecture, development and testing spread throughout the time periods.

```
1 abstract implementation individual.impl
2  annex simdase {**
3   time_min => 1.0;
4   time_max => 36.0;
5   cost => {
```

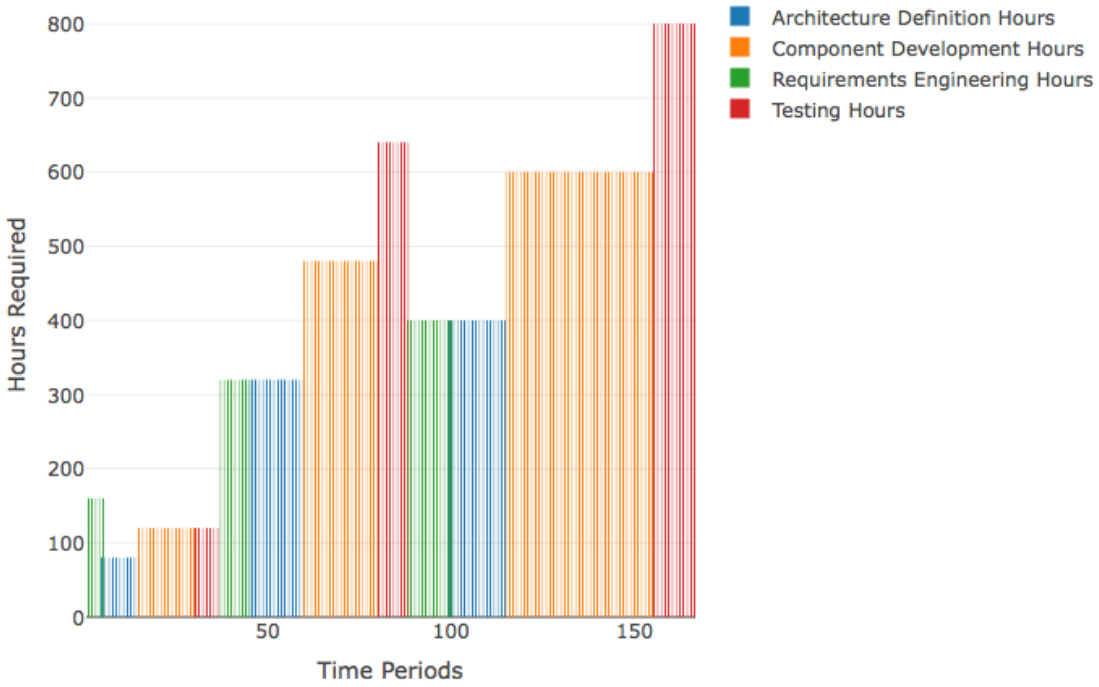

Figure 5.9: Nutrition System Scenario 1 Output (https://plot.ly/~bulletshot60/43)

```
6   requirements_engineering => {
7    4.0 "Business Analysts" for if(1.0 <= t && t <= 5.0) { 40.0 } else { 0.0 }
8   };
9   architecture_definition => {
10   2.0 "Software Architects" for if(5.0 <= t && t <= 15.0) { 40.0 } else {
         0.0 }
11  };
12  component_development => {
13   3.0 "Software Developers" for if(15.0 <= t && t <= 30.0) { 40.0 } else {
         0.0 }
14  };
15  testing => {
16   2.0 "Software Testers" for if(30.0 <= t && t <= 36.0) { 40.0 } else { 0.0
         },
17   1.0 "Hardware Testers" for if(30.0 <= t && t <= 36.0) { 40.0 } else { 0.0
         }
```

```
18     };
19    };
20   **};
21 end individual.impl;
```

Listing 5.12: Nutrition System Scenario 1 Individual

Next, we show, in listing 5.13, the instantiation of the meal planner, which will be developed from periods 37 to 88. The other meal planners (allergy, preference and allergy preference), not shown, will be co-developed alongside the normal meal planner.

```
1 abstract implementation meal_planner.impl extends meal_planner_abstract.impl
2  annex simdase {**
3   time_min => 37.0;
4   time_max => 88.0;
5   cost => {
6    requirements_engineering => {
7     2.0 "Business Analysts" for if(37.0 <= t && t <= 45.0) { 40.0 } else {
          0.0 }
8    };
9    architecture_definition => {
10    2.0 "Software Architects" for if(45.0 <= t && t <= 60.0) { 40.0 } else {
          0.0 }
11   };
12   component_development => {
13    3.0 "Software Developers" for if(60.0 <= t && t <= 80.0) { 40.0 } else {
          0.0 }
14   };
15   testing => {
16    2.0 "Software Testers" for if(80.0 <= t && t <= 88.0) { 40.0 } else { 0.0
          },
17    2.0 "Hardware Testers" for if(80.0 <= t && t <= 88.0) { 40.0 } else { 0.0
          }
18   };
19  };
```

```
 **};
end meal_planner.impl;
```

Listing 5.13: Nutrition System Scenario 1 Meal Planner

Next, we show, in listing 5.14, the instantiation of the community gatherer, which will be developed from periods 88 to 166. As the same gatherer specification is used both at the community and the national levels, a scaling factor is applied here. This allows the costs to vary from the national to the community while reusing architectural components. It is assumed that developing a community gatherer will be easier than a national one, thus the hours are scaled so that less hours will be required from each person for the community gatherer.

```
abstract implementation consumer_gatherer.impl
 annex simdase {**
  time_min => 88.0;
  time_max => 166.0;
  scaling_factor => 1.0/(i+1.0);
  cost => {
   requirements_engineering => {
    2.0 "Business Analysts" for if(88.0 <= t && t <= 100.0) { 40.0 } else {
        0.0 }
   };
   architecture_definition => {
    2.0 "Software Architects" for if(100.0 <= t && t <= 115.0) { 40.0 } else
        { 0.0 }
   };
   component_development => {
    3.0 "Software Developers" for if(115.0 <= t && t <= 155.0) { 40.0 } else
        { 0.0 }
   };
   testing => {
    2.0 "Software Testers" for if(155.0 <= t && t <= 166.0) { 40.0 } else {
        0.0 },
    2.0 "Hardware Testers" for if(155.0 <= t && t <= 166.0) { 40.0 } else {
        0.0 }
```

```
19     };
20    };
21   **};
22 end consumer_gatherer.impl;
```

Listing 5.14: Nutrition System Scenario 1 Gatherer

Finally, we show, in listing 5.15, the definition of the distributor which will be co-developed alongside the gatherer. The distributor, like the gatherer, is scaled as the same component definition is used at the community and national levels yet different costs can be used in each level.

```
1  abstract implementation abstract_redistributor.impl
2   annex simdase {**
3    time_min => 88.0;
4    time_max => 166.0;
5    scaling_factor => 1.0/(i+1.0);
6    cost => {
7     requirements_engineering => {
8      2.0 "Business Analysts" for if(88.0 <= t && t <= 100.0) { 40.0 } else {
           0.0 }
9     };
10    architecture_definition => {
11     2.0 "Software Architects" for if(100.0 <= t && t <= 115.0) { 40.0 } else
           { 0.0 }
12    };
13    component_development => {
14     3.0 "Software Developers" for if(115.0 <= t && t <= 155.0) { 40.0 } else
           { 0.0 }
15    };
16    testing => {
17     2.0 "Software Testers" for if(155.0 <= t && t <= 166.0) { 40.0 } else {
           0.0 },
18     2.0 "Hardware Testers" for if(155.0 <= t && t <= 166.0) { 40.0 } else {
           0.0 }
19    };
```

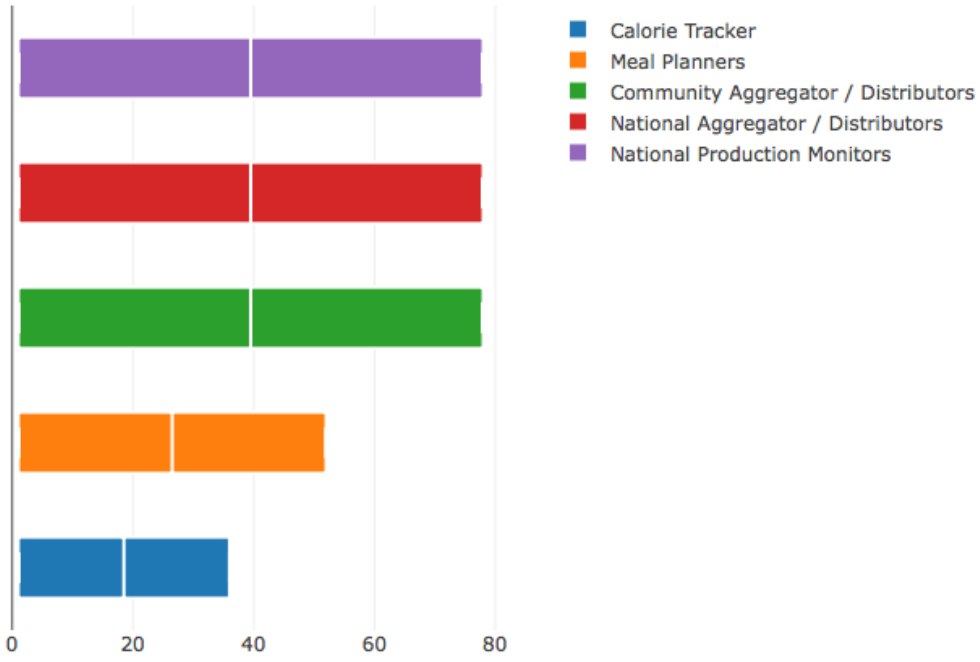

Figure 5.10: Nutrition System Scenario 2 Timeline

```
20    };
21   **};
22 end abstract_redistributor.impl;
```

Listing 5.15: Nutrition System Scenario 1 Distributor

5.1.3.2 Dynamic Adaptive Nutrition System Components During Same Periods Scenario

In the nutrition system's second scenario, shown in figure 5.10, all components will be co-developed with the caloric tracker finishing development first followed by the meal planners. Finally, the gatherers and distributors will follow the meal planners. As each system finishes development, the result of the development will be integrated into the next component to ensure correct functionality.

In this scenario, annex instantiations are the same as the previous scenario except for time minimum and time maximum. The minimum time is set to 1.0 and the maximum time has been moved backwards to reflect the change in the minimum time. The evaluation of the SIMDASE instantiation is shown in figure 5.11. Note that costs are more evenly spread now. In addition, some

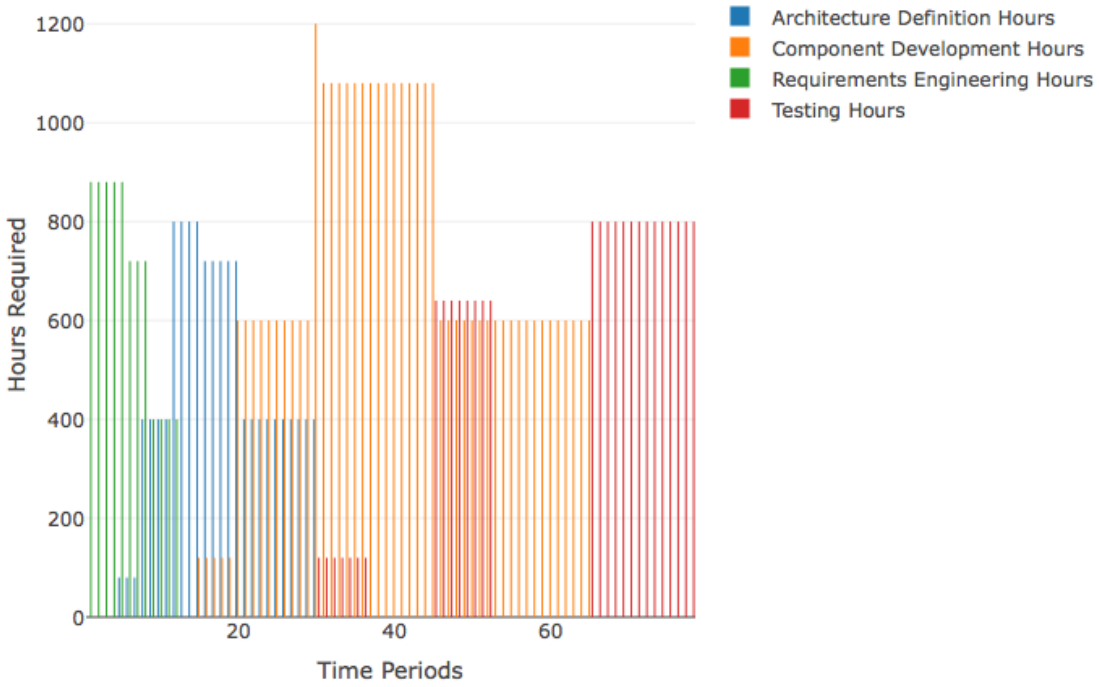

Figure 5.11: Nutrition System Scenario 2 Output (https://plot.ly/~bulletshot60/41)

costs, like requirements engineering, is done for all components around the same time rather than being spread out over time. This would allow the requirements engineers from this project to be allocated for a shorter period of time, allowing them to be moved to other projects as their work on the nutrition monitor completes.

5.1.3.3 Dynamic Adaptive Nutrition System Purchased Components Scenario

In the nutrition system's third scenario, shown in figure 5.12, it has been decided that a caloric expenditure tracker will not be built. A pre-existing tracker, developed by another company, will be used in its place. For purposes of this example, the Fitbit Alta will be used and several will be procured for development purposes.

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features, subcomponents, behavior specification, etc.). The evaluation of the SIMDASE annex is shown in figure 5.13. Note the small reductions in time needed for requirements engineering and architecture definition found in periods 1 through 20.

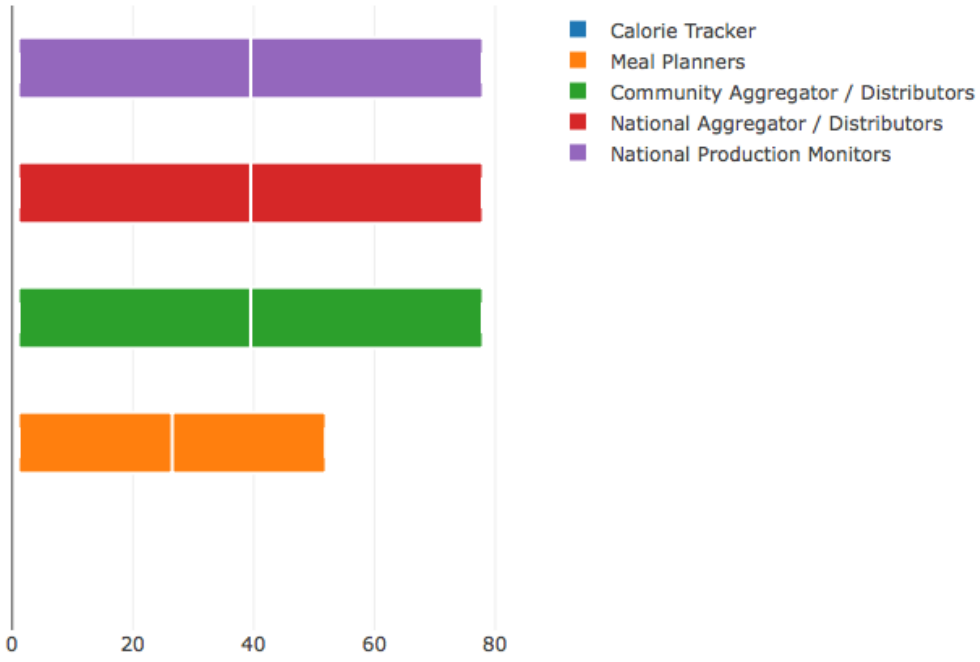

Figure 5.12: Nutrition System Scenario 3 Timeline

For this scenario, only the individual SIMDASE instantiation, shown in listing 5.16, needs to change. The system will now be developed in 5 time periods. Procurement of the Altas for development and testing will occur during the first time period. Development (integration of the Fitbit API) and testing will occur after procurement.

```
1  abstract implementation individual.impl
2   annex simdase {**
3    time_min => 1.0;
4    time_max => 5.0;
5    cost => {
6     component_development => {
7      1200.0 for "Fitbit Procurement" at 1.0
8     };
9     software_system_integration => {
10     2.0 "Developers" for if(1.0 <= t && t <= 3.0) { 40.0 } else { 0.0 },
11     1.0 "Tester" for if(3.0 <= t && t <= 5.0) { 40.0 } else { 0.0 }
12    };
```

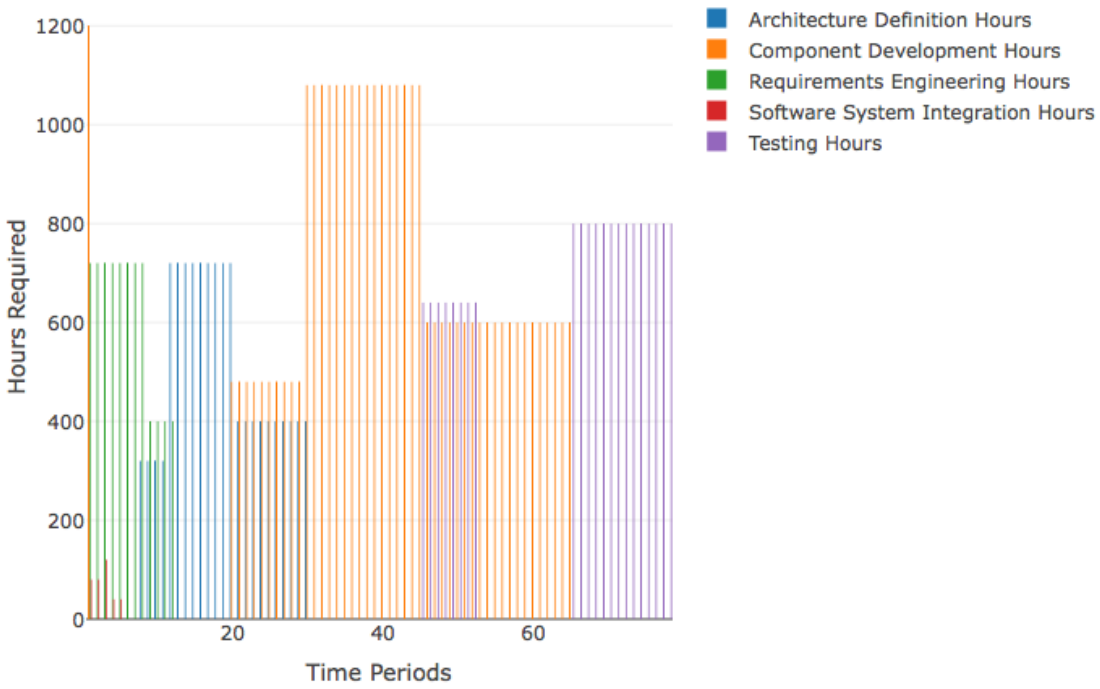

Figure 5.13: Nutrition System Scenario 3 Output (https://plot.ly/~bulletshot60/39)

```
13    };
14   **};
15 end individual.impl;
```

Listing 5.16: Nutrition System Scenario 3 Individual

5.1.3.4 Dynamic Adaptive Nutrition System Undetermined Scenario

In the final scenario for the nutrition system, it has been decided that it is not possible to directly estimate the costs of each component and that some costs, for the meal planner, will fluctuate throughout the development process.

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features, subcomponents, behavior specification, etc.). The evaluation of the SIMDASE instantiation is shown in figure 5.14. Note the wide variability in each period introduced now where as before costs were relatively stable.

The individual component, which consists of procured components does not change for this

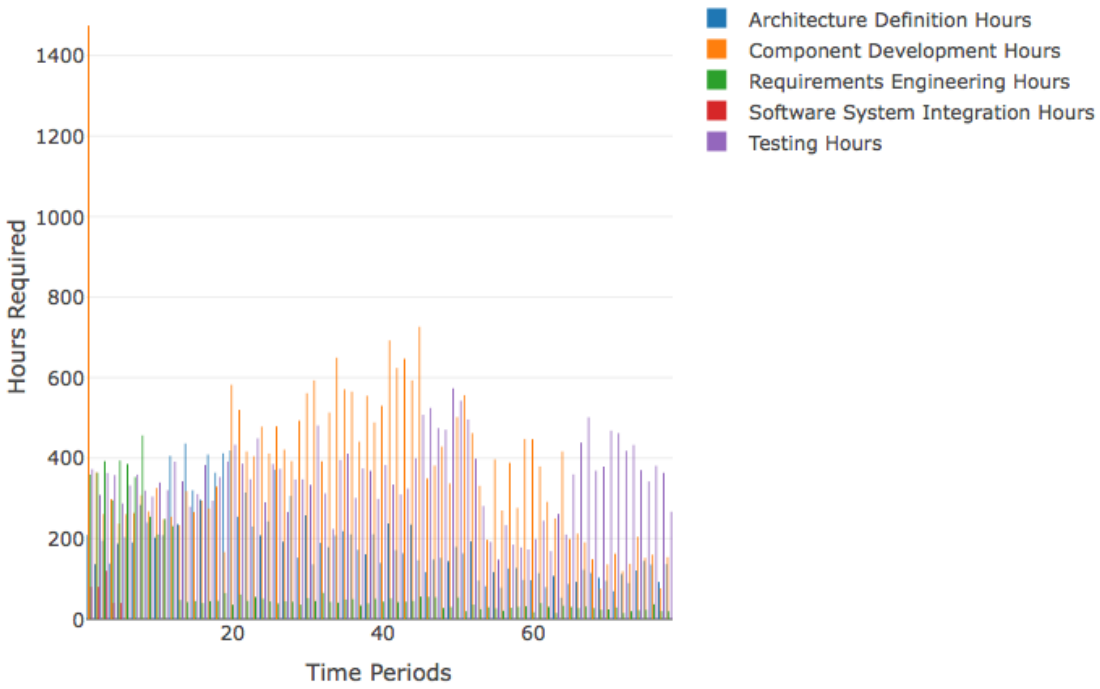

Figure 5.14: Nutrition System Scenario 4 Output (https://plot.ly/~bulletshot60/37)

scenario. Thus, we first present the SIMDASE instantiation for the the meal planner component, shown in listing 5.17. In this scenario, note that the meal planner costs are placed on a sinusoidal wave as well as randomized. This will cause costs to fluctuate greatly between 0 and 60 throughout the entire development of the meal planner.

```
1  abstract implementation meal_planner.impl extends meal_planner_abstract.impl
2   annex simdase {**
3    time_min => 1.0;
4    time_max => 52.0;
5    cost => {
6     requirements_engineering => {
7      2.0 "Business Analysts" for if(1.0 <= t && t <= 8.0) { ((Math.cos(t) + 1)
           * random()) * 40.0 } else { random() * 5.0 }
8     };
9     architecture_definition => {
10     2.0 "Software Architects" for if(8.0 <= t && t <= 20.0) { ((Math.cos(t) +
           1) * random()) * 40.0 } else { random() * 20.0 }
```

```
11     };
12     component_development => {
13      3.0 "Software Developers" for if(20.0 <= t && t <= 45.0) { ((Math.cos(t)
            + 1) * random()) * 40.0 } else { random() * 20.0 }
14     };
15     testing => {
16      2.0 "Software Testers" for if(45.0 <= t && t <= 52.0) { ((Math.cos(t) +
            1) * random()) * 40.0 } else { random() * 20.0 },
17      2.0 "Hardware Testers" for if(45.0 <= t && t <= 52.0) { ((Math.cos(t) +
            1) * random()) * 40.0 } else { random() * 20.0 }
18     };
19    };
20   **};
21 end meal_planner.impl;
```

Listing 5.17: Nutrition System Scenario 4 Meal Planner

For the remaining two systems, the gatherer (shown in listing 5.18) and the distributor (shown in listing 5.19), now have random fluctuations also inserted. These fluctuations will not vary as greatly as those of the meal planner and they will remain between 0 and 40.

```
1 abstract implementation consumer_gatherer.impl
2  annex simdase {**
3   time_min => 1.0;
4   time_max => 78.0;
5   scaling_factor => 1.0/(i+1.0);
6   cost => {
7    requirements_engineering => {
8     2.0 "Business Analysts" for if(1.0 <= t && t <= 12.0) { random() * 40.0 }
          else { random() * 5.0 }
9    };
10   architecture_definition => {
11    2.0 "Software Architects" for if(12.0 <= t && t <= 30.0) { random() *
          40.0 } else { random() * 20.0 }
12   };
```

```
13    component_development => {
14     3.0 "Software Developers" for if(30.0 <= t && t <= 65.0) { random() *
          40.0 } else { random() * 20.0 }
15    };
16    testing => {
17     2.0 "Software Testers" for if(65.0 <= t && t <= 78.0) { random() * 40.0 }
          else { random() * 20.0 },
18     2.0 "Hardware Testers" for if(65.0 <= t && t <= 78.0) { random() * 40.0 }
          else { random() * 20.0 }
19    };
20   };
21  **};
22 end consumer_gatherer.impl;
```

Listing 5.18: Nutrition System Scenario 4 Gatherer

```
1 abstract implementation abstract_redistributor.impl
2  annex simdase {**
3   time_min => 1.0;
4   time_max => 78.0;
5   scaling_factor => 1.0/(i+1.0);
6   cost => {
7    requirements_engineering => {
8     2.0 "Business Analysts" for if(1.0 <= t && t <= 12.0) { random() * 40.0 }
          else { random() * 5.0 }
9    };
10   architecture_definition => {
11    2.0 "Software Architects" for if(12.0 <= t && t <= 30.0) { random() *
          40.0 } else { random() * 20.0 }
12   };
13   component_development => {
14    3.0 "Software Developers" for if(30.0 <= t && t <= 65.0) { random() *
          40.0 } else { random() * 20.0 }
15   };
```

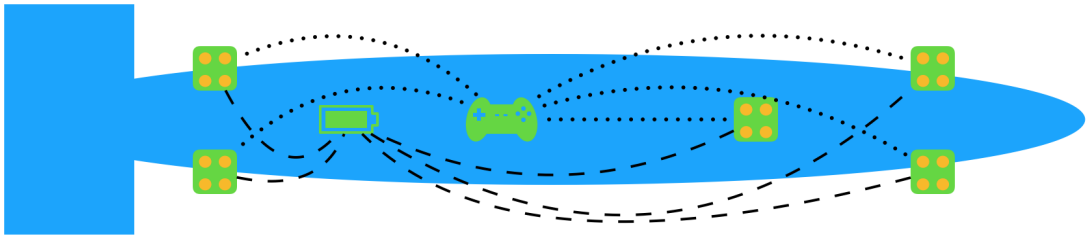

Figure 5.15: Underwater Vehicle System Architecture

```
16     testing => {
17       2.0 "Software Testers" for if(65.0 <= t && t <= 78.0) { random() * 40.0 }
             else { random() * 20.0 },
18       2.0 "Hardware Testers" for if(65.0 <= t && t <= 78.0) { random() * 40.0 }
             else { random() * 20.0 }
19     };
20   };
21  **};
22 end abstract_redistributor.impl;
```

Listing 5.19: Nutrition System Scenario 4 Distributor

5.1.4 Dynamic Adaptive Underwater Vehicle

The final system we introduce is a dynamic adaptive underwater manned vehicle [24]. The vehicle, whose architecture is given in figure 5.15, has 5 sets of sensors, 4 external sets and 1 internal set, that monitor the environment. The underwater vehicle also has a mission controller that reads from the sensors as well as an energy source that enables / disables the sensors and provides them power in accordance with the mission controller's plan.

The sensors, of which there are 7 types, all have the same profile. They each have a port allowing them to be disabled and enabled. In addition, they have ports allowing them to report the power draw, sensor readings and error rates. In each external sensor group there is a water salinity sensor, a water temperature sensor, a water pressure sensor and a radar sensor. In the internal group, there is a carbon dioxide sensor, a carbon monoxide sensor, an oxygen sensor and a air pressure sensor.

```
abstract sensor
 features
  powered_on: in event data port Base_Types::Boolean;
  power_draw: out data port Base_Types::Integer;
  sensor_data: out event data port Base_Types::Integer;
  error_rate: out data port Base_Types::Integer;
end sensor;
```

Listing 5.20: Underwater Vehicle System Sensor

Each sensor is attached to an adaptive energy source, shown in listing 5.21 that continuously monitors the power draw as well as power production. In cases where power draw exceeds power production, the energy source can disable sensors in a round robin like manner ensuring that each sensor is enabled for a short time to produce data before it is disabled again. The energy source has three modes, normal, degraded and critical, that it can adapt between freely. Each adaptation dictates how many sensors should be disabled. Note that the internal sensor group always has priority over the external sensor groups.

Due to the inability of underwater vehicles to recharge, it was decided that dynamic adaptivity was a necessary component of the energy source. If for any reason the energy source degrades, or power consumption spikes above what the source is able to provide, it can adapt to the new scenario.

```
abstract abstract_power_manager
 features
  exterior_sensor_left_front_1_powered_on: out event data port
      Base_Types::Boolean;
  exterior_sensor_left_front_2_powered_on: out event data port
      Base_Types::Boolean;
  exterior_sensor_left_front_3_powered_on: out event data port
      Base_Types::Boolean;
  exterior_sensor_left_front_4_powered_on: out event data port
      Base_Types::Boolean;
  exterior_sensor_right_front_1_powered_on: out event data port
      Base_Types::Boolean;
```

```
8   exterior_sensor_right_front_2_powered_on: out event data port
        Base_Types::Boolean;
9   exterior_sensor_right_front_3_powered_on: out event data port
        Base_Types::Boolean;
10  exterior_sensor_right_front_4_powered_on: out event data port
        Base_Types::Boolean;
11  exterior_sensor_left_rear_1_powered_on: out event data port
        Base_Types::Boolean;
12  exterior_sensor_left_rear_2_powered_on: out event data port
        Base_Types::Boolean;
13  exterior_sensor_left_rear_3_powered_on: out event data port
        Base_Types::Boolean;
14  exterior_sensor_left_rear_4_powered_on: out event data port
        Base_Types::Boolean;
15  exterior_sensor_right_rear_1_powered_on: out event data port
        Base_Types::Boolean;
16  exterior_sensor_right_rear_2_powered_on: out event data port
        Base_Types::Boolean;
17  exterior_sensor_right_rear_3_powered_on: out event data port
        Base_Types::Boolean;
18  exterior_sensor_right_rear_4_powered_on: out event data port
        Base_Types::Boolean;
19  inside_sensor_1_powered_on: out event data port Base_Types::Boolean;
20  inside_sensor_2_powered_on: out event data port Base_Types::Boolean;
21  inside_sensor_3_powered_on: out event data port Base_Types::Boolean;
22  inside_sensor_4_powered_on: out event data port Base_Types::Boolean;
23
24  exterior_sensor_left_front_1_power_draw: in data port Base_Types::Integer;
25  exterior_sensor_left_front_2_power_draw: in data port Base_Types::Integer;
26  exterior_sensor_left_front_3_power_draw: in data port Base_Types::Integer;
27  exterior_sensor_left_front_4_power_draw: in data port Base_Types::Integer;
28  exterior_sensor_right_front_1_power_draw: in data port Base_Types::Integer;
29  exterior_sensor_right_front_2_power_draw: in data port Base_Types::Integer;
```

```
30    exterior_sensor_right_front_3_power_draw: in data port Base_Types::Integer;
31    exterior_sensor_right_front_4_power_draw: in data port Base_Types::Integer;
32    exterior_sensor_left_rear_1_power_draw: in data port Base_Types::Integer;
33    exterior_sensor_left_rear_2_power_draw: in data port Base_Types::Integer;
34    exterior_sensor_left_rear_3_power_draw: in data port Base_Types::Integer;
35    exterior_sensor_left_rear_4_power_draw: in data port Base_Types::Integer;
36    exterior_sensor_right_rear_1_power_draw: in data port Base_Types::Integer;
37    exterior_sensor_right_rear_2_power_draw: in data port Base_Types::Integer;
38    exterior_sensor_right_rear_3_power_draw: in data port Base_Types::Integer;
39    exterior_sensor_right_rear_4_power_draw: in data port Base_Types::Integer;
40    inside_sensor_1_power_draw: in data port Base_Types::Integer;
41    inside_sensor_2_power_draw: in data port Base_Types::Integer;
42    inside_sensor_3_power_draw: in data port Base_Types::Integer;
43    inside_sensor_4_power_draw: in data port Base_Types::Integer;
44 end abstract_power_manager;
```

Listing 5.21: Underwater Vehicle System Energy Source

In addition to being attached to an energy source, each sensor is also attached to the mission controller, shown in listing 5.22. The mission controller, like the energy source, can also adapt between a normal and a degraded mode. The degraded mode has additional functionality that allows the mission controller to continue operation even if insufficient data is being received.

The decision to make the energy source dynamic was due to the need for the controller to continue operation even if sensors fail. As the data and error rates of the sensors are variable, adding adaptivity to accommodate this fluctuation was deemed necessary.

```
1 abstract mission_controller_abstract
2  features
3    interior_sensor_1_sensor_data: in event data port Base_Types::Integer;
4    interior_sensor_1_error_rate: in data port Base_Types::Integer;
5    interior_sensor_2_sensor_data: in event data port Base_Types::Integer;
6    interior_sensor_2_error_rate: in data port Base_Types::Integer;
7    interior_sensor_3_sensor_data: in event data port Base_Types::Integer;
8    interior_sensor_3_error_rate: in data port Base_Types::Integer;
9    interior_sensor_4_sensor_data: in event data port Base_Types::Integer;
```

```
10  interior_sensor_4_error_rate: in data port Base_Types::Integer;
11  exterior_left_front_sensor_1_sensor_data: in event data port
        Base_Types::Integer;
12  exterior_left_front_sensor_1_error_rate: in data port Base_Types::Integer;
13  exterior_left_front_sensor_2_sensor_data: in event data port
        Base_Types::Integer;
14  exterior_left_front_sensor_2_error_rate: in data port Base_Types::Integer;
15  exterior_left_front_sensor_3_sensor_data: in event data port
        Base_Types::Integer;
16  exterior_left_front_sensor_3_error_rate: in data port Base_Types::Integer;
17  exterior_left_front_sensor_4_sensor_data: in event data port
        Base_Types::Integer;
18  exterior_left_front_sensor_4_error_rate: in data port Base_Types::Integer;
19  exterior_right_front_sensor_1_sensor_data: in event data port
        Base_Types::Integer;
20  exterior_right_front_sensor_1_error_rate: in data port Base_Types::Integer;
21  exterior_right_front_sensor_2_sensor_data: in event data port
        Base_Types::Integer;
22  exterior_right_front_sensor_2_error_rate: in data port Base_Types::Integer;
23  exterior_right_front_sensor_3_sensor_data: in event data port
        Base_Types::Integer;
24  exterior_right_front_sensor_3_error_rate: in data port Base_Types::Integer;
25  exterior_right_front_sensor_4_sensor_data: in event data port
        Base_Types::Integer;
26  exterior_right_front_sensor_4_error_rate: in data port Base_Types::Integer;
27  exterior_left_rear_sensor_1_sensor_data: in event data port
        Base_Types::Integer;
28  exterior_left_rear_sensor_1_error_rate: in data port Base_Types::Integer;
29  exterior_left_rear_sensor_2_sensor_data: in event data port
        Base_Types::Integer;
30  exterior_left_rear_sensor_2_error_rate: in data port Base_Types::Integer;
31  exterior_left_rear_sensor_3_sensor_data: in event data port
        Base_Types::Integer;
```

```
32  exterior_left_rear_sensor_3_error_rate: in data port Base_Types::Integer;
33  exterior_left_rear_sensor_4_sensor_data: in event data port
        Base_Types::Integer;
34  exterior_left_rear_sensor_4_error_rate: in data port Base_Types::Integer;
35  exterior_right_rear_sensor_1_sensor_data: in event data port
        Base_Types::Integer;
36  exterior_right_rear_sensor_1_error_rate: in data port Base_Types::Integer;
37  exterior_right_rear_sensor_2_sensor_data: in event data port
        Base_Types::Integer;
38  exterior_right_rear_sensor_2_error_rate: in data port Base_Types::Integer;
39  exterior_right_rear_sensor_3_sensor_data: in event data port
        Base_Types::Integer;
40  exterior_right_rear_sensor_3_error_rate: in data port Base_Types::Integer;
41  exterior_right_rear_sensor_4_sensor_data: in event data port
        Base_Types::Integer;
42  exterior_right_rear_sensor_4_error_rate: in data port Base_Types::Integer;
43
44  use_interior_sensor_1: out data port Base_Types::Boolean;
45  use_interior_sensor_2: out data port Base_Types::Boolean;
46  use_interior_sensor_3: out data port Base_Types::Boolean;
47  use_interior_sensor_4: out data port Base_Types::Boolean;
48  use_exterior_l_f_sensor_1: out data port Base_Types::Boolean;
49  use_exterior_l_f_sensor_2: out data port Base_Types::Boolean;
50  use_exterior_l_f_sensor_3: out data port Base_Types::Boolean;
51  use_exterior_l_f_sensor_4: out data port Base_Types::Boolean;
52  use_exterior_r_f_sensor_1: out data port Base_Types::Boolean;
53  use_exterior_r_f_sensor_2: out data port Base_Types::Boolean;
54  use_exterior_r_f_sensor_3: out data port Base_Types::Boolean;
55  use_exterior_r_f_sensor_4: out data port Base_Types::Boolean;
56  use_exterior_l_r_sensor_1: out data port Base_Types::Boolean;
57  use_exterior_l_r_sensor_2: out data port Base_Types::Boolean;
58  use_exterior_l_r_sensor_3: out data port Base_Types::Boolean;
59  use_exterior_l_r_sensor_4: out data port Base_Types::Boolean;
```

```
60    use_exterior_r_r_sensor_1: out data port Base_Types::Boolean;
61    use_exterior_r_r_sensor_2: out data port Base_Types::Boolean;
62    use_exterior_r_r_sensor_3: out data port Base_Types::Boolean;
63    use_exterior_r_r_sensor_4: out data port Base_Types::Boolean;
64 end mission_controller_abstract;
```

Listing 5.22: Underwater Vehicle System Mission Control

5.1.5 Dynamic Adaptive Underwater Vehicle Waterfall Scenario

We present a single cost scenario for the underwater vehicle system, a traditional waterfall scenario, whose timeline is shown in figure 5.16. In this scenario, the software for the underwater vehicle will be developed in 5 overlapping phases. During the first phase, sensor readers that can determine the error rate and power draw will be developed. During the second phase, the adaptive energy managers will be developed. Phase 3 will be used to develop the two mission controllers. Finally, phase 4 will be used to integrate the sensors and phase 5 will be used for full system testing. Time periods once again are one week.

We now provide architecture snippets with the SIMDASE implementation of the above scenario. Note, to conserve space, previously listed component parts are omitted (i.e. features, subcomponents, behavior specification, etc.). The evaluation of the SIMDASE instantiation is shown in figure 5.17. Note that some subcomponents have widely fluctuating costs while some have steady recurring costs. Also note that sensor procurement costs, which occur during later periods, are represented as relatively expensive, one-time costs.

In listing 5.23, we present the SIMDASE instantiation for one of the 8 sensors used in the system. The sensors will be developed and integrated into readers, one at time, throughout the fourth phase. This sensor will be developed from period 78 to 88. The sensor will be purchased during the first period and integration and testing will occur during the remainder of the period.

```
1 device implementation water_temperature_sensor.impl extends sensor.impl
2  annex simdase {**
3   time_min => 78.0;
4   time_max => 88.0;
5   cost => {
6    using_externally_available_software => {
```

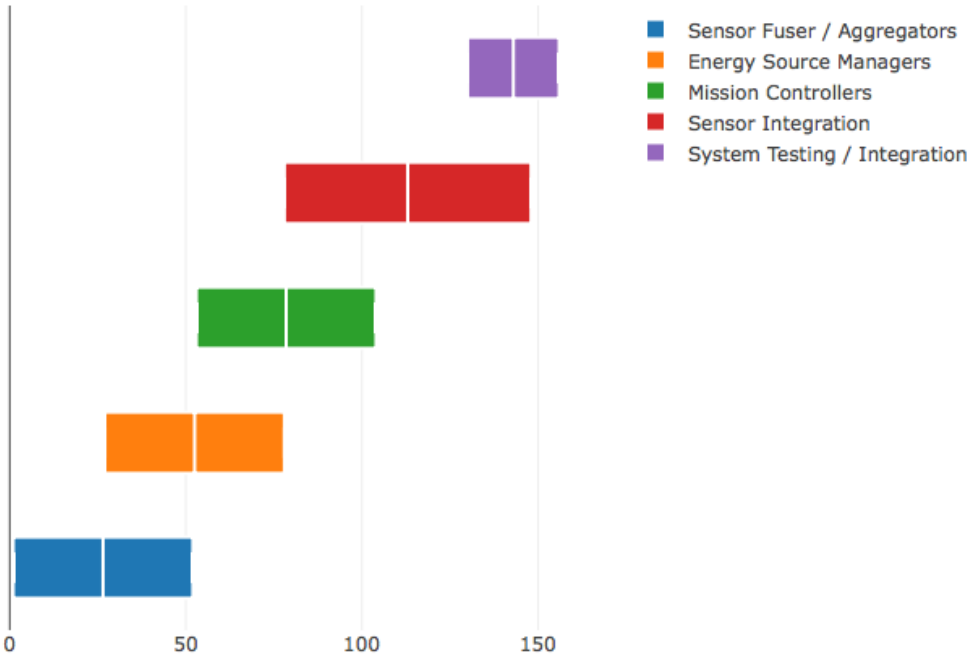

Figure 5.16: Underwater Vehicle System Scenario 1 Timeline

```
7       600.0 for "sensor purchase" at 78.0
8      };
9      software_system_integration => {
10      2.0 "Software Developers" for 40.0,
11      2.0 "Testers" for 40.0
12     };
13    };
14   **};
15 end water_temperature_sensor.impl;
```

Listing 5.23: Underwater Vehicle System Scenario 1 Sensor

In listing 5.24, we present the SIMDASE instantiation for the normal energy manager which will be developed during periods 27 to 48. The remaining two energy managers will be co-developed alongside the normal manager.

```
1 abstract implementation normal_power_manager.impl extends
      abstract_power_manager.impl
```

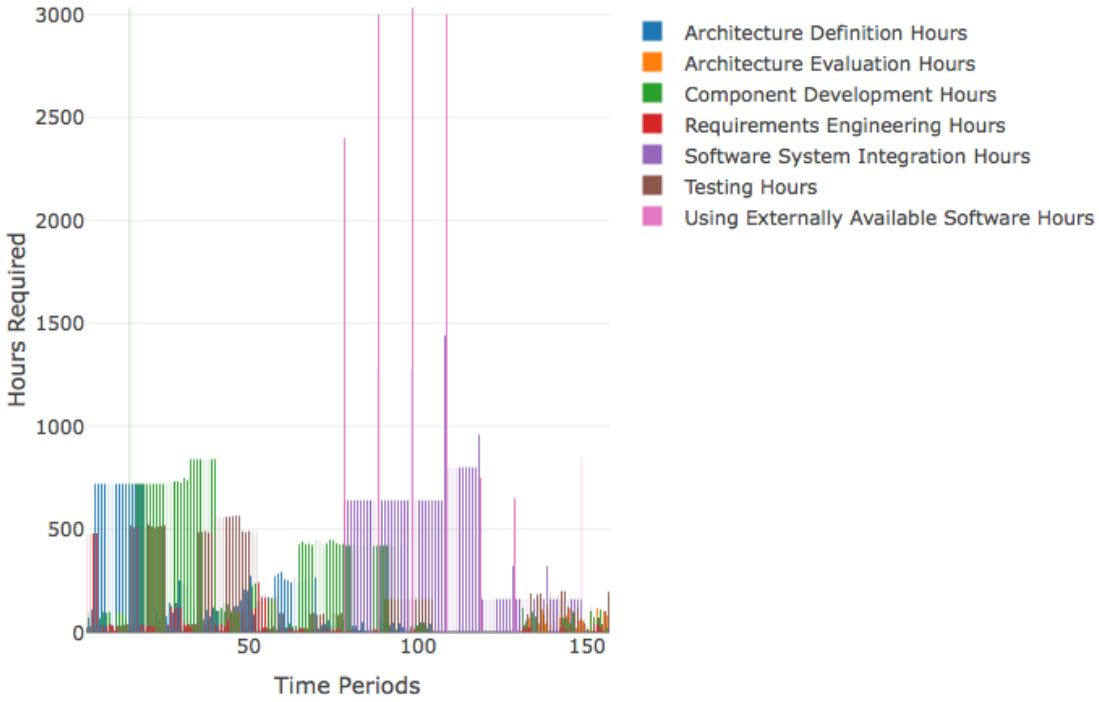

Figure 5.17: Underwater Vehicle System Scenario 1 Output (https://plot.ly/~bulletshot60/45)

```
annex simdase {**
 time_min => 27.0;
 time_max => 47.0;
 cost => {
  requirements_engineering => {
   2.0 "Business Analysts" for if(27.0 <= t && t <= 30.0) { 40.0 } else {
       5.0 * random() }
  };
  architecture_definition => {
   3.0 "Software Architects" for if(30.0 <= t && t <= 33.0) { 40.0 } else {
       10.0 * random() }
  };
  component_development => {
   3.0 "Software Developers" for if(33.0 <= t && t <= 40.0) { 40.0 } else {
       10.0 * random() }
  };
```

```
15    testing => {
16     2.0 "Software Testers" for if(40.0 <= t && t <= 47.0) { 40.0 } else { 5.0
          * random() }
17    };
18   };
19  **};
20 end normal_power_manager.impl;
```

Listing 5.24: Underwater Vehicle System Scenario 1 Energy Source

In listing 5.24, we present the SIMDASE instantiation for the normal mission controller which will be developed during periods 53 to 104. The remaining mission controller will be co-developed alongside the normal controller.

```
1 abstract implementation normal_mission_controller.impl extends
     mission_controller_abstract.impl
2  annex simdase {**
3   time_min => 53.0;
4   time_max => 104.0;
5   cost => {
6    requirements_engineering => {
7     2.0 "Business Analysts" for if(53.0 <= t && t <= 58.0) { 40.0 } else {
          5.0 * random() }
8    };
9    architecture_definition => {
10    3.0 "Software Architects" for if(58.0 <= t && t <= 70.0) { 40.0 } else {
          10.0 * random() }
11   };
12   component_development => {
13    3.0 "Software Developers" for if(65.0 <= t && t <= 95.0) { 70.0 } else {
          10.0 * random() }
14   };
15   testing => {
16    2.0 "Software Testers" for if(90.0 <= t && t <= 104.0) { 40.0 } else {
          5.0 * random() }
```

```
17      };
18    };
19   **};
20 end normal_mission_controller.impl;
```

Listing 5.25: Underwater Vehicle System Scenario 1 Mission Control

5.1.5.1 Summary

In this section, we have introduced three examples of varying complexity and scenarios for each example. For each scenario, we have provided an instantiation of the SIMDASE annex representing the introduced cost scenario as well as a visual representation of the output produced by the SIMDASE evaluator.

The purpose of these examples is to demonstrate that SIMDASE is sufficiently flexible to support a number of common and uncommon cost scenarios. For each scenario, we have also demonstrated not only the ability to represent the scenario in the architectural model but the ability to evaluate the representation to produce analyzable data.

5.2 Demonstrating Cost / Effort Estimation Model Robustness

Claim C2: SIMDASE's effort estimation formula is robust with regard to input parameters, including user-defined input parameters that could return variodic or sinusoidal values.

5.2.1 Evaluation Plan

To demonstrate claim C2, we take the most complex scenario from each example introduced in the flexibility analysis and we perform a sensitivity analysis on each. These examples contain user-defined sinusoidal functions, ranges, periodic / random scaling and other complexities that can have an undetermined effect on cost. To ensure that both our cost / effort framework and our evaluations of these examples are sound, we repeatedly evaluated the models 100, 10000 and 1000000 times taking the average, mean, minimum and maximum cost each time. We show that the estimation model responds appropriately no matter the input.

The primary reason for this claim is not due to the input being wildly variable, although that is a factor. The evaluation of an annex instantiation is not influenced solely by user input, but also by the architectural style chosen as well as time. Some of the analyses for this section will demonstrate this.

5.2.2 Tracking System Undetermined Scenario Sensitivity Analysis

For example 1 in this evaluation, we will use the third scenario of the tracking system which includes multiple unknown values. We will specifically focus on time period 2 which includes both architecture activities (definition and evaluation) as well as development activities. For architecture definition, 2 software architects will work between 0 and 40 hours during the time period. For architecture evaluation, 2 software architects will work between 0 and 40 hours during the time period. For development, 3 software developers will work between 0 and 40 hours during the time period. In the simulation results below, we will report our results using 3 values. The first will be the number (between 0 and 100) representing the percentage of time used by software architects. The second will be the number (between 0 and 100) representing the percentage of time used by software developers. The final will be a number representing the total number of hours worked by both software architects and software developers during the simulation of time period 2.

In figure 5.18, we present the analysis resulting from repeatedly evaluating the tracking system third scenario's model 100 times. As was previously stated, the y axis represents the software developers percentage of hours used and the x axis represents the software architects percentage of hours used. The z axis represents the cost, or number of hours required by both groups.

For this simulation, several discontinuities were present due to randomly selecting scaling factors for each group. With only 100 selections, several groups of scaling factors were not selected. Discounting the discontinuities, the graph has the expected gradually increasing slope.

In figure 5.18, we present the analysis resulting from repeatedly evaluating the tracking system third scenario's model 10,000 times. In this model, there are less discontinuities present due to the larger number of evaluations. Once again, discounting the few discontinuities present, the graph has the expected slope from 0,0 to 100,100.

In figure 5.18, we present the analysis resulting from repeatedly evaluating the tracking system third scenario's model 1,000,000 times. In this model, there are even less discontinuities present than the 10,000 simulation. Once again, discounting the few discontinuities present, the

graph has the expected slope from 0,0 to 100,100.

5.2.3 Dynamic Adaptive Nutrition System Undetermined Scenario Sensitivity Analysis

For example 2 of the sensitivity analysis, we will use the data from the fourth scenario of the nutrition system, and we will use this data to provide two different sensitivity analysis. The first will be with respect to architecture definition and component development. The second will be with respect to architecture definition and time.

For the first analysis, we will use component development costs (percentage of 40 hours used) as one axis and software architecture costs (percentage of 40 hours used) as the other axis. As before, we will conduct the analysis at a particular point in time, period 20. The z axis will, as with the previous sensitivity analysis, represent total total hours needed. This analysis provides results similar to our previous sensitivity analysis as can be seen in figures 5.21, 5.22 and 5.23.

In figure 5.21, several discontinuities are present, yet as expected, they disappear with larger iteration counts (figures 5.22 and 5.23). Overall, these data plots show a stable increase from 0 to 80 with no unexpected spikes.

As SIMDASE evaluations are influenced not only by user-entered cost estimations, it is necessary to perform sensitivity analyses that take into account other external factors. One of the external factors that can affect estimations is that of time.

For the second analysis, the SIMDASE instantiation for the fourth scenario of the nutrition system also includes costs that exist along a sinusoidal wave. As this wave varies based on the time period, we will do another sensitivity analysis using time as one of the axes. Architecture definition, represented by the percentage of 40 hours used, will be the other axis. An example of doing this with 100 randomly selected data plots can be seen in figure 5.24.

No outstanding plot points appear in the 100 iteration version. The same is true for both 10000 (figure 5.25) and 1000000 (figure 5.26). However, with more plot points randomly chosen, the sinusoidal nature of the costs are easier to visualize.

Both analyses using the data from the fourth scenario of the nutrition system result in a stable evaluation despite the usage of a sinusoidal wave.

5.2.4 Dynamic Adaptive Underwater Vehicle Sensitivity Analysis

The other external influencing factor of cost estimations is that of the architecture style. SIMDASE uses the structure the architecture to produce cost estimations. As such, performing a sensitivity analysis that includes architecture scaling was deemed necessary. For this simulation, we randomly generated 100 architectures and assigned the same SIMDASE instantiation to one of the components.

For the final sensitivity analysis, we use the sole scenario of the underwater vehicle. This scenario includes our most complex input, a component that includes a cost scaled logarithmically based on the component's depth in the architecture. In the following graphs, we once again evaluate the SIMDASE instantiation 100, 10000 and 1000000 times. One axis will represent the architecture definition costs (percentage of 40 hours used) and the second axis will represent the depth of the chosen component within the randomly generated architectures (between 0 and 100).

Figure 5.27 results in a very uninteresting graph. The logarithmic nature of the cost scaling can barely be discerned due to the large number of discontinuities. However, no unexpected rises or falls are visible either.

In figure 5.28, the logarithmic nature of the cost scaling starts to become visible as is the effect of randomly selecting a percentage of hours used for the architecture definition costs. The full nature of both are, as expected, visible in the 1000000 iteration version (figure 5.29).

This analysis demonstrates that SIMDASE is also able to robustly and reliably handle complex scaling such as an inverted logarithmic scale in addition to sinusoidal costs.

5.2.4.1 Summary

In this evaluation, we have taken three examples which contain one or more unknown development costs. Each example was evaluated 100, 10,000 and 1,000,000 times plotting the results on a topological graph.

The purpose of this evaluation was to demonstrate the robustness (no unexpected irregularities) of SIMDASE with regard to user defined input as well as external factors. In example 1, we provided a simple example containing no sinusoidal functions or scaling. This demonstrates that SIMDASE can robustly evaluate simple scenarios. Example 2 introduced sinusoidal functions and example 3 introduced logarithmic functions. These examples taken together show that SIMDASE

can also robustly evaluate complex and specific scenarios.

5.3 Demonstrating Improvements of XAGREE over AGREE

Claim C3: XAGREE, in general, reduces the size of verification assets, reduces the complexity of verification assets, and reduces the amount of duplication across verification assets when compared to traditional AGREE.

5.3.1 Evaluation Plan

To demonstrate claim C3, we verify the models introduced in the flexibility analysis using both XAGREE and AGREE. We then measure the cyclomatic complexity [35], duplication and lines of code of each analysis. We show that, in general, XAGREE has equal or lower average cyclomatic complexity across all verification assets, equal or lower duplicated lines of code across all verification assets, and equal or lower lines of code across all verification assets.

Note, for simplicity, we will show samples containing both the AGREE and XAGREE annexes for the first example only. For cases where the AGREE and XAGREE annexes are exactly the same, the AGREE annex will contain only a comment stating that it is an exact duplicate of the XAGREE annex. For the remaining examples, we will list only the measured values of the verification assets.

5.3.2 Tracking System Verification / Validation

Verification of the tracking system is comprised of evaluating two claims. First, it must be verified that all data produced by the system is within the expected range, and, second, it must be verified that the system sends information via one and only one protocol at a time.

In listing 5.26, we present the verification of information being sent via only one protocol. Note that the evaluation also includes verification of data being within range. Since AGREE / XAGREE reason compositionally about models, this is only a part of the evaluation for that claim. We will present additional components containing verification assets for that claim shortly.

```
1 system fork_lift_tracker
2  annex xagree {**
3   guarantee "wifi x is positive": wifi_tracking_x >= 0.0;
```

```
4   guarantee "wifi y is positive": wifi_tracking_y >= 0.0;
5   guarantee "wifi error margin is positive": wifi_tracking_error_margin >=
        0.0;
6   guarantee "wifi speed is positive": wifi_tracking_speed >= 0.0;
7
8   guarantee "bluetooth x is positive": bluetooth_tracking_x >= 0.0;
9   guarantee "bluetooth y is positive": bluetooth_tracking_y >= 0.0;
10  guarantee "bluetooth error margin is positive":
        bluetooth_tracking_error_margin >= 0.0;
11  guarantee "bluetooth speed is positive": bluetooth_tracking_speed >= 0.0;
12
13  guarantee "radio x is positive": radio_tracking_x >= 0.0;
14  guarantee "radio y is positive": radio_tracking_y >= 0.0;
15  guarantee "radio error margin is positive": radio_tracking_error_margin >=
        0.0;
16  guarantee "radio speed is positive": radio_tracking_speed >= 0.0;
17
18  guarantee "x is only sent by one protocol at a time": (wifi_tracking_x >=
        0.0 and bluetooth_tracking_x = 0.0 and radio_tracking_x = 0.0) or
19                  (wifi_tracking_x = 0.0 and bluetooth_tracking_x >= 0.0 and
                        radio_tracking_x = 0.0) or
20                  (wifi_tracking_x = 0.0 and bluetooth_tracking_x = 0.0 and
                        radio_tracking_x >= 0.0);
21  guarantee "y is only sent by one protocol at a time": (wifi_tracking_y >=
        0.0 and bluetooth_tracking_y = 0.0 and radio_tracking_y = 0.0) or
22                  (wifi_tracking_y = 0.0 and bluetooth_tracking_y >= 0.0 and
                        radio_tracking_y = 0.0) or
23                  (wifi_tracking_y = 0.0 and bluetooth_tracking_y = 0.0 and
                        radio_tracking_y >= 0.0);
24  guarantee "error margin is only sent by one protocol at a time":
        (wifi_tracking_error_margin >= 0.0 and bluetooth_tracking_error_margin
        = 0.0 and radio_tracking_error_margin = 0.0) or
25                          (wifi_tracking_error_margin = 0.0 and
```

```
                                bluetooth_tracking_error_margin >= 0.0 and
                                radio_tracking_error_margin = 0.0) or
26                          (wifi_tracking_error_margin = 0.0 and
                                bluetooth_tracking_error_margin = 0.0 and
                                radio_tracking_error_margin >= 0.0);
27   guarantee "speed is only sent by one protocol at a time":
        (wifi_tracking_speed >= 0.0 and bluetooth_tracking_speed = 0.0 and
        radio_tracking_speed = 0.0) or
28                      (wifi_tracking_speed = 0.0 and bluetooth_tracking_speed
                           >= 0.0 and radio_tracking_speed = 0.0) or
29                      (wifi_tracking_speed = 0.0 and bluetooth_tracking_speed =
                            0.0 and radio_tracking_speed >= 0.0);
30  **};
31  annex agree {**
32   --duplicate of xagree annex
33  **};
34 end fork_lift_tracker;
35
36 system implementation fork_lift_tracker.impl
37  annex xagree {**
38   node Counter(init: real, incr: real, reset: bool) returns (count: real);
39   let
40    count = if reset then init else prev(count, init) + incr;
41   tel;
42   eq protocol: real = Counter(0.0, 1.0, prev(protocol = 2.0, false));
43
44   assert wifi_tracking_x = if protocol = 0.0 then 1.0 else 0.0;
45   assert wifi_tracking_y = if protocol = 0.0 then 1.0 else 0.0;
46   assert wifi_tracking_error_margin = if protocol = 0.0 then 1.0 else 0.0;
47   assert wifi_tracking_speed = if protocol = 0.0 then 1.0 else 0.0;
48
49   assert bluetooth_tracking_x = if protocol = 1.0 then 1.0 else 0.0;
50   assert bluetooth_tracking_y = if protocol = 1.0 then 1.0 else 0.0;
```

```
51   assert bluetooth_tracking_error_margin = if protocol = 1.0 then 1.0 else
         0.0;
52   assert bluetooth_tracking_speed = if protocol = 1.0 then 1.0 else 0.0;
53
54   assert radio_tracking_x = if protocol = 2.0 then 1.0 else 0.0;
55   assert radio_tracking_y = if protocol = 2.0 then 1.0 else 0.0;
56   assert radio_tracking_error_margin = if protocol = 2.0 then 1.0 else 0.0;
57   assert radio_tracking_speed = if protocol = 2.0 then 1.0 else 0.0;
58  **};
59  annex agree {**
60   --duplicate of xagree annex
61  **};
62 end fork_lift_tracker.impl;
```

Listing 5.26: Tracking System Verification Sample 1

In listing 5.26, note in this sample that XAGREE offers no improvements over traditional AGREE. This is expected as inheritance is not used in this part of the model.

In listing 5.27, we show further assets for verifying the claim that data produced is within the expected range. For this sampling of the verification assets, we will focus on the communication modules which all implement from an abstract interface. We include both the interface and the wireless module.

```
1  abstract communication_module
2   annex xagree {**
3    assume "x is positive": send_x >= 0.0;
4    assume "y is positive": send_y >= 0.0;
5    assume "error_margin is positive": send_error_margin >= 0.0;
6    assume "speed is positive": send_speed >= 0.0;
7
8    guarantee "x is positive": data_send_x >= 0.0;
9    guarantee "y is positive": data_send_y >= 0.0;
10   guarantee "error_margin is positive": data_send_error_margin >= 0.0;
11   guarantee "speed is positive": data_send_speed >= 0.0;
12   guarantee "signal strength between 0 and 100": 0 <= signal_strength and
```

```
        signal_strength <= 100;
 **};
 annex agree {**
  --duplicate of xagree annex
 **};
end communication_module;

abstract implementation communication_module.impl
 annex xagree {**
  assert data_send_x = send_x;
  assert data_send_y = send_y;
  assert data_send_error_margin = send_error_margin;
  assert data_send_speed = send_speed;
  node Counter(init: int, incr: int, reset: bool) returns (count: int);
  let
   count = if reset then init else prev(count, init) + incr;
  tel;
  assert signal_strength = Counter(0, 1, prev(signal_strength = 100, false));
 **};
 annex agree {**
  --duplicate of xagree annex
 **};
end communication_module.impl;

device wireless_module extends communication_module
 annex xagree {** inherit; **};
 annex agree {**
  assume "x is positive": send_x >= 0.0;
  assume "y is positive": send_y >= 0.0;
  assume "error_margin is positive": send_error_margin >= 0.0;
  assume "speed is positive": send_speed >= 0.0;

  guarantee "x is positive": data_send_x >= 0.0;
```

```
45   guarantee "y is positive": data_send_y >= 0.0;
46   guarantee "error_margin is positive": data_send_error_margin >= 0.0;
47   guarantee "speed is positive": data_send_speed >= 0.0;
48   guarantee "signal strength between 0 and 100": 0 <= signal_strength and
         signal_strength <= 100;
49  **};
50 end wireless_module;
51
52 device implementation wireless_module.impl extends communication_module.impl
53  annex xagree {** inherit; **};
54  annex agree {**
55   assert data_send_x = send_x;
56   assert data_send_y = send_y;
57   assert data_send_error_margin = send_error_margin;
58   assert data_send_speed = send_speed;
59   node Counter(init: int, incr: int, reset: bool) returns (count: int);
60   let
61    count = if reset then init else prev(count, init) + incr;
62   tel;
63   assert signal_strength = Counter(0, 1, prev(signal_strength = 100, false));
64  **};
65 end wireless_module.impl;
```

Listing 5.27: Tracking System Verification Sample 2

Note, in listing 5.27, that XAGREE does not improve on the specification of the abstract interface, but for the communication modules which inherit from the abstract interface, XAGREE reduces their verification assets down to a single line.

In this case, XAGREE reduced the amount of duplication necessary to fully verify the model. It also reduced the overall length of the verification assets. We will see in the next examples how XAGREE can improve cyclomatic complexity as well as duplication / size.

Table 5.2: Nutrition System AGREE Summary

Component Name	Total Lines	Duplicated Lines	Cyclomatic Complexity
meal_planner	4	4	1
meal_planner.impl	12	12	1
allergy_meal_planner	8	4	1
allergy_meal_planner.impl	12	8	5
preference_meal_planner	8	4	1
preference_meal_planner.impl	12	8	5
preference_allergy_meal_planner	12	4	1
preference_allergy_meal_planner.impl	12	8	5
divider_and_storer	9	9	1
divider_and_storer.impl	9	9	4
community_gatherer	20	20	1
community_gatherer.impl	4	4	1
community_redistributor	20	20	1
community_store_for_later_redistributor	20	20	1
national_gatherer	20	20	1
national_gatherer.impl	4	4	1
national_redistributor	20	20	1
national_store_for_later_redistributor	20	20	1
Average	12.55555556	11	1.833333333

5.3.3 Dynamic Adaptive Nutrition System Verification / Validation

For the nutrition system, we will list a summary of the measured qualities of the verification assets using both AGREE (table 5.2) and XAGREE (table 5.3). Note that any component having equal verification asset qualities with both AGREE and XAGREE are not listed. A discussion of the verification assets and the differences between AGREE and XAGREE is also included.

5.3.3.1 Nutrition System Verification / Validation Discussion

From the two tables summarizing components with different line counts, cyclomatic complexity and / or duplication, we can see that XAGREE, on average, decreases some or all of the qualities for most components. It should be noted, as in the forklift tracking system, all of the components listed use inheritance. As can be expected, there is no demonstrable benefit to using XAGREE over AGREE in models not incorporating inheritance.

The primary goals needed to verify the nutrition system were to ensure 1) food production and food needs values were always positive, 2) that the amount of food received by a family was not more than the family needed, and 3) the amount of food received by a community was not more than the community needed. The last two goals help to ensure fair distribution. It should be noted,

Table 5.3: Nutrition System XAGREE Summary

Component Name	Total Lines	Duplicated Lines	Cyclomatic Complexity
meal_planner	1	0	1
meal_planner.impl	4	0	1
allergy_meal_planner	4	0	1
allergy_meal_planner.impl	4	0	5
preference_meal_planner	4	0	1
preference_meal_planner.impl	4	0	5
preference_allergy_meal_planner	8	0	1
preference_allergy_meal_planner.impl	4	0	5
divider_and_storer	1	0	1
divider_and_storer.impl	1	0	1
community_gatherer	1	0	1
community_gatherer.impl	1	0	1
community_redistributor	1	0	1
community_store_for_later_redistributor	1	0	1
national_gatherer	1	0	1
national_gatherer.impl	1	0	1
national_redistributor	1	9	1
national_store_for_later_redistributor	1	0	1
Average	2.388888889	0.5	1.666666667

however, that it is completely acceptable for food amounts given to be less than needed indicating a food shortage. The verification assets throughout the entire model account for the possibility. It is also possible for food production to exceed needs. In this case, food will be temporarily stored until its expiration date nears. Food whose expiration date has passed is discarded.

5.3.4 Dynamic Adaptive Underwater Vehicle Verification / Validation

For the underwater vehicle system, we will list a summary of the measured qualities of the verification assets of the individual components using both AGREE (table 5.4) and XAGREE (table 5.5). Note that any components having equal verification asset qualities with both AGREE and XAGREE are not listed. A discussion of the verification assets and the differences between AGREE and XAGREE is also included.

5.3.4.1 Underwater Vehicle Verification / Validation Discussion

From the two tables summarizing components with different line counts, cyclomatic complexity and / or duplication, we can see that XAGREE, on average, decreases some or all of the counts for most components. It should be noted, as in the forklift tracking system and the nutrition

Table 5.4: Underwater Vehicle AGREE Summary

Component Name	Total Lines	Duplicated Lines	Cyclomatic Complexity
sensor	4	0	1
sensor.impl	3	0	1
water_temperature_sensor	4	4	1
water_temperature_sensor.impl	3	2	1
water_salinity_sensor	4	4	1
water_salinity_sensor.impl	3	2	1
radar_sensor	4	4	1
radar_sensor.impl	3	2	1
pressure_sensor	4	4	1
pressure_sensor.impl	3	2	1
oxygen_sensor	4	4	1
oxygen_sensor.impl	3	2	1
carbon_dioxide_sensor	4	4	1
carbon_dioxide_sensor.impl	3	2	1
carbon_monoxide_sensor	4	4	1
carbon_monoxide_sensor.impl	3	2	1
normal_mission_controller	61	61	1
normal_mission_controller.impl	20	20	1
degraded_mission_controller	61	60	1
degraded_mission_controller.impl	20	20	1
normal_power_manager	20	20	1
normal_power_manager.impl	21	0	15
degraded_power_manager	20	20	1
degraded_power_manager.impl	21	0	15
critical_power_manager	20	20	1
critical_power_manager.impl	21	0	15
exterior_sensor_set	16	16	1
interior_sensor_set	16	16	1
exterior_aggregator	64	64	1
Average	15.06896552	12.37931034	2.448275862

Table 5.5: Underwater Vehicle XAGREE Summary

Component Name	Total Lines	Duplicated Lines	Cyclomatic Complexity
sensor	3	0	1
sensor.impl	4	0	1
water_temperature_sensor	1	0	1
water_temperature_sensor.impl	1	0	1
water_salinity_sensor	1	0	1
water_salinity_sensor.impl	1	0	1
radar_sensor	1	0	1
radar_sensor.impl	1	0	1
pressure_sensor	1	0	1
pressure_sensor.impl	1	0	1
oxygen_sensor	1	0	1
oxygen_sensor.impl	1	0	1
carbon_dioxide_sensor	1	0	1
carbon_dioxide_sensor.impl	1	0	1
carbon_monoxide_sensor	1	0	1
carbon_monoxide_sensor.impl	1	0	1
normal_mission_controller	1	0	1
normal_mission_controller.impl	1	0	1
degraded_mission_controller	1	0	1
degraded_mission_controller.impl	1	0	1
normal_power_manager	1	0	1
normal_power_manager.impl	1	0	1
degraded_power_manager	1	0	1
degraded_power_manager.impl	1	0	1
critical_power_manager	1	0	1
critical_power_manager.impl	1	0	1
exterior_sensor_set	1	0	1
interior_sensor_set	1	0	1
exterior_aggregator	1	0	1
Average	1.172413793	0	1

system, all of the components listed use inheritance. As can be expected, there is no demonstrable benefit to using XAGREE over AGREE in models not incorporating inheritance.

Note, however, that despite a general decrease across all three measurements, it is possible for individual components to fare worse when using XAGREE. This is particularly true if abstract components needed to be added to account for inheritance of verification assets, as was true with *sensor.impl*.

The primary goals needed to verify the underwater vehicle system were to ensure that 1) sensors correctly transmitted sensor data as well as error rates, 2) the energy source could disable and enable sensors, 3) sensors correctly responded to being disabled and 4) the mission controller correctly responded to having multiple sensors with high error rates or multiple offline sensors. The last goal primarily revolved around checking the ability to swap between mission controllers based on sensor degradation and unavailability.

5.3.4.2 Summary

In this evaluation, we have evaluated the effectiveness of XAGREE to produce verification assets that are equal or better in terms of complexity and size with regard to traditional AGREE. The purpose of this evaluation is to show the improvement that XAGREE makes on traditional AGREE.

To demonstrate the stated improvements, we have taken three models and annotated them with assume / guarantee assets using both AGREE and XAGREE. Several measures of size and complexity were then taken of each model and reported. In general, XAGREE's assets had less duplication and lower complexity than those of AGREE.

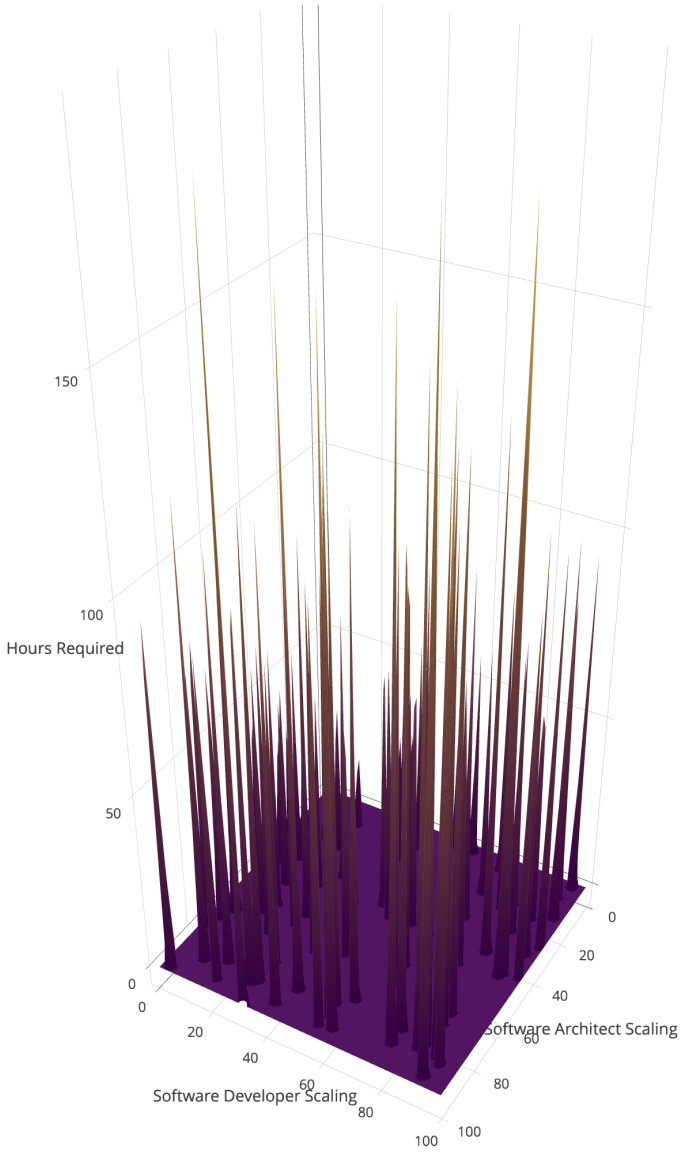

Figure 5.18: Tracking System Cost Sensitivity Analysis - 100 (https://plot.ly/~bulletshot60/6)

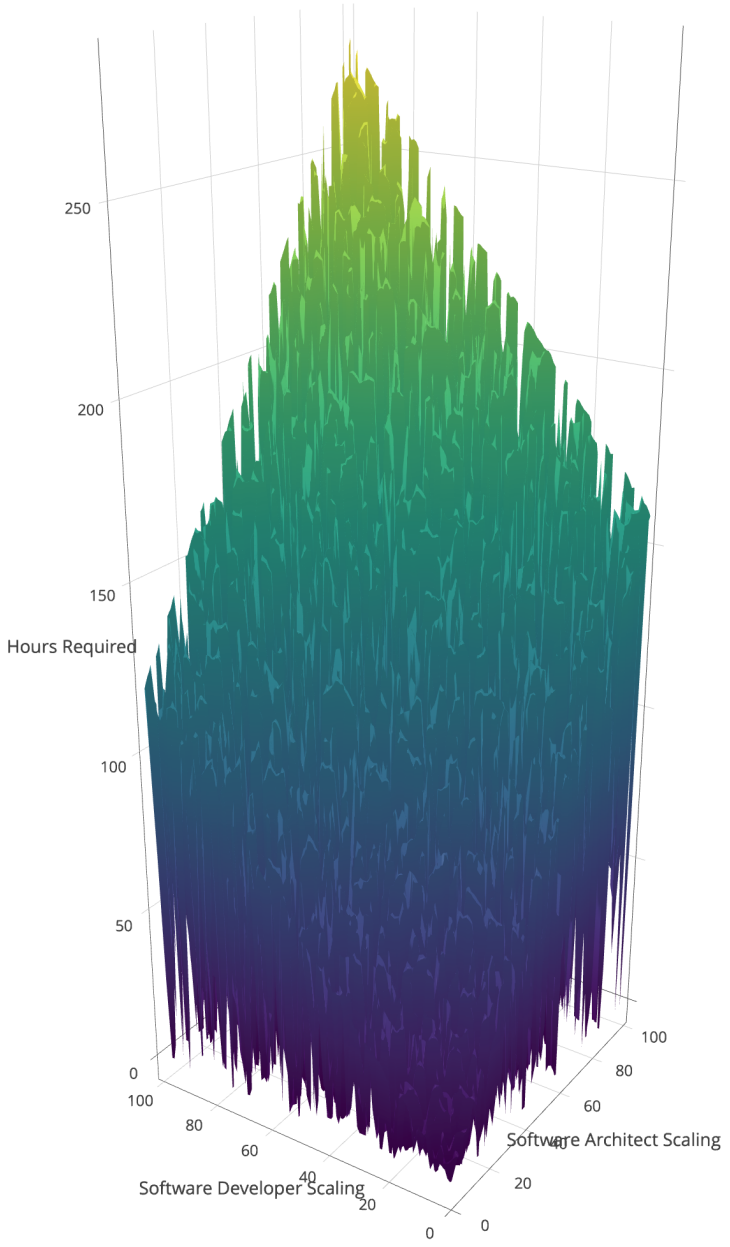

Figure 5.19: Tracking System Sensitivity Analysis - 10000 (https://plot.ly/~bulletshot60/8)

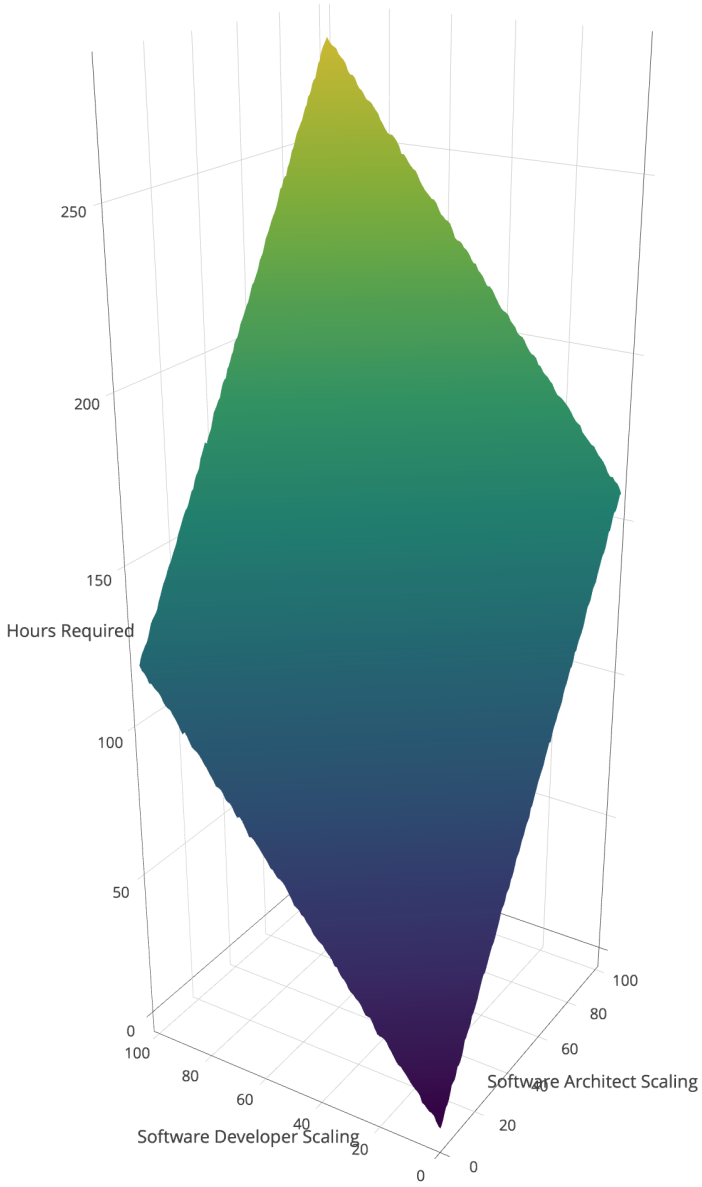

Figure 5.20: Tracking System Sensitivity Analysis - 1000000 (https://plot.ly/~bulletshot60/10)

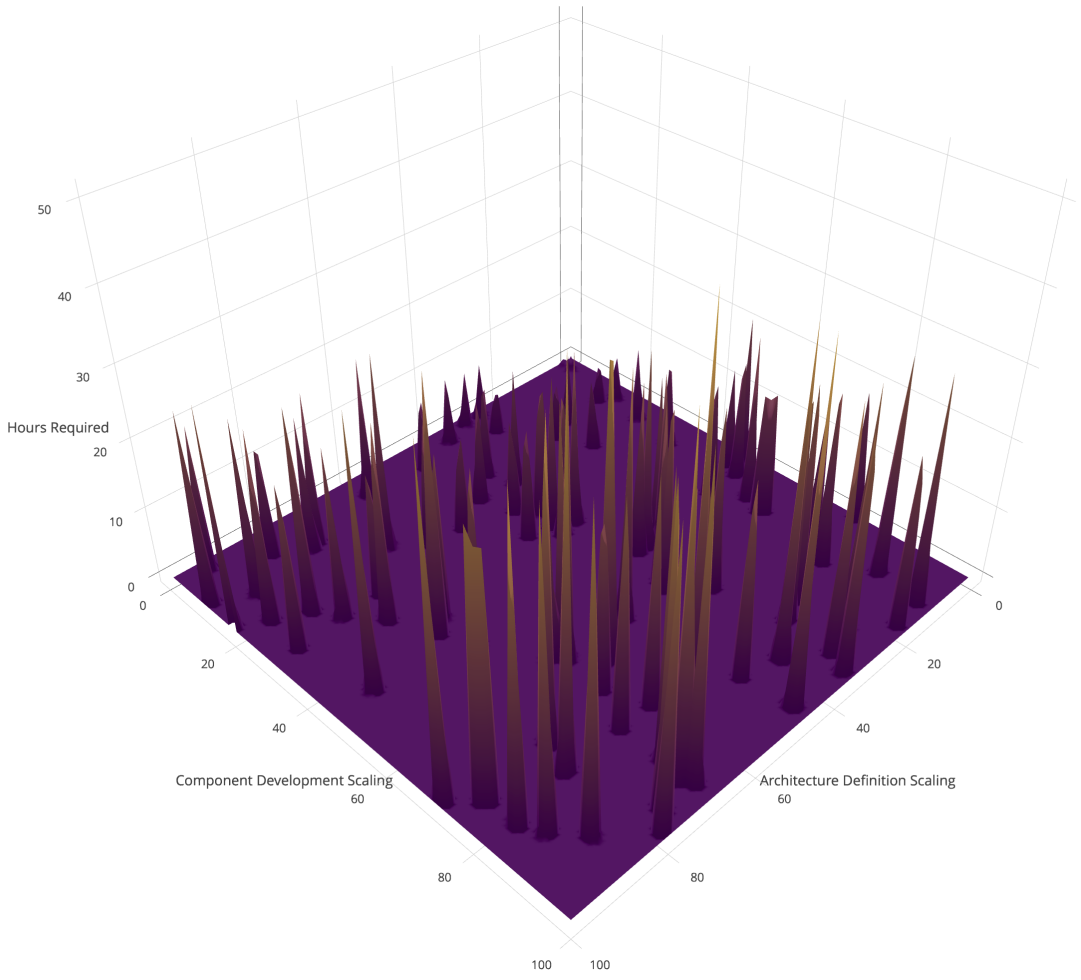

Figure 5.21: Nutrition System Sensitivity Analysis 1 - 100 (https://plot.ly/~bulletshot60/12)

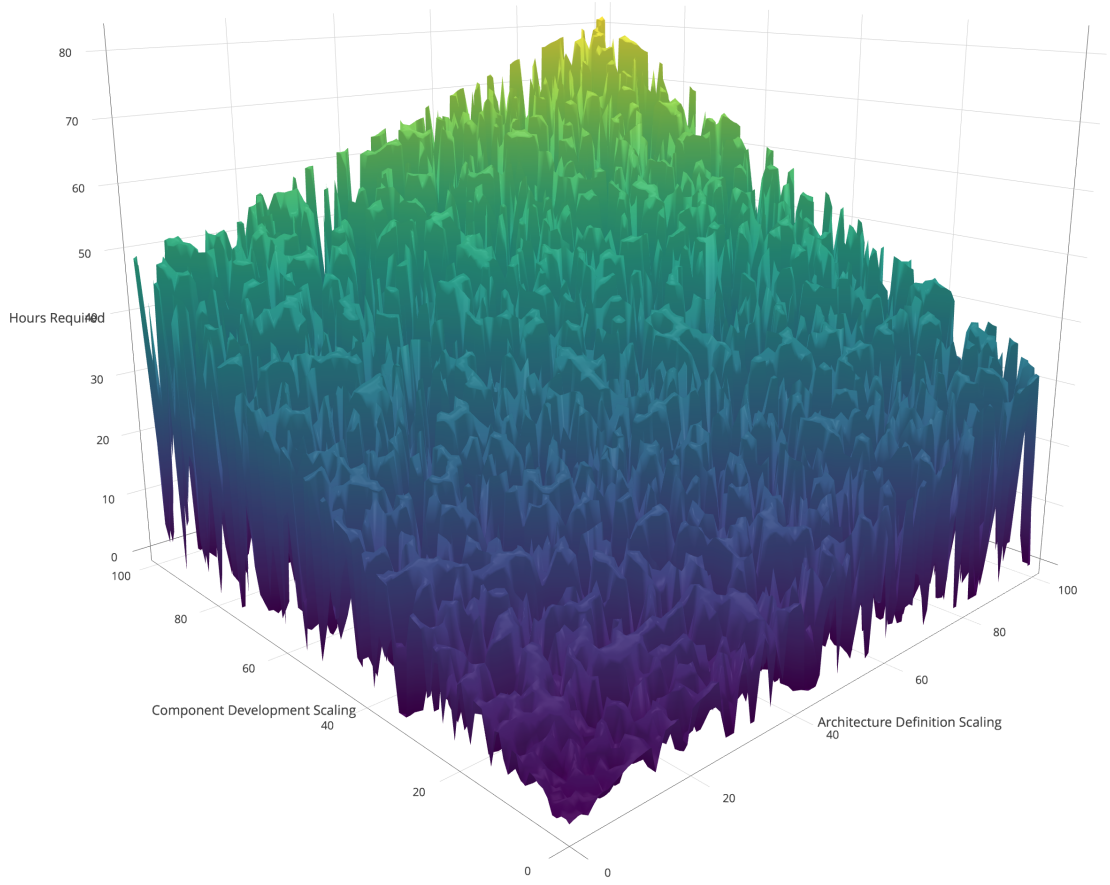

Figure 5.22: Nutrition System Sensitivity Analysis 1 - 10000 (https://plot.ly/~bulletshot60/14)

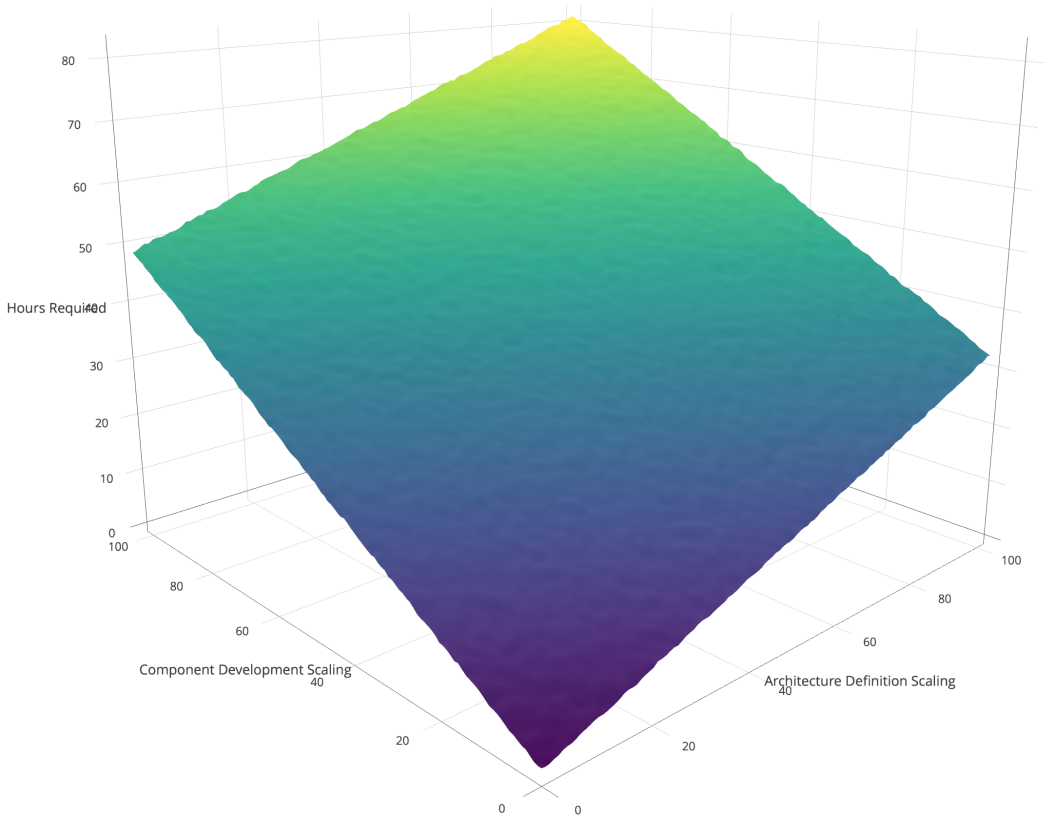

Figure 5.23: Nutrition System Sensitivity Analysis 1 - 1000000 (https://plot.ly/~bulletshot60/16)

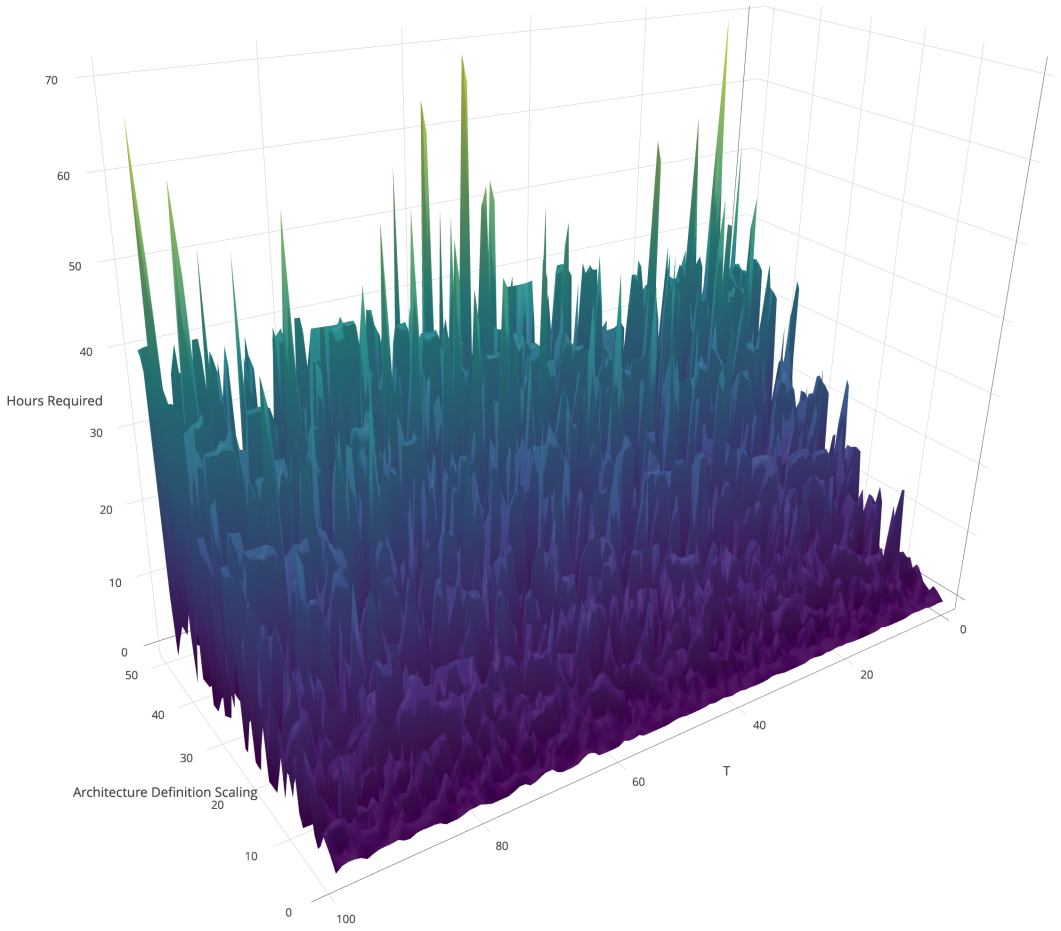

Figure 5.24: Nutrition System Sensitivity Analysis 2 - 100 (https://plot.ly/~bulletshot60/18)

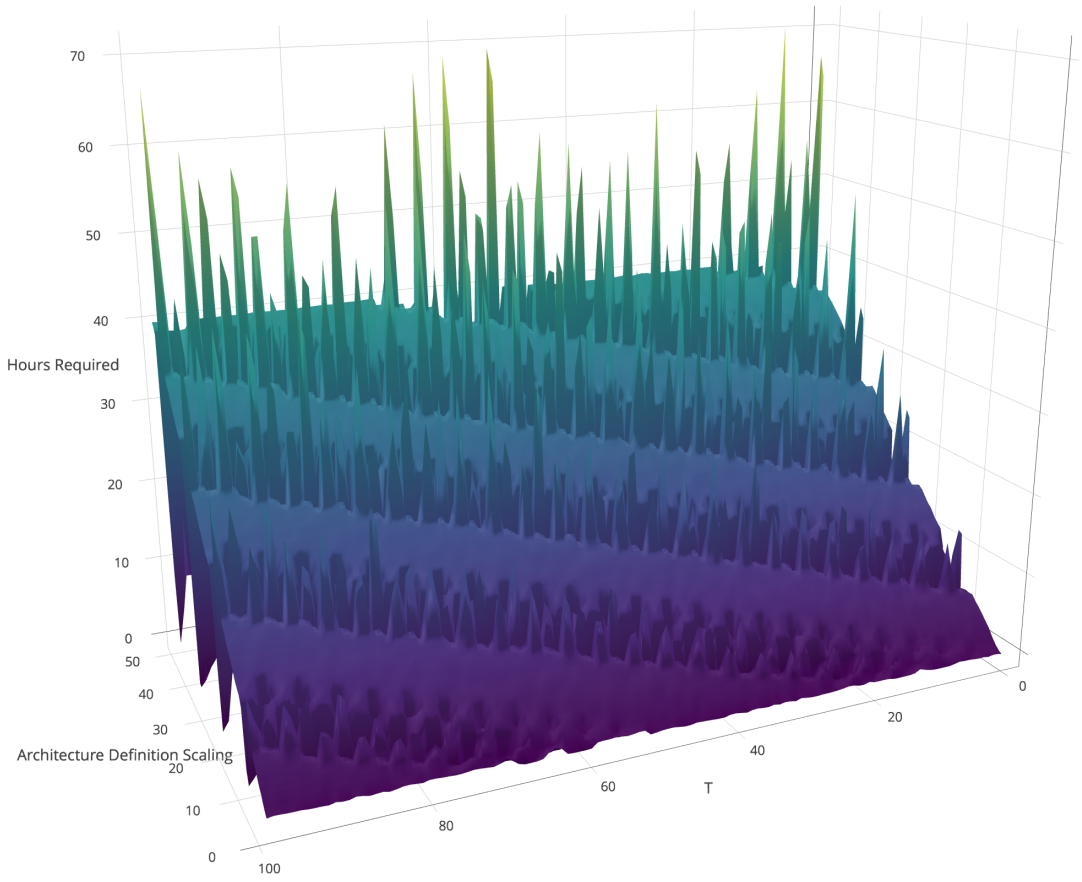

Figure 5.25: Nutrition System Sensitivity Analysis 2 - 10000 (https://plot.ly/~bulletshot60/20)

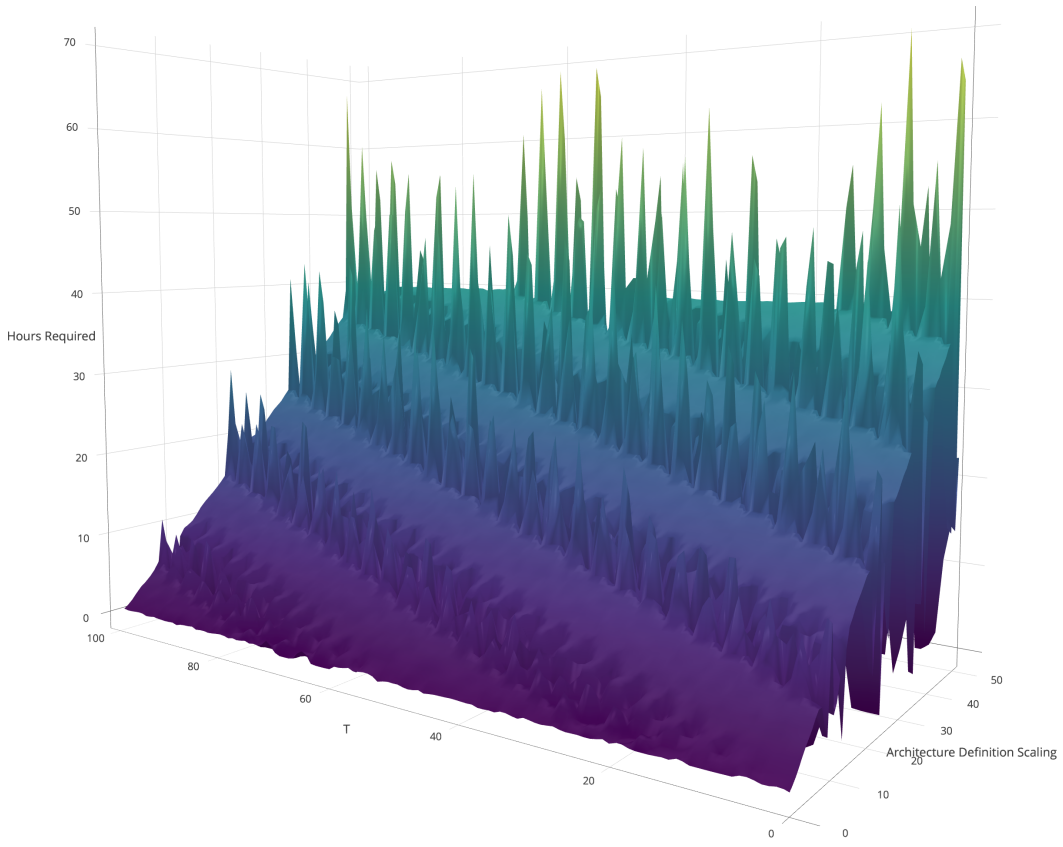

Figure 5.26: Nutrition System Sensitivity Analysis 2 - 1000000 (https://plot.ly/~bulletshot60/22)

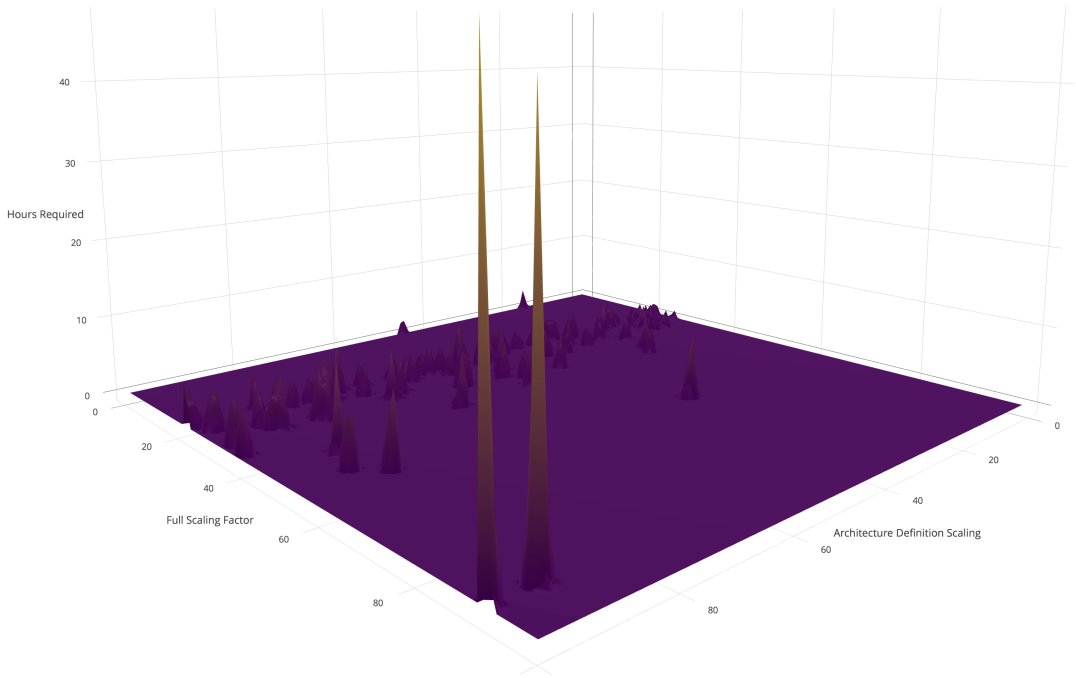

Figure 5.27: Underwater Vehicle System Sensitivity Analysis - 100 (https://plot.ly/~bulletshot60/31)

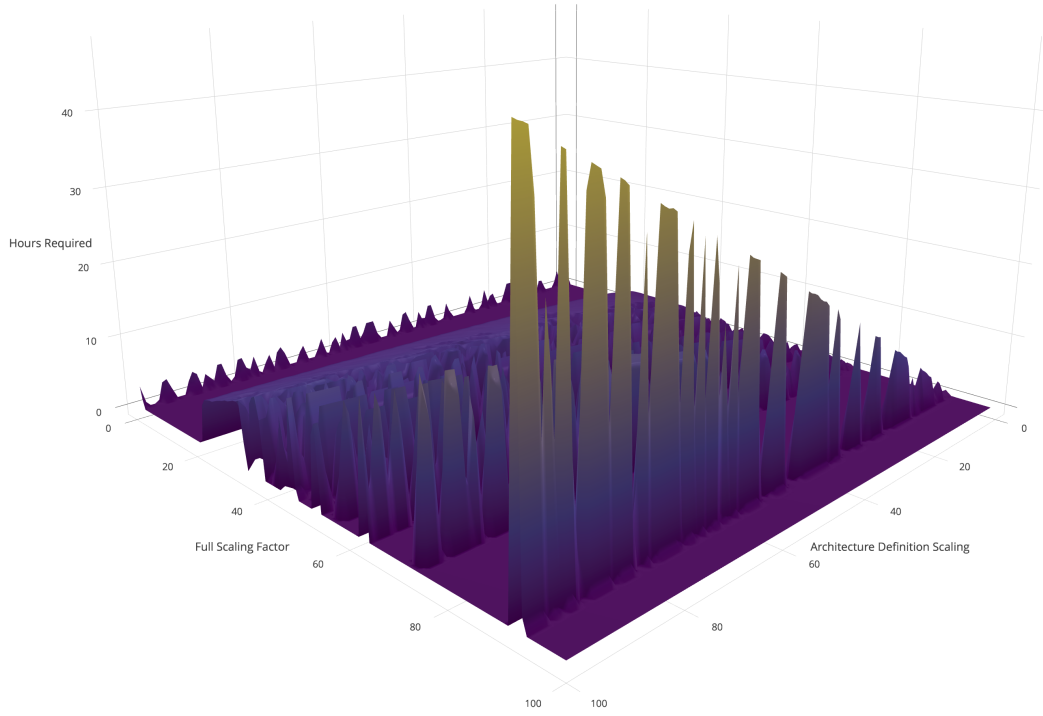

Figure 5.28: Underwater Vehicle System Sensitivity Analysis - 10000 (https://plot.ly/~bulletshot60/29)

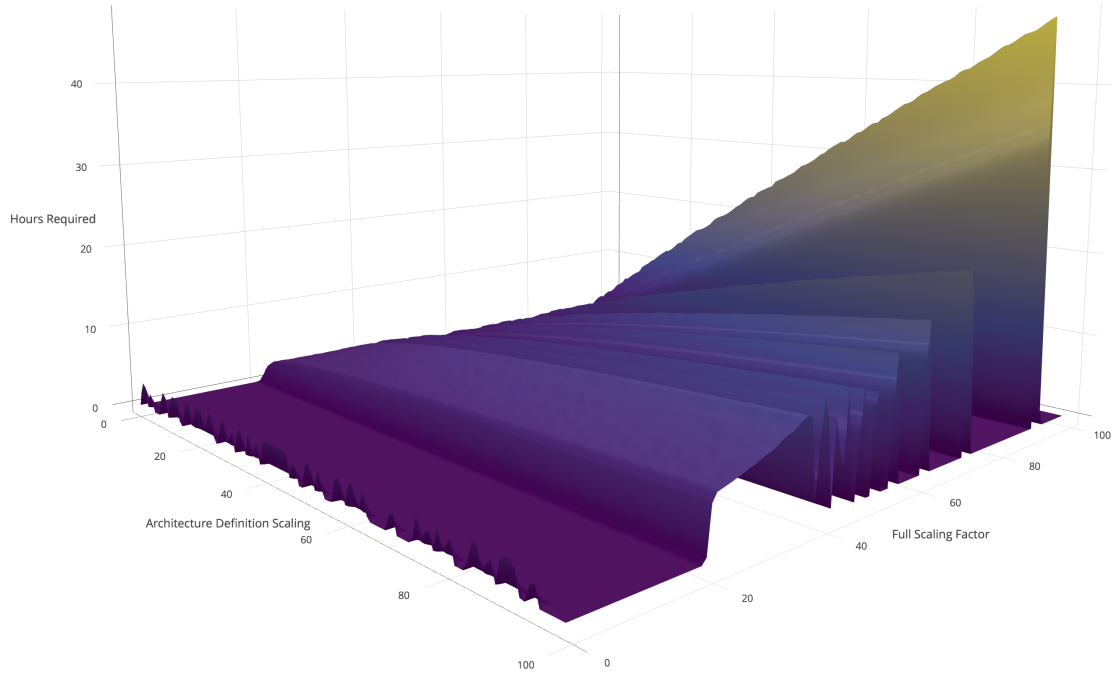

Figure 5.29: Underwater Vehicle System Sensitivity Analysis - 1000000 (https://plot.ly/~bulletshot60/27)

Chapter 6

Discussion

In this section, we will first discuss the limitations of the techniques used to evaluate the tools developed as part of this research. We will justify the choices made during the evaluation, and we will also discuss the contributions of both XAGREE and SIMDASE.

6.1 Justification of & Limitations of Evaluation

In the first evaluation (for claim C1), we chose to leave the cost amounts in hours per employee. The rationale behind this choice was twofold. First, any average hourly amount chosen for a particular type of employee will eventually become outdated dampening the relevance of the evaluation results. Second, it is straightforward to compute the dollar cost once an average salary amount is chosen for each employee type.

Additionally, all analysis for the claim C1's evaluation was done within the research group. For that reason, most planning costs are not represented within any of the evaluations, only construction costs (architecture definition, development, testing). The reason for this choice is due to the author of this work having 8 years of practical experience developing software but very little practical experience planning software. In order to avoid having unreasonable or unfeasible planning costs as part of the examples, the decision was made to limit cost estimation to the construction phase only.

In the second evaluation (for claim C2), it has been noted that the sensitivity analysis might seem an unreasonable evaluation considering that the majority of the SIMDASE formulas

are nothing more than recursive addition. However, the sensitivity analysis is actually focused on a single term rather than the entirety of the framework. $C_{static}(v))^{s(i)}$ represents the cost of a single variant. However, both portions of this equation are user-definable. Additionally, due to the recursive nature of the formulas, this equation can be evaluated multiple times, and this formula is evaluated in the context of an architecture and calendar time, both of which have an effect on the outcome of an evaluation. For example, costs may change over time, or the style of architecture chosen might cause the cost to fluctuate. For these reasons, we decided that a sensitivity analysis that evaluated the architecture style, time and variant costs was necessary.

Finally, in the third evaluation (claim C3), we choose three simple measures of complexity: cyclomatic complexity, line count and amount of duplication. These three measures were chosen because they are objective and not limited by author subjectivity like measures of reusability. The measures also not do require averaging the subjective estimation of a group of people, such as most measures of maintainability. Additionally, our initial findings suggested that, in general, asset complexity and duplication are reduced when inheritance is used. We choose to use measures that reflect and measure those findings. We realize that cases exist where this is not true, thus we provide a way to disable inheritance within XAGREE.

6.2 SIMDASE Field Contributions

In our evaluation of claim C1, we demonstrate that SIMDASE is not only flexible enough to support both common and uncommon development scenarios, we have also demonstrated that it is quite robust and the formulas can handle complex user-defined formulas reliably. However, SIMDASE's primary contributions do not lie just in its flexible syntax and reliability. SIMDASE makes three primary contributions / advancements to the field of DSPLs and cost estimation.

First, with respect to cost estimation, SIMDASE takes a unique approach in representing cost. SIMDASE represents cost as the number of employees required to complete a task and the number of hours required of those employees rather than using a fixed number. This matches the way that many managers, in our experience, choose to calculate cost. This representation also yields additional information that traditional cost analysis methods do not natively yield. This includes the total number of employees that will be required over the course of a project as well as the total number of man-hours that will be contributed. While SIMDASE is not the first cost estimation

method to use an hourly representation, it is the first method within the AADL community to do so. It is also one of the first to use this as the sole representation method rather than "tacking it on" as an additional option.

Second, also with respect to cost information, SIMDASE places the cost information into the architecture as an annotation. This allows project managers to track cost in the same manner that they track the attributes of the architecture of a project. Having the cost tied to the architecture also means that as requirements change, morphing the system architecture, it is less likely that the cost calculations will become out of sync with the architecture. This differentiates SIMDASE from other model-based methods as the model used is an accepted standards-based architecture description language rather than a custom language used only with the cost description language.

Finally, with respect to the DSPL community, SIMDASE provides an objective method of choosing between the designs of a DSPL. This will hopefully allow experts in the domain to quantify why certain DSPL designs are better than others as well as yield additional insight into how much additional cost adopting a DPSL design over a SPL design incurs. It also allows developers as well as managers to calculate the expected return on investment that can be expected from a given DSPL design. This will hopefully help convince potential industrial partners that DSPLs are worth the risk and extra development effort.

6.3 XAGREE Field Contributions

XAGREE makes two main contributions to the field of verification with AADL. First, it introduces the idea of polymorphic subtyping [20], but only with simple data types. AGREE does not support complex data types, nor does XAGREE. This is a planned feature for future work in XAGREE that will further serve to differentiate XAGREE from AGREE (see section 8). Second, XAGREE introduces the idea of inherited verification assets that can be overridden or extended. This allows for duplication of assets to be partially eliminated improving maintainability and complexity. This also has the added benefit of making it easier to add and verify new components provided they extend existing components.

Chapter 7

Related Work

In this section, we will discuss works related to both XAGREE and SIMDASE. We will also discuss work that is loosely related to our DSPL DSS framework, including research that is likely to be incorporated into the framework at a later date.

7.1 Works Related to XAGREE

Compositional assume-guarantee verification is a popular verification technique, and it has been used successfully in many other ecosystems outside of AADL. Some examples of this are [11] and [22]. Our work differs from these groups in where in the development process verification is applied to the system. We apply compositional verification to the architecture during the design phase of the development life cycle, while these other projects apply verification later in the process.

Our work is most similar to the work performed by the following groups, particularly [39] and [56], both of which used AADL. Additional architecture-based techniques exist, such as [29], [15] and [27]. Our work differs from these groups in that we are explicitly focused on allowing the verification assets to be reused in the same manner as SPL assets, exploiting the inheritance features of the AADL language.

7.2 Works Related to SIMDASE

SIMDASE is a model-based software cost estimation framework. There are many other model-based cost estimation techniques, for example, COCOMO II [9], COCOMO [8], SLIM [43, 42], Checkpoint [10], SEER [32], COPLIMO [7] and SIMPLE [14]. We differ from each of these primarily in focus. SIMDASE is focused on DSPLs and SPLs, estimating the costs associated with variability and dynamic adaptive components which are unique to DSPLs. Additionally, SIMDASE, like SIMPLE, works by leveraging managerial knowledge to produce an estimate instead of using data from other projects, such as COCOMO II.

Our framework is most similar to SIMPLE, the framework from which SIMDASE is derived. We differ from SIMPLE in several key respects. First, we require an architectural model for estimations whereas SIMPLE can be executed without a model. However, the presence of a model allows SIMDASE to produce more accurate estimates. Second, SIMDASE allows scaling of the cost of estimation with the depth at which an entity occurs within the architecture. Finally, SIMDASE considers adaptive and non-adaptive variation points as separately estimated entities.

7.3 Additional Related Works

Our work in creating a framework to decide between designs of a DSPL is, to the best of our knowledge, not being undertaken by any group. However, certain parts of the process we have outlined, including tracking of quality attributes are similar to many other groups. Our work is heavily based on two (2012 and 2013) literature surveys [51, 50] which track quality attributes that are commonly measured in dynamic adaptive systems and how dynamic adaptivity changed these attributes. Our work differs in that we attempt to make the process more rigorous rather than simply tracking dynamic adaptive projects through to completion.

Other research groups have already produced results from the inclusion of quality attribute measurement in a decision support system [41, 30]. Inclusion of quality attribute estimation into our DSS would allow for a more detailed analysis of the appropriateness of a design basing the comparison of designs on multiple factors rather than just cost and return on investment alone. As work on our DSS continues, these works and others like them will need to be reviewed to see what can be reused.

Chapter 8

Future Work

As mentioned in section 4, and shown in figure 8.1, not all of the framework required for selecting between DSPL designs has been completed in this work. The parts of the framework shown in green are mostly completed and have been evaluated as part of the work for this dissertation. The parts shown in yellow are partially complete but require a great deal more work. While XAGREE (Architecture Validation & Verification) and SIMDASE (Cost Estimation) are mostly complete, there is still additional work that can be completed in these areas.

8.1 Future Work in Quality Attribute Estimation

Estimation or measurement of quality attributes is an important step within our DSS, however, at the current time, we have only completed work determining how specific quality attributes change when dynamic adaptivity is introduced into the model. Many of the techniques for measuring these quality attributes work well for architectural models that do not incorporate dynamic adaptivity, but they are less than optimal for architecture models that do include dynamic adaptivity.

Future work in this area will include evaluating existing measurement techniques for the quality attributes that are commonly affected by dynamic adaptivity's presence. Changes or improvements that can be made to these techniques to make them more efficient for architectures which include variability will be explored. Also, determining which quality attribute measurements can be calculated from within SIMDASE will be determined.

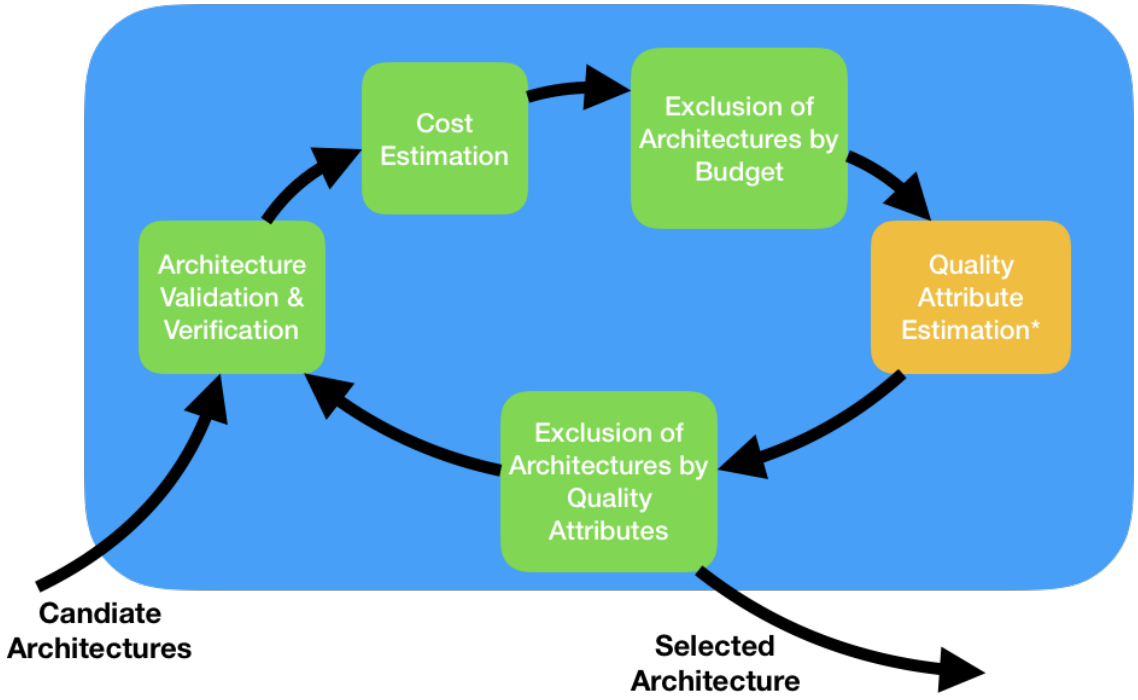

Figure 8.1: Decision Support System Outline

8.2 Future Work in SIMDASE

SIMDASE's support for cost estimation of architectural models including variability and its ability to infer information from the architecture's structure are powerful. However, due to the exacting nature of AADL specifications, SIMDASE could infer additional information from the model that would prove useful in cost estimation. For example, it is possible in AADL through the use of additional annexes to represent the complexity of components. Using this information, the cost of a component could be scaled based on the component's complexity which would allow some degree of automatic cost calculation. Additionally, inheritance support in SIMDASE would be useful as would the idea of packages (costs that are common to multiple components packaged into a reusable asset).

8.3 Future Work in XAGREE

At the current moment, XAGREE supports subtype polymorphism [20] and inheritance between XAGREE instantiations in related components for non-timing related statements. Tra-

ditional AGREE also has excellent support for validating timing and event occurrence, however, XAGREE does not add inheritance or subtype polymorphism support in these statements. Adding support for inheritance for these areas of AGREE would be the primary additional work left in this area. However, there are other areas that could be considered. One such area would be support for complex data types and port groupings.

Chapter 9

Conclusion

In the proposal for dissertation, we promised to provide cost estimation and improved verification / validation tooling for DASs. In this dissertation, we have presented the outline of a new framework for deciding between designs of a DSPL. We have also presented two specific components of this framework: SIMDASE (section 4.2) and XAGREE (section 3). SIMDASE is a cost estimation tool, based on SIMPLE, that provides for architecture-led cost estimation of SPLs and DSPLs, or any architecture that can be expressed as a SPL or DSPL, such as DASs. XAGREE is an extension of the AADL tool AGREE adding inheritance support to the language. This addition allows XAGREE to produce more maintainable and less complex verification assets than AGREE for architectures that incorporate inheritance such as SPLs and DSPLs.

As part of this dissertation work, we have provided an analysis (section 5) of XAGREE's improvements on AGREE for architectural models using inheritance. We have also provided an analysis of SIMDASE's flexibility as well as its ability to robustly evaluate user input, including input containing sinusoidal and logarithmic functions with respect to time and architecture description.

Due to the work presented in this dissertation, it is now possible to more fully use the language capabilities of AADL during the verification / validation of systems utilizing inheritance. It is also possible to annotate the architecture of a SPL or DSPL with cost information, analyzing that cost information on demand. Both of these advancements to AADL and its annexes will provide benefit for the DAS / DSPL communities which heavily use AADL architectures having inheritance.

Appendices

Appendix A List of Abbreviations

- AADL - Architecture Analysis & Design Language
- AGREE - Assume Guarantee REasoning Environment
- BLE - Bluetooth Low Enegery
- CPS - Cyber Physical System
- DAS - Dynamic Adaptive System
- DSPL - Dynamic Software Product Line
- DSS - Decision Support System
- GPS - Global Positioning System
- HTML - HyperText Markup Language
- ISO - International Standards Organization
- ITS - Intelligent Transportation System
- MAPE - Monitor Analyze Plan Execute
- OSATE - Open Source AADL Tool Environment
- PDF - Portable Document Format
- RFID - Radio Frequency IDentifier
- SAE - Society of Automotive Engineers
- SDLC - Software Development Life Cycle
- SIMDASE - Structured Intuitive Model for Dynamic Adaptive System Economics
- SIMPLE - Structured Intuitive Model for Product Line Economics
- SMT - Satisfiability Modulo Theorem
- SPL - Software Product Line
- XAGREE - eXtended Assume Guarantee REasoning Environment

Appendix B ISO 25010 Quality Attributes

- **Functional Suitability** - degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions
  - *Functional Completeness* - degree to which the set of functions covers all the specified tasks and user objectives
  - *Functional Correctness* - degree to which a product or system provides the correct results with the needed degree of precision
  - *Functional Appropriateness* - degree to which the functions facilitate the accomplishment of specified tasks and objectives
- **Performance Efficiency** - performance relative to the amount of resources used under stated conditions
  - *Time Behaviour* - degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements
  - *Resource Utilization* - degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements
  - *Capacity* - degree to which the maximum limits of a product or system parameter meet requirements
- **Compatibility** - degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment
  - *Co-existence* - degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product
  - *Interoperability* - degree to which two or more systems, products or components can exchange information and use the information that has been exchanged
- **Usability** - degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use

  - *Appropriateness Recognizability* - degree to which users can recognize whether a product or system is appropriate for their needs
  - *Learnability* - degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system with effectiveness, efficiency, freedom from risk and satisfaction in a specified context of use
  - *Operability* - degree to which a product or system has attributes that make it easy to operate and control
  - *User Error Protection* - degree to which a system protects users against making errors
  - *User Interface Aesthetics* - degree to which a user interface enables pleasing and satisfying interaction for the user
  - *Accessibility* - degree to which a product or system can be used by people with the widest range of characteristics and capabilities to achieve a specified goal in a specified context of use

- **Reliability** - degree to which a system, product or component performs specified functions under specified conditions for a specified period of time
  - *Maturity* - degree to which a system, product or component meets needs for reliability under normal operation
  - *Availability* - degree to which a system, product or component is operational and accessible when required for use
  - *Fault Tolerance* - degree to which a system, product or component operates as intended despite the presence of hardware or software faults
  - *Recoverability* - degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system

- **Security** - degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
  - *Confidentiality* - degree to which a product or system ensures that data are accessible only to those authorized to have access

  - *Integrity* - degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data
  - *Non-repudiation* - degree to which actions or events can be proven to have taken place, so that the events or actions cannot be repudiated later
  - *Accountability* - degree to which the actions of an entity can be traced uniquely to the entity
  - *Authenticity* - degree to which the identity of a subject or resource can be proved to be the one claimed

- **Maintainability** - degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
  - *Modularity* - degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components
  - *Reusability* - degree to which an asset can be used in more than one system, or in building other assets
  - *Analysability* - degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified
  - *Modifiability* - degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality
  - *Testability* - degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met

- **Portability** - degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another
  - *Adaptability* - degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments

- *Installability* - degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment
- *Replaceability* - degree to which a product can replace another specified software product for the same purpose in the same environment

Appendix C EMV2 Error Ontology

- ServiceError
  - ItemOmission
  - ServiceOmission
  - SequenceOmission
    - TransientServiceOmission
    - LateServiceStart
    - EarlyServiceTermination
    - BoundedOmissionInterval
  - ItemCommission
  - ServiceCommission
  - SequenceCommission
    - EarlyServiceStart
    - LateServiceTermination
- TimingRelatedError
  - ItemTimingError
    - EarlyDelivery
    - LateDelivery
  - SequenceTimingError
    - HighRate
    - LowRate
    - RateJitter
  - ServiceTimingError
    - DelayedService
    - EarlyService
- ValueRelatedError

- ItemValueError
    - UndetectableValueError
    - DetectableValueError
        - OutOfRange
        - BelowRange
        - AboveRange
        - OutOfBounds
- SequenceValueError
    - BoundedValueChange
    - StuckValue
    - OutOfOrder
- ServiceValueError
    - OutOfCalibration

- ReplicationError
    - AsymmetricReplicatesError
        - AsymmetricValue
            - AsymmetricApproximateValue
            - AsymmetricExactValue
        - AsymmetricTiming
        - AsymmetricOmission
            - AsymmetricItemOmission
            - AsymmetricServiceOmission
    - SymmetricReplicatesError
        - SymmetricValue
            - SymmetricApproximateValue
            - SymmetricExactValue
        - SymmetricTiming

        * SymmetricOmission
            · SymmetricItemOmission
            · SymmetricServiceOmission

- ConcurrencyError
    - RaceCondition
    - ReadWriteRace
    - WriteWriteRace
    - MutExError
    - Deadlock
    - Starvation

Appendix D SIMDASE Annex Language Reference

D.1 Active Variants Statement

D.1.1 Format

```
active_variants => {
    <IDENTIFIER_1> : [<VARIANT_1>, <VARIANT_2>, ... <VARIANT_N>],
    <IDENTIFIER_2> : [<VARIANT_1>, <VARIANT_2>, ... <VARIANT_N>],
    ...
    <IDENTIFIER_N> : [<VARIANT_1>, <VARIANT_2>, ... <VARIANT_N>],
};
```

D.1.2 Description

The active variants statement allows you to specify the variants are to be part of the current cost estimation for individual subcomponents (the current component included). The identifier for each substatement is a fully qualified component name (¡package_name¿::¡component_classifier¿.¡implementation_name¿) and the variant name references an identifier from that components variant list.

D.1.3 Example

```
annex simdase {**
    variants => {
     "wifi_enabled":{
            "wireless_module","rfid_reader","send_tracking_info_via_wifi",
            "protocol_management_process"
        };
        "ble_enabled":{
            "bluetooth_module","rfid_reader","send_tracking_info_via_ble",
            "protocol_management_process"
        };
    };
    active_variants => {
        "fork_lift_tracker.impl": ["wifi_enabled", "ble_enabled"];
    };
```

```
};
```

D.2 Cost Properties Statement

D.2.1 Format

```
cost => {
    <COST_CATEGORY_1> => {
        <EMPLOYEE_BASED_COST_1>,
        <EMPLOYEE_BASED_COST_2>,
        ...
        <EMPLOYEE_BASED_COST_N>,
        <FIXED_COST_1>,
        <FIXED_COST_2>,
        ...
        <FIXED_COST_N>
    };
    <COST_CATEGORY_2> => {
        <EMPLOYEE_BASED_COST_1>,
        <EMPLOYEE_BASED_COST_2>,
        ...
        <EMPLOYEE_BASED_COST_N>,
        <FIXED_COST_1>,
        <FIXED_COST_2>,
        ...
        <FIXED_COST_N>
    };
    ...
    <COST_CATEGORY_N> => {
        <EMPLOYEE_BASED_COST_1>,
        <EMPLOYEE_BASED_COST_2>,
        ...
        <EMPLOYEE_BASED_COST_N>,
        <FIXED_COST_1>,
```

```
29            <FIXED_COST_2>,
30            ...
31            <FIXED_COST_N>
32        };
33 };
```

D.2.2 Description

The cost properties statement allows you to specify one or more costs for a given category. The default categories provided with SIMDASE are default categories used by the Structured Intuitive Model for Product Line Economics (SIMPLE), see section D.9 for listing. However, custom categories can be specified using any valid XText identifier.

The costs are represented as either fixed or employee based costs where employee based costs have the form:

```
1 <NUMBER_OF_EMPLOYEES> "<EMPLOYEE_TYPE>" for <STATEMENT>
```

and fixed costs have the form:

```
1 <COST> for "<COST_DESCRIPTION>" at <TIME_PERIOD>
```

Statements, used with employee based costs, can have any standard mathematical operation (+, -, *, /, %) as well as exponentiation (**) along with several functions (sin, cos, tan, random). Also available is the depth of the component in the architecture (i), the current time period (t) and the scaling factor for the current component (s()).

Statements also support branching logic statements.

```
1 if(<BOOLEAN_CONDITION>) { <TRUE_STATEMENT> } else { <FALSE_STATEMENT> }
```

Boolean conditions support all statement operators as well as the standard C logical operators (&&, ||, !, ==, ¡=, ¡, ¿=, ¿, !=).

D.2.3 Example

```
1 annex simdase {**
2     cost => {
3         organizational_planning => {
4             2.0 "Project Manager" for if(t == 0.0) { 40.0 } else { 0.0 },
```

```
5               20000.0 for "Planning Software Purchase" at 0.0
6           };
7       };
8   };
```

D.3 Is Adaptive Statement

D.3.1 Format

```
1 is_adaptive => <TRUE | FALSE>;
```

D.3.2 Description

The is adaptive statement is a reporting statement that allows you to specify whether the costs specified included adaptation based costs as well as traditional costs.

D.3.3 Example

```
1 annex simdase {**
2     is_adaptive => true;
3 };
```

D.4 Reevaluate Averaging Statement

D.4.1 Format

```
1 run for <COUNT> times taking <AVERAGE | MINIMUM | MAXIMUM> result;
```

D.4.2 Description

When cost statements include calls to the random() function, it is sometimes desirable to calculate the cost for those components many times taking either the average, minimum or maximum result rather than computing the cost only once. This statement facilitates that purpose.

D.4.3 Example

```
1 annex simdase {**
2     run for 2.0 times taking average result;
```

```
};
```

D.5 Scaling Factor Statement

D.5.1 Format

```
scaling_factor => <STATEMENT>;
```

D.5.2 Description

For abstract components that are included in multiple layers of the architecture, it sometimes necessary to scale the cost of that layers component based on either time, depth in architecture or other factors. This statement allows you to scale the cost of a component based on any factor available via a normal statement.

Statements, used with employee based costs, can have any standard mathematical operation (+, -, *, /, %) as well as exponentiation (**) along with several functions (sin, cos, tan, random). Also available is the depth of the component in the architecture (i), the current time period (t) and the scaling factor for the current component (s()).

Statements also support branching logic statements.

```
if(<BOOLEAN_CONDITION>) { <TRUE_STATEMENT> } else { <FALSE_STATEMENT> }
```

Boolean conditions support all statement operators as well as the standard C logical operators (&&, ||, !, ==, ¡=, ¡, ¿=, ¿, !=).

D.5.3 Example

```
annex simdase {**
    scaling_factor => 1 / (i + 1);
};
```

D.6 Time Minimum Statement

D.6.1 Format

```
tmin => <STARTING_TIME_PERIOD>;
```

D.6.2 Description

Components are rarely developed during a single time period or sprint. The time minimum and time maximum statements allow cost to be calculated across multiple time periods or sprints.

D.6.3 Example

```
annex simdase {**
    tmin => 1.0;
};
```

D.7 Time Maximum Statement

D.7.1 Format

```
tmax => <ENDING_TIME_PERIOD>;
```

D.7.2 Description

Components are rarely developed during a single time period or sprint. The time maximum and time minimum statements allow cost to be calculated across multiple time periods or sprints.

D.7.3 Example

```
annex simdase {**
    tmax => 20.0;
};
```

D.8 Variants Statement

D.8.1 Format

```
variants => {
    <VARIANT_1>: {<COMPONENT_1>, <COMPONENT_2, ... <COMPONENT_N>};
    <VARIANT_2>: {<COMPONENT_1>, <COMPONENT_2, ... <COMPONENT_N>};
    ...
    <VARIANT_N>: {<COMPONENT_1>, <COMPONENT_2, ... <COMPONENT_N>};
};
```

D.8.2 Description

The variants statement allows you to specify a list of variants and which subcomponents of the current component are associated with that variant. Active variant statements can then be used to determine which subcomponents should have their cost factored into the current calculation.

D.8.3 Example

```
annex simdase {**
    variants => {
        "wifi_enabled":{
            "wireless_module","rfid_reader","send_tracking_info_via_wifi",
            "protocol_management_process"
        };
        "ble_enabled":{
            "bluetooth_module","rfid_reader","send_tracking_info_via_ble",
            "protocol_management_process"
        };
    };
};
```

D.9 SIMDASE Default Cost Categories

- architecture_definition
- architecture_evaluation
- component_development
- mining_existing_assets
- requirements_engineering
- software_system_integration
- testing
- understanding_relevant_domains

- using_externally_available_software
- configuration_management
- commission_analysis
- measurement_tracking
- process_discipline
- scoping
- technical_planning
- technical_risk_management
- tool_support
- building_business_case
- customer_interface_management
- developing_acquisition_strategy
- funding
- launching_institutionalizing
- market_analysis
- operations
- organizational_planning
- organizational_risk_management
- structuring_organization
- technology_forecasting
- training

Appendix E XText Grammar

```
AnnexLibrary returns aadl2::AnnexLibrary:
 SimdaseLibrary;

AnnexSubclause returns aadl2::AnnexSubclause:
 SimdaseSubclause;

SimdaseLibrary:
 {SimdaseContractLibrary} contract=SimdaseContract;

SimdaseSubclause:
 {SimdaseContractSubclause} contract=SimdaseContract;

SimdaseContract returns Contract:
 {SimdaseContract} (statement+=SIMDASEStatement)*;

SIMDASEStatement:
 {IsAdaptiveStatement} 'is_adaptive' '=>' value=Boolean ';'
 | {TMax} 'time_max' '=>' value=Number ';'
 | {TMin} 'time_min' '=>' value=Number ';'
 | {ScalingFactor} 'scaling_factor' '=>' statement=Statement ';'
 | {Variants} 'variants' '=>' '{' variants=Variants? '}' ';'
 | {ActiveVariants} 'active_variants' '=>' '{' variants=ActiveVariants? '}'
     ';'
 | {ReEvaluateAveraging} 'run' 'for' count=Number 'times' 'taking'
     type=('average' | 'minimum' | 'maximum') 'result' ';'
 | CostProperties ';';

Variants:
 first=Variant (';' rest+=Variant)* ';';

Variant:
 name=STRING ':' '{' variants=VariantNames? '}';
```

```
31
32 ActiveVariants:
33  first=ActiveVariant (';' rest+=ActiveVariant)* ';';
34
35 ActiveVariant:
36  name=STRING ':' '[' variants=VariantNames? ']';
37
38 VariantNames:
39  first=STRING (',' rest+=STRING)*;
40
41 CostProperties:
42  {CostProperties} 'cost' '=>' '{' costStatements+=CostStatement* '}';
43
44 CostStatement:
45  type='architecture_definition' '=>' value=CostValue
46  | type='architecture_evaluation' '=>' value=CostValue
47  | type='component_development' '=>' value=CostValue
48  | type='mining_existing_assets' '=>' value=CostValue
49  | type='requirements_engineering' '=>' value=CostValue
50  | type='software_system_integration' '=>' value=CostValue
51  | type='testing' '=>' value=CostValue
52  | type='understanding_relevant_domains' '=>' value=CostValue
53  | type='using_externally_available_software' '=>' value=CostValue
54  | type='configuration_management' '=>' value=CostValue
55  | type='commission_analysis' '=>' value=CostValue
56  | type='measurement_tracking' '=>' value=CostValue
57  | type='process_discipline' '=>' value=CostValue
58  | type='scoping' '=>' value=CostValue
59  | type='technical_planning' '=>' value=CostValue
60  | type='technical_risk_management' '=>' value=CostValue
61  | type='tool_support' '=>' value=CostValue
62  | type='building_business_case' '=>' value=CostValue
63  | type='customer_interface_management' '=>' value=CostValue
```

```
64  | type='developing_acquisition_strategy' '=>' value=CostValue
65  | type='funding' '=>' value=CostValue
66  | type='launching_institutionalizing' '=>' value=CostValue
67  | type='market_analysis' '=>' value=CostValue
68  | type='operations' '=>' value=CostValue
69  | type='organizational_planning' '=>' value=CostValue
70  | type='organizational_risk_management' '=>' value=CostValue
71  | type='structuring_organization' '=>' value=CostValue
72  | type='technology_forecasting' '=>' value=CostValue
73  | type='training' '=>' value=CostValue
74  | type=ID '=>' value=CostValue;
75
76 CostValue:
77  '{' first=EmployeeStatement (',' rest+=EmployeeStatement)* '}' ';';
78
79 EmployeeStatement:
80  {HourlyCost} count=Number employeeType=STRING 'for' cost=Statement
81  | {FixedCost} cost=Number 'for' costReason=STRING 'at' period=Number;
82
83 Boolean:
84  value='true' | value='false';
85
86 Number:
87  (negative='-')? floatValue=FLOAT ;
88
89 Statement:
90  statement=StatementLvl2 (op+=('+' | '-') rest+=StatementLvl2)*;
91
92 StatementLvl2:
93  statement=StatementLvl3 (op+=('*' | '/' | '%') rest+=StatementLvl3)*;
94
95 StatementLvl3:
96  statement=StatementLvl4 (op+='**' rest+=StatementLvl4)*;
```

```
97
98 StatementLvl4:
99  {Integer} value=Number | {T} 't' | {I} 'i'
100  | {Sin} 'sin(' statement=Statement ')'
101  | {Cos} 'cos(' statement=Statement ')'
102  | {Tan} 'tan(' statement=Statement ')'
103  | {Paren} '(' statement=Statement ')'
104  | {If} 'if(' booleanStatement=BooleanStatement ')' '{'
        trueStatement=Statement '}' ('else' '{' falseStatement=Statement '}')?
105  | {Random} 'random()'
106  | {StatementScalingFactor} 's()';
107
108 BooleanStatement:
109  statement=BooleanStatementLvl2 (op+='||' rest+=BooleanStatementLvl2)*;
110
111 BooleanStatementLvl2:
112  statement=BooleanStatementLvl3 (op+='&&' rest+=BooleanStatementLvl3)*;
113
114 BooleanStatementLvl3:
115  statement=BooleanStatementLvl4 (op+=('>' | '>=' | '<' | '<=' | '==')
        rest+=BooleanStatementLvl4)*;
116
117 BooleanStatementLvl4:
118  {WithNot} '!' statement=BooleanStatementLvl4 | {WithoutNot}
        statement=BooleanStatementLvl5;
119
120 BooleanStatementLvl5:
121  statement=BooleanStatementLvl6 (op+=('+' | '-') rest+=BooleanStatementLvl6)*;
122
123 BooleanStatementLvl6:
124  statement=BooleanStatementLvl7 (op+=('*' | '/' | '%')
        rest+=BooleanStatementLvl7)*;
125
```

```
126 BooleanStatementLvl7:
127  statement=BooleanStatementLvl8 (op+='**' rest+=BooleanStatementLvl8)*;
128
129 BooleanStatementLvl8:
130  {BoolBoolean} value=Boolean | {BoolT} 't' | {BoolI} 'i' | {BoolInteger}
         value=Number
131  | {BoolSin} 'sin(' statement=Statement ')'
132  | {BoolCos} 'cos(' statement=Statement ')'
133  | {BoolTan} 'tan(' statement=Statement ')'
134  | {BoolParen} '(' statement=BooleanStatement ')'
135  | {BoolRandom} 'random()'
136  | {BoolScalingFactor} 's()';
137
138 terminal FLOAT: INTEGER_LIT '.' INTEGER_LIT;
```

Bibliography

[1] GitHub agree inheritance support. `https://github.com/smaccm/smaccm/issues/39`. Accessed: 2018-05-28.

[2] https://www.iso.org/standard/35733.html.

[3] Radhakisan Baheti and Helen Gill. Cyber-physical systems. *The impact of control technology*, 12:161–166, 2011.

[4] Amel Bennaceur, Ciaran McCormick, Jesús García-Galán, Charith Perera, Andrew Smith, Andrea Zisman, and Bashar Nuseibeh. Feed me, feed me: An exemplar for engineering adaptive software. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2016 IEEE/ACM 11th International Symposium on*, pages 89–95. IEEE, 2016.

[5] Daniel M Berry, Betty HC Cheng, and Jia Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *11th International Workshop on Requirements Engineering Foundation for Software Quality (REFSQ)*, page 5, 2005.

[6] Barry Boehm, Chris Abts, and Sunita Chulani. Software development cost estimation approachesa survey. *Annals of software engineering*, 10(1-4):177–205, 2000.

[7] Barry W Boehm, A Winsor Brown, Raymond J Madachy, and Ye Yang. A Software Product Line Life Cycle Cost Estimation Model. *ISESE*, 2004.

[8] Barry W Boehm et al. *Software engineering economics*, volume 197. Prentice-hall Englewood Cliffs (NJ), 1981.

[9] Barry W Boehm, Ray Madachy, Bert Steece, et al. *Software cost estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[10] Jones Capers. Applied software measurement, 1996.

[11] Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Transactions on Software Engineering*, 30(6):388–402, 2004.

[12] Paul Clements and John McGregor. Better, faster, cheaper: Pick any three. *Business horizons*, 55(2):201–208, 2012.

[13] Paul Clements and Linda Northrop. *Software product lines*. Addison-Wesley,, 2002.

[14] Paul C Clements, John D McGregor, and Sholom G Cohen. The structured intuitive model for product line economics (simple). Technical report, DTIC Document, 2005.

[15] Darren Cofer, Andrew Gacek, Steven Miller, Michael W Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In *NASA Formal Methods Symposium*, pages 126–140. Springer, 2012.

[16] Rogério De Lemos and José Luiz Fiadeiro. An architectural support for self-adaptive software for treating faults. In *Proceedings of the first workshop on Self-healing systems*, pages 39–42. ACM, 2002.

[17] Rogério De Lemos, Holger Giese, Hausi A Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M Villegas, Thomas Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[18] Ismayle de Sousa Santos, Magno Luã de Jesus Souza, Michelle Larissa Luciano Carvalho, Thalisson Alves Oliveira, Eduardo Santana de Almeida, and Rossana Maria de Castro Andrade. Dynamically adaptable software is all about modeling contextual variability and avoiding failures. *IEEE Software*, 34(6):72–77, 2017.

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

[20] Roland Dietrich and Frank Hagl. A polymorphic type system with subtypes for prolog. In *European Symposium on Programming*, pages 79–93. Springer, 1988.

[21] Peter Feiler, David Gluch, and John Hudak. The architecture analysis & design language (aadl): An introduction. 2008.

[22] Philip WL Fong and Robert D Cameron. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):379–409, 2000.

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

[24] Simos Gerasimou, Radu Calinescu, Stepan Shevtsov, and Danny Weyns. Undersea: an exemplar for engineering self-adaptive unmanned underwater vehicles. In *Proceedings of the 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 83–89. IEEE Press, 2017.

[25] Ilias Gerostathopoulos, Tomas Bures, Petr Hnetynka, Adam Hujecek, Frantisek Plasil, and Dominik Skoda. Strengthening adaptation in cyber-physical systems via meta-adaptation strategies. *ACM Transactions on Cyber-Physical Systems*, 1(3):13, 2017.

[26] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, 2012.

[27] Alwyn E Goodloe and César A Muñoz. Compositional verification of a communication protocol for a remotely operated aircraft. *Science of Computer Programming*, 78(7):813–827, 2013.

[28] Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. *Computer*, 41(4), 2008.

[29] Pao-Ann Hsiung, Yean-Ru Chen, and Yen-Hung Lin. Model checking safety-critical systems using safecharts. *IEEE Transactions on Computers*, 56(5):692–705, 2007.

[30] Hoh Peter In, Jongmoon Baik, Sangsoo Kim, Ye Yang, and Barry W Boehm. A quality-based cost estimation model for the product line life cycle. *Commun. ACM*, 2006.

[31] Juan F Inglés-Romero and Cristina Vicente-Chicote. Towards a formal approach for prototyping and verifying self-adaptive systems. In *International Conference on Advanced Information Systems Engineering*, pages 432–446. Springer, 2013.

[32] Randall Jensen. An improved macrolevel software development resource estimation model. In *5th ISPA Conference*, pages 88–92, 1983.

[33] Patrick Kinney et al. Zigbee technology: Wireless control that simply works. In *Communications design conference*, volume 2, pages 1–7, 2003.

[34] Xiao-zhu Lin, Hai-yan Wu, Dong-Xing Jiang, and Feng-yuan Ren. A self-adaptive architecture to control the performance of multi-host web servers. In *Telecommunication Networks and Applications Conference, 2007. ATNAC 2007. Australasian*, pages 75–80. IEEE, 2007.

[35] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[36] Ethan T McGee and John D McGregor. Using dynamic adaptive systems in safety-critical domains. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 115–121. ACM, 2016.

[37] Emanuela Merelli, Nicola Paoletti, and Luca Tesei. Adaptability checking in complex systems. *Science of Computer Programming*, 115:23–46, 2016.

[38] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run. time to support dynamic adaptation. *Computer*, 42(10), 2009.

[39] Anitha Murugesan, Michael W Whalen, Sanjai Rayadurgam, and Mats PE Heimdahl. Compositional verification of a medical device system. In *ACM SIGAda Ada Letters*, volume 33, pages 51–64. ACM, 2013.

[40] Dirk Niebuhr and Andreas Rausch. Guaranteeing correctness of component bindings in dynamic adaptive systems based on runtime testing. In *Proceedings of the 4th international workshop on Services integration in pervasive environments*, pages 7–12. ACM, 2009.

[41] Andy J Nolan, Silvia Abrahao, Paul Clements, John D McGregor, and Sholom Cohen. Towards the Integration of Quality Attributes into a Software Product Line Cost Model. In *2011 15th International Software Product Line Conference (SPLC)*, pages 203–212. IEEE, July 2011.

[42] Lawrence Putnam and Ware Myers. Measures for excellence, 1992.

[43] Lawrence H Putnam and Ware Myers. *Measures for excellence: reliable software on time, within budget*. Prentice Hall Professional Technical Reference, 1991.

[44] Xin Qi, Kun Wang, Anpeng Huang, Lei Shu, and Yan Liu. A harvesting-rate oriented self-adaptive algorithm in energy-harvesting wireless body area networks. In *Industrial Informatics (INDIN), 2015 IEEE 13th International Conference on*, pages 966–971. IEEE, 2015.

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

[46] Pete Sawyer, Nelly Bencomo, Jon Whittle, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 95–103. IEEE, 2010.

[47] Ralph H Sprague Jr. A framework for the development of decision support systems. *MIS quarterly*, pages 1–26, 1980.

[48] Jacob Swanson, Myra B Cohen, Matthew B Dwyer, Brady J Garvin, and Justin Firestone. Beyond the rainbow: self-adaptive failure avoidance in configurable systems. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 377–388. ACM, 2014.

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

[50] Danny Weyns and Tanvir Ahmad. Claims and evidence for architecture-based self-adaptation: a systematic literature review. In *European Conference on Software Architecture*, pages 249–265. Springer, 2013.

[51] Danny Weyns, M Usman Iftikhar, Sam Malek, and Jesper Andersson. Claims and supporting evidence for self-adaptive systems: A literature study. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 89–98. IEEE Press, 2012.

[52] Wayne Wolf. Cyber-physical systems. *Computer*, 42(3):88–89, 2009.

[53] Wenhua Yang, Chang Xu, Yepang Liu, Chun Cao, Xiaoxing Ma, and Jian Lu. Verifying self-adaptive applications suffering uncertainty. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 199–210. ACM, 2014.

[54] Zhibin Yang, Kai Hu, Dianfu Ma, and Lei Pi. Towards a formal semantics for the aadl behavior annex. In *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, pages 1166–1171, April 2009.

[55] Jaeyoung Yun, Jinsu Park, and Woongki Baek. Hars: A heterogeneity-aware runtime system for self-adaptive multithreaded applications. In *Proceedings of the 52nd Annual Design Automation Conference*, page 107. ACM, 2015.

[56] Y Yushtein, Marco Bozzano, Alessandro Cimatti, J-P Katoen, Viet Yen Nguyen, Th Noll, Xavier Olive, and Marco Roveri. System-software co-engineering: Dependability and safety perspective. In *Space Mission Challenges for Information Technology (SMC-IT), 2011 IEEE Fourth International Conference on*, pages 18–25. IEEE, 2011.