

SOFTWARE TESTABILITY MEASURE FOR SAE ARCHITECTURE ANALYSIS  
AND DESIGN LANGUAGE (AADL)

---

A Thesis  
Presented to  
the Graduate School of  
Clemson University

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science  
Computer Science

---

by  
Hung Hoang Manh Vo  
August 2012

---

Accepted by:  
Dr. John D. McGregor, Committee Chair  
Dr. Murali Sitaraman  
Dr. D. E. Stevenson

## ABSTRACT

Testability is an important quality attribute of software, especially for critical systems such as avionics, medical, and automotive. Improvement in the early testability of software architecture, the first artifact of the software system, will help reduce issues and costs later in the development process.

AADL, an architecture analysis description language suitable for critical embedded, real-time systems, can be used for design documentation, analysis and code generation. Because the capability of AADL can be extended, it is possible to add new analyses to its core language. Tools such as the Open Source AADL Tool Environment (OSATE) provide plugins for processing AADL models. Although adding new plugins in OSATE extends AADL, there currently exists no AADL extension for testability measurement. The purpose of this thesis is to propose such a method to measure the testability of AADL models as well as to develop a testability plugin in OSATE.

Much research has been conducted on testability of hardware, software and embedded systems, resulting in several approaches for measuring this quality attribute. Among them, the approach measuring testability as a product of controllability and observability using information transfer graph (ITG) is the most applicable for measuring the testability of AADL models. This thesis proposes a method applying this approach to AADL models. A complete testability measure plugin for OSATE was developed based on this approach and detailed examples are given in this thesis to demonstrate its applicability.

## DEDICATION

To my dear family, my wife Song Thuan, my father Vo Dinh Khuynh, my younger sister Chia and especially my mother Hoang Thi Minh Thien; I wish you the best in heaven.

## ACKNOWLEDGMENTS

I would like to give special thanks to my advisor, Dr. John McGregor, for his guidance and support throughout my study at Clemson University. I would also like to thank Dr. Murali Sitaraman and Dr. D. E. Stevenson for their guidance during the early phase of my study and for serving as my committee members. In addition, I appreciate Dr. Mark Smotherman for his guidance and help with funding paperwork and curriculum issues. I owe a special thank you to the Vietnam Education Foundation for its sponsorship and funding and for giving me the opportunity to further my education. Thanks also go to Ms. Barbara Ramirez, Director of the Class of 1941 Studio, for her help in editing this thesis. Finally, I would like to thank all my friends in Clemson and especially my family for their encouragement, love and support.

## TABLE OF CONTENTS

	Page
ABSTRACT .....	ii
DEDICATION .....	iii
ACKNOWLEDGMENTS .....	iv
LIST OF TABLES .....	vii
LIST OF FIGURES .....	viii
CHAPTER	
I. CHAPTER ONE - INTRODUCTION.....	1
Research approach and contributions .....	3
Thesis organization .....	3
II. CHAPTER TWO - BACKGROUND.....	5
AADL .....	5
OSATE.....	10
Testability as controllability and observability.....	11
Information theory .....	12
Information transfer graph .....	14
Flows in information transfer graph .....	16
Information channel .....	15
Testability measurement using information transfer graph .....	18
III. CHAPTER THREE - THEORY EXPOSITION IN DETAIL.....	25
Testability measurement for AADL models.....	25
Connection instances in the AADL instance model .....	25
Converting the AADL instance model into ITG.....	31
Calculating the upper bounded testability.....	37
Channel capacity calculation .....	42

Table of Contents (Continued)

	Page
IV. CHAPTER FOUR – EXPERIMENTAL RESULTS.....	44
Define the new property set .....	44
Example 1 .....	44
Example 2 .....	61
V. CHAPTER FIVE – RELATED WORKS.....	65
Testability analysis for data flow software .....	65
Framework for model-based testing AADL models.....	66
Test case generation based on AADL system component modes .....	67
VI. CHAPTER SIX – CONCLUSIONS AND FUTURE WORKS .....	68
REFERENCES .....	70

## LIST OF TABLES

Table	Page
Table 1 – Controllability /observability /testability measure for flow 1 .....	59
Table 2 - Controllability /observability /testability measure for flow 2 .....	60
Table 3 - Controllability /observability /testability measure for flow 3 .....	60
Table 4 - Calculation results for example 2 .....	64

## LIST OF FIGURES

Figure	Page
Figure 1 - Three types of AADL model representation (12) .....	9
Figure 2 - Relationships of entropy, conditional information and transinformation .....	13
Figure 3 - Sample of Information Transfer Graph (ITG) .....	15
Figure 4 - Transition modes .....	15
Figure 5 - Sample ITG with 3 flows .....	16
Figure 6 - three flows generated from ITG in figure 5 .....	17
Figure 7 - Information channel M with input X and output Y .....	17
Figure 8 - Line channel .....	18
Figure 9 - semantic connection and connection declarations (12).....	26
Figure 10 -typical data port connections.....	28
Figure 11 - ITG of the piece of system in figure 10 .....	29
Figure 12 - typical event data port connections .....	29
Figure 13 - ITG of the piece of system in figure 12 .....	30
Figure 14 - sample AADL model to be converted to ITG .....	34
Figure 15 - result final specific model for “main” process from above AADL model.....	37
Figure 16 - a module before and after transformation .....	39
Figure 17 - super sources with infinite line capacity outputs added to existing ITG .....	39

Figure 18 - super sink with infinite line capacity inputs added to existing ITG...	40
Figure 19 - testability property set definition .....	44
Figure 20 - Sample model (part 1).....	45
Figure 21 - Sample model (part 2).....	46
Figure 22 - Sample model (part 3).....	47
Figure 23 - The AADL graphical representation of above AADL model .....	48
Figure 24 - ITG of the example system .....	52
Figure 25 - three generated flows numbered 1-3 from left to right .....	53
Figure 26 - Flow 2 (left) and flow 3 (right) after transformation .....	55
Figure 27 - demonstration of measuring $T'(SS, \text{reg2})$ (left) and $T'(\text{reg2}, SO)$ (right) on flow 3 .....	56
Figure 28 - demonstration of measuring $T'(SS, \text{decr})$ (left) and $T'(\text{decr}, SO)$ (right) on flow 2 .....	58
Figure 29 - Full system model .....	61
Figure 30 - Avionics_system_periodic_IO under test .....	62
Figure 31 - ITG for the system instance .....	63

## CHAPTER ONE

### INTRODUCTION

Software architecture of a program or computing system, defined as “the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them” (1), has recently become a critical focus in the design of software systems. It is important for three main reasons: First, it is the communication vehicle among the stakeholders. Second, it is often the first artifact of a software system created from the documents specifying its requirements, meaning it has a significant impact on the software development life cycle; as high level decisions are made during the process of creating the software architecture, it serves as a blueprint for the developers in the later phases. Finally, software architecture is a transferable abstraction of a system; it can be reused and applied to other systems requiring similar quality attributes and functional capabilities.

Software architecture provides the foundation for achieving such important quality attributes in a computer system as modifiability, performance, security and testability, the latter being derived from hardware testability (1-4). Testability has been defined several ways: as the ability to test software easily (5,6), as the probability of a program revealing faults (7), or as controllability and observability (4,8,9). Although there are many definitions of testability, software exhibiting high testability reduces the cost, time and effort of testing while at the same time enhancing confidence in the software after fewer tests (7), important because testing frequently costs 30%-70% of the

development budget; more importantly, improvement in testability improves the final product, especially in critical systems.

To improve testability, the first step is to design a method to measure it. Based on results of this method, a testing strategy can be derived, one that will help focus the testing using human and financial resources efficiently (8). If no fault is found during testing, the testability measure will indicate that the program is viable: the higher the testability score, the more viable the program (10).

While it is necessary to have testability measured early in the software architecture design process, it is also advantageous to have it run automatically for large projects to reduce human effort and save time and cost. With the support of Architecture Description Languages (ADL) such as AADL, xADL, Wright, Rapide, ACME, and MetaH (11), this automation is possible. Because ADL, a formal language developed for capturing the architecture of a system, can support architecture analysis, visualization, and simulation, it can be extended to support testability measurements automatically.

AADL, an ADL previously developed to describe avionics system architecture, is currently being widely used to model the software and hardware architecture of embedded, real-time systems, complex systems of systems, and specialized performance capability systems (12). Open Source AADL Tool Environment (OSATE) (13) developed by The Software Engineering Institute is a powerful toolset for front-end processing of AADL models. It provides tools for model statistics; miss rate and flow latency analysis; security, and safety levels checks. However, none of these can be used for testability measurement. OSATE, built using the Eclipse platform, can be extended by

developing a new plugin. Therefore, AADL with the support of OSATE is a good choice for developing a program measuring software testability.

Thus, the aim of this thesis is to propose a software testability measurement method for AADL models and subsequently build a plugin on OSATE that can automatically analyze architecture components written in AADL. Based on these quantified testability measures, a test strategy can be proposed as well as insight can be gained in how to improve software architecture for higher testability.

### Research Approach and Contributions

There are many approaches to measure testability including sensitivity (14), domain/ range ratio (15), class complexity or complexity of object interactions (6,16), and controllability and observability (5,17). Among them, measuring testability as controllability and observability as proposed by (5,17) is most applicable for AADL models. For this reason, this research adapts it to develop an OSATE plugin for AADL model testability measurement.

In addition, this thesis details the method theory and the plugin implementation for measuring AADL model testability as controllability and observability. The source code and its instructions are also provided. To validate the plugin as well as the approach in this thesis, detailed examples are provided.

### Thesis Organization

This thesis is organized into 6 chapters. Chapter 1 introduces the problem, the research approach, its contributions and the organization of the thesis. The next chapter

reviews background information on AADL, OSATE, testability measure as controllability and observability, and information theory. Chapter 3 details the theory more fully, specifically the approach for measuring the testability of software components in AADL models as controllability and observability using information transfer graphs. Chapter 4 discusses the implementation and experiment results, while Chapter 5 compares past research to that presented in this thesis. The last chapter includes the conclusions and proposes future work.

## CHAPTER TWO

### BACKGROUND

#### AADL

AADL, a formal architecture analysis description language standardized by the Society of Automotive Engineers (SAE), was first developed for avionics, but later adapted for such embedded, real-time systems as automotive, autonomous and medical equipment. It supports a full Model Based Development life cycle and can be used for design documentation, architecture analysis, code generation, guidance in system integration, and system tuning and upgrades over the lifecycle (18). This chapter briefly discusses the basis of AADL as well as its applications and capabilities. More detail about AADL can be found in (12).

#### *AADL applications and benefits*

Based on more than 12 years of research and over 40 experiments (18), AADL applications can be found in many fields including avionics (19), aerospace (20), military (21) automotive (22), cyber-physical systems (23) and medical (24,25) in addition to various real-time concurrent processing domains including critical safety applications. The most wide-spread applications of AADL occur in the avionics field, especially those used by Honeywell(26), Airbus(27,28), and Rockwell Collins(29,30) for modeling, analyzing, and capturing reference architecture of large modern aircraft systems, military helicopters, and air transport and regional aircrafts(21). Those are complex, real-time,

fault tolerant, and performance-critical systems and systems of systems requiring reliability, security and safety.

The benefits of AADL include integration support, reduction in the development time and cost via describing dynamic behavior runtime architecture, analysis throughout development life-cycle and early prediction of non-functional quality attributes such as performance, security, safety, and reliability. System integration for complex, critical systems is one of the highest areas of program risk (18). With AADL, system integration can be done automatically if the system AADL model is fully specified and the source code for the software components is provided. With a single architectural model that can be delivered to multiple contractors, an AADL document is a communication vehicle between stakeholders. AADL also supports system evolution and large-scale development by taking advantage of reference architectures and the reuse of components in software product line development. Those benefits result from AADL because it is a well-defined architecture document with strong semantics, clear annotations commonly used across organizations, and extensible capability (31) (12).

#### *AADL basics* (12)

The formal modeling concepts in AADL are used to describe and analyze the architectures of systems or systems of systems through their components and the interactions among them. The abstractions of system components in AADL include application software, execution platforms (hardware) and composites (system). These component abstractions are applied to specify the systems and to map the software elements onto hardware platforms.

System components can contain application software and execution hardware or other systems to represent systems of systems. System components allow the integration of other components into distinct units, while application software includes thread, thread groups, processes, data, and subprograms. Execution platforms include four types of components, the processor, memory, device, and bus. These components are defined below:

- Application software:
  - Thread: active components can be concurrently executed and organized into thread groups.
  - Thread group: logical unit for organizing threads, data and other thread groups
  - Process: protected address space that threads run on
  - Data: data types and static data in source text
  - Subprogram: callable piece of executable code (functions, methods).
- Execution Platform:
  - Processor: component that schedules and executes threads.
  - Memory: component that stores data and code
  - Device: represents components that are external environments or interfaces with external environments such as sensors and actuators
  - Bus: component that provides shared access and interconnections between other execution platform components

- Composite:
  - System: component that enables the integration of other components including application software, execution platforms and other composite components.

Each component in an AADL system specification is visible to other components by its declaration and is realized by its implementation. A declaration characterizes a component by its identity and its possible interfaces with other components. Those interfaces are defined as features such as connection ports, port groups, data accesses, bus accesses, and subprograms. A component implementation shows the internal structure of a component and the way the internal subcomponents interact with one another. In both component declaration and component implementation, detailed characteristics of a component are also specified by such properties as type of data, bandwidth of connections, and security level of a component and execution deadline of a thread.

An AADL model includes both a declarative model and an instance model. An AADL instance model, which represents the instantiation of a system or the physical system runtime architectures, is used to analyze the runtime behaviors. A system instance is created by instantiating the top-level system implementation and recursively instantiating the subcomponents. Once instantiated, the application software components can be bound to their corresponding execution platform. Depending on the level of detail of the component implementations in the declarative model, system instance can be fully instantiated or partially instantiated, with some analyses requiring a system instance to be fully instantiated.

An AADL declarative model consists of textual and graphical language and XML representation with precise semantics for modeling the architecture of embedded software systems and their target platforms. However, an AADL instance model can be represented in XML format only. The figure below taken from (12) exemplifies the three ways to represent a piece of an AADL declarative model.

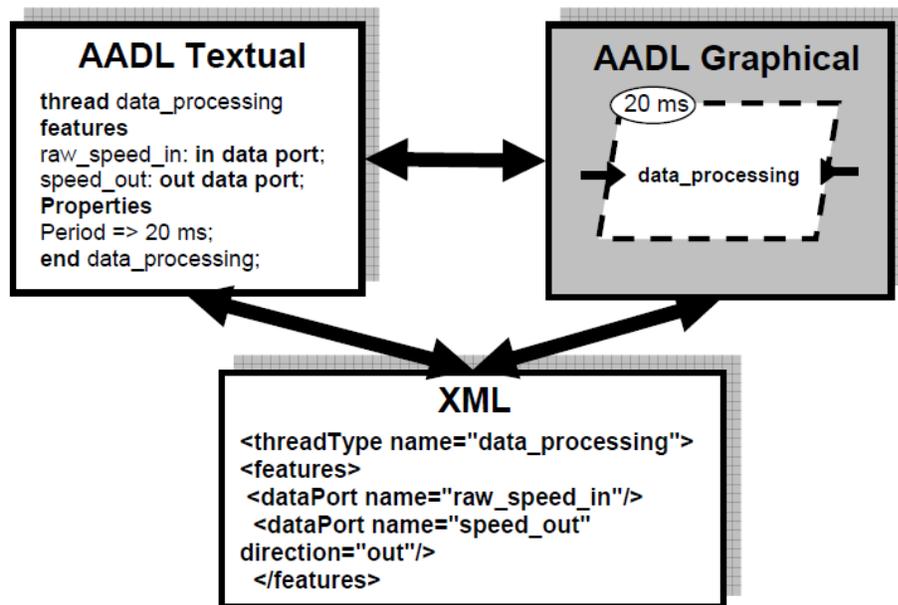


Figure 1 - Three types of AADL model representation (12)

### *AADL and Model Based Development for system analysis (12,18)*

AADL inherited the Model Based Development process used in MetaH developed by Honeywell, extending it with additional capability and flexibility. The Model Based Development process uses the concepts of architectural specification, architecture analysis, and automated integration to construct the final system.

AADL models can be used to design or integrate new systems or analyze existing ones. Those system specifications can be partially defined or fully specified. The

architectural analysis can be conducted on AADL models at different levels of details. At a very early development phase, a high level architectural model is analyzed for early trade-off analysis and selection of component architectures. Early prediction and analysis for critical system qualities such as performance, schedulability, and reliability are supported by AADL. Architecture specifications can be further refined during the development life cycle, with analysis being rerun each time the specification is updated. Large systems and system product lines can be built automatically from existing system design databases. Team-based development is also supported by organizing AADL elements into packages.

The extensible quality of AADL core language supports building new analysis tools, for example ones specifically written for system developers. New analysis tool or procedures can be applied to the existing AADL models by adding properties to components and additional modeling components such as ports and connections. In addition, new property sets and annex libraries can be defined, enabling extension and customization of an AADL specification to meet project or domain specific requirements. Some examples of annex libraries can be found in (32-35).

### OSATE

Tools such as Open Source AADL Tool Environment (OSATE) (13) provide toolsets for processing AADL models. OSATE is developed on top of Eclipse as a set of plugins. In addition to providing an environment for editing AADL models textually, graphically, and XML-based, OSATE also provides a set of plugins for analyzing AADL models, specifically their statistics, miss rate, security, safety and flow latency. OSATE

analysis tools can be extended by developing new plugins. While various researchers have methods for generating test cases based on AADL (36) and creating frameworks for model-based testing AADL models (37), no OSATE plugin allows for the testability analysis of AADL models. To address this need, this thesis developed an OSATE plugin that measures testability for software components in AADL models.

### Testability as controllability and observability

Testability is defined as the ease of revealing faults during testing. A high testability module of a system is more likely to reveal faults (if any) while testing than a low testability module (7). Testability can be considered as a combination of controllability and observability, with a system having these two properties being said to be testable (9). Controllability of a component is defined as the ease of bringing inputs from the software to the input of the component, while observability is the ease of propagating outputs from the component to the output of the software. (38)

According to (39,40), tools referred to as CATA and its extended version SATAN were developed to measure hardware testability as a product of controllability and observability. These methods, which were based on information theory, were then adapted to software systems(2,17), co-designed systems (41) and embedded data-flow software (42).

## Information Theory

### *Uncertainty*

For a discrete random variable  $X$  with values in the finite set  $\{x_1, x_2, \dots, x_n\}$  whose probability for the event  $X = x_i$  occurring is  $p(X=x_i) = p_i$ , uncertainty is defined as

$$I(x_i) = \log \frac{1}{p_i} = -\log p_i$$

It measures the information brought by the realization of the event  $X = x_i$ . The more likely that event  $X=x_i$  occurs, the less information the occurrence of this event provides.

### *Entropy – information quantity*

Entropy is defined by the expectation of the uncertainty of a random variable  $X$ .

$$H(X) = E[I(X)] = \sum_{i=1}^n p_i I(x_i) = - \sum_{i=1}^n p_i \log p_i$$

Entropy measures the information  $X$  contains. If the probability distribution is uniform,  $p_i = 1/n$  for all  $i$ , then  $H(X) = \log(n)$ .

### *Joint Information*

$$H(X, Y) = - \sum_{i=1}^n \sum_{j=1}^m p_{ij} \log p_{ij} = H(Y, X)$$

This is the information both  $X$  and  $Y$  contain.

### *Conditional Information*

$$H(X/y_j) = - \sum_i p(x_i/y_j) \log p(x_i/y_j)$$

$$H(X/Y) = \sum_i H(X/y_j) \cdot p(y_j)$$

$H(X/Y)$  quantifies the remaining uncertainty of a random variable  $X$  given that the value of other random variable  $Y$  is known. It measures the average information required to specify  $X$  when  $Y$  is known.

Properties:

$$H(X, Y) \leq H(X) + H(Y)$$

Equality if  $X$  and  $Y$  are independent

$$H(X/Y) \leq H(X)$$

$$H(X/Y) \leq H(Y)$$

$$H(X, Y) = H(X) + H(Y/X) = H(Y) + H(X/Y) = H(Y, X)$$

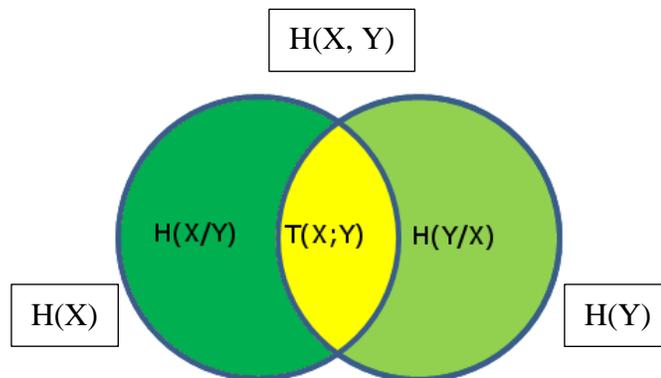
*Transinformation*

$$T(X; Y) = H(X) - H(X/Y) = H(Y) - H(Y/X)$$

Property:

$$T(X; Y) = T(Y; X)$$

$$0 \leq T(X; Y) \leq \min(H(X); H(Y))$$



**Figure 2 - Relationships of entropy, conditional information and transinformation**

Transinformation is the difference between the amount of prior information about  $X$  and the amount of information for the same variable  $X$  under the assumption that the value of variable  $Y$  is known.  $T(X; Y)$  expresses the uncertainty about  $X$  that can be reduced by the knowledge of the realization of  $Y$ . This is the amount of information contained in  $Y$  about  $X$ . Transinformation is a symmetrical function; therefore, it is also the amount of information contained in  $X$  about  $Y$ .

### Information Transfer Graph (43)(38)

Information transfers through a system are modeled using an information transfer graph (ITG), which can represent both control and data-flow in the same graph. ITG, which is used for testability analysis, is a bipartite directed graph containing places, transitions and arcs. The places and transitions are considered as nodes of the ITG. Places consist of

- The modules, which are components of a system
- The inputs for the system, called sources
- The observable outputs of the system, called sinks

Transitions represent modes of information transfer between places, while arcs connect places to transitions or transitions to places; transition is the media that conveys information between system modules, sources, and sinks.

This thesis uses the following graphical representation for elements in an ITG: an arc is represented by a line with arrow showing the information flow direction, a transition by a bar, modules by circles, sources by trapezoids, and sinks by diamonds as seen in the example below:

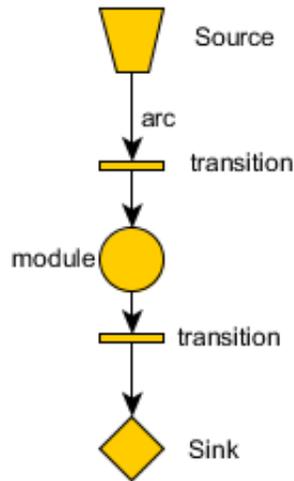


Figure 3 - Sample of Information Transfer Graph (ITG)

There are three transition modes which are divided into two groups: the OR mode group including attribution and selection modes and the AND mode group including the junction mode. The OR mode represents the situation when a place receives/ transmits data from/to one of its predecessor /successor places, while an AND mode represents when a place must receive data from all the predecessor places.

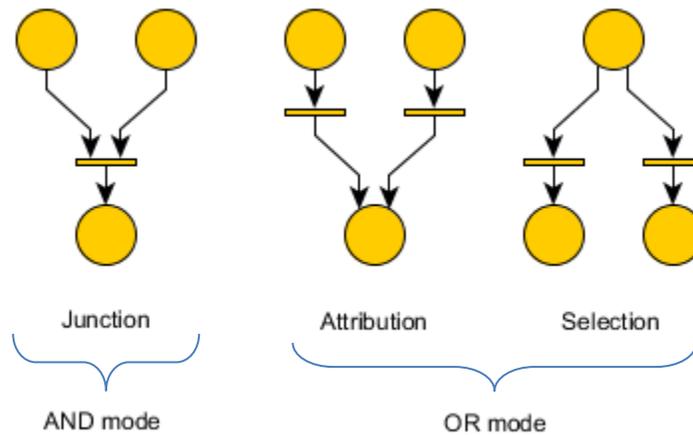


Figure 4 - Transition modes

### Flows in Information Transfer Graph

$X$  is the set of vertices representing all places and transitions of the ITG. A flow  $F$  is a sub-network of ITG built on a nonempty set of vertices  $X'$ , which is a subset of  $X$  such that:

- If a place belongs to  $X'$ , at least one of its predecessor transitions and at least one of its successor transitions are also in  $X'$  (except if the place is a source or a sink)
- If a transition belongs to  $X'$ , all places adjacent to it also belong to  $X'$
- For any place in  $X'$ , there exists a path in  $X'$  beginning from a source in  $X'$ , going through the place and ending at a sink in  $X'$

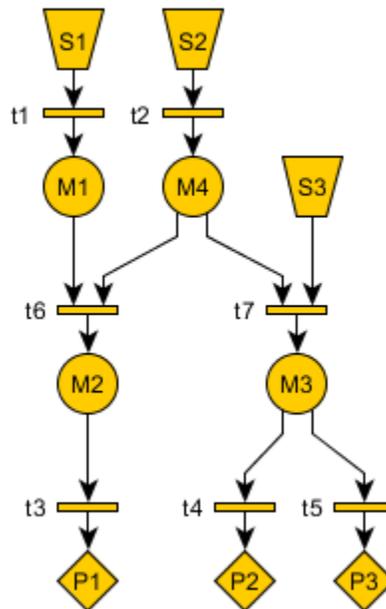


Figure 5 - Sample ITG with 3 flows

Figure 5 shows an example of an ITG with three flows:

- F1: (S1,S2,M1,M2,M4,P1)
- F2: (S2,S3,M3,M4,P3)
- F3: (S2,M3,M3,M4,P2)

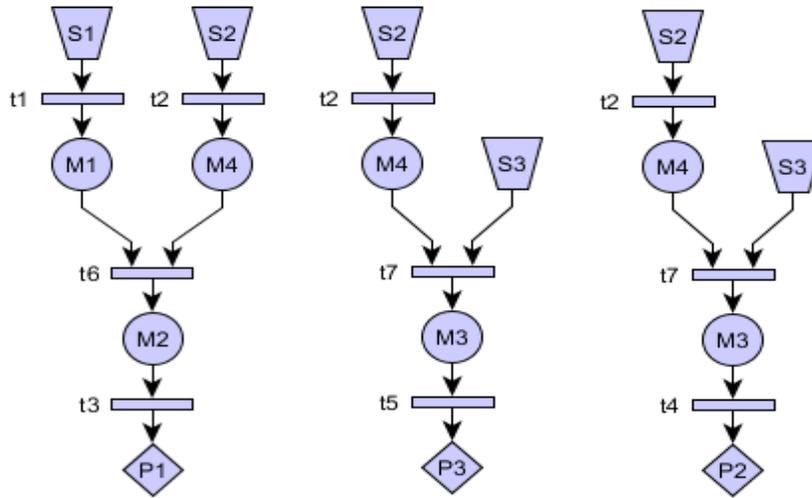


Figure 6 - three flows generated from ITG in figure 5

### Information Channel



Figure 7 - Information channel M with input X and output Y

An information channel M is a system that produces output information Y from input information X. Channel capacity  $C(X,Y)$  is the maximum of transinformation  $T(X,Y)$  over all possible distributions of X. Two typical types of channel capacity are line capacity and module capacity. Module capacity  $C(M)$ , which is the maximum

information quantity of the module inputs that can be observed at its outputs, expresses information lost through the module.



**Figure 8 - Line channel**

A line is a channel that carries the identity function  $Y = X$ . Line capacity  $C(X)$  is the maximum information of  $X$ , which can be achieved when the probability distribution of  $X$  is uniform. In binary representation,  $C(X) = N$ , where  $N$  is the number of bits needed to represent all possible values of  $X$ .

$$C(X) = \max T(X; X) = \max H(X) = \log 2^N = N$$

Testability measurement using Information Transfer Graph (43)(44)

In a flow on an ITG, all arcs coming into a module can route all the data needed to test it, while all arcs going away from a module contain the test results generated from this module. Thus, the testing of a module in a flow can be conducted involving only the materials represented by this flow, meaning a policy for testing and diagnosis can be defined in the terms of the flows.

There are controllability and observability problems in data flows. Arcs coming into a module representing the different data paths are sufficient to completely test it (complete controllability). However, when a module is considered in a flow, it is possible that the amount of data needed to test it cannot be delivered from the sources, leading to reduced controllability. Similarly, arcs starting from a module represent data paths sufficient to convey all the test results from this module (complete observability).

However, when a module is considered in a flow, it is possible that the results from the test may not be fully conveyed to the sinks, leading to a lack of observability.

Furthermore, a module belonging to several flows can have different controllability / observability in these flows, meaning that its complete test cannot be achieved by a single flow. Measures of controllability and observability for each flow are combined to provide a measure of testability at the architecture level. Flows in conjunction with controllability, observability and testability are used to provide insight about the modifications of the architecture specifications needed and to provide a testing strategy for later development phases.

When analyzing an ITG, there will be more than one flow present. Different flows have different sources and different sinks, meaning the control and observation points are different for each. Therefore, a module may have different testability measures based on its different flows. These measures depend on the amount of information issued from the sources of the flow going through the module and producing outputs at sinks. While the relative controllability of a module corresponding to a flow reflects the maximum amount of information that the module can receive at its inputs from the sources of the flow being considered, its relative observability reflects the maximum amount of information that can be transmitted from the module to the sinks. These two quantities are combined to define the testability measure of a module.

*Module capacity and transinformation:*

Quantifying controllability, observability and testability requires channel capacity and transinformation. Channel capacity includes two typical types, module and line capacity. Module capacity is the maximum amount of information that can be available at the outputs of the module based on its input. Line capacity (or arc capacity) is the maximum amount of information that can be sent over the arc. Capacity of module  $M$  is denoted as  $C(M)$  and capacity of a line (an arc)  $X$  is denoted as  $C(X)$ . There are two methods for calculating module capacity, exact evaluation and statistical evaluation (17). If no information is available about the module, one with  $k$  bits of output has an approximate capacity of  $C(M) = k$ .

Transinformation between two random variables  $X$  and  $Y$  denoted as  $T(X, Y)$  quantifies the information  $Y$  contains about  $X$ .  $T(X, Y)$  is a symmetrical function, meaning that  $T(X, Y) = T(Y, X)$ . The capacity of a module with input  $X$  and output  $Y$  is a maximum of  $T(X, Y)$  for the set of available distributions of  $X$ . For a flow  $F$  with a source  $S$ , sink  $O$ , and a module  $M$  with input  $X$  and output  $Y$ ,  $T(S, X)$  measures the amount of information that  $X$  receives from  $S$ ; this is the concept of controllability. Similarly,  $T(Y, O)$  measures the amount of information of  $Y$  that can be observed at  $O$  (concept of observability).

If F is any flow in the ITG and M is a module in flow F, the variable of this flow F is denoted as follow:

$X_M$ : inputs of module M

$Y_M$ : outputs of module M

$S_F$ : inputs of flow F

$O_F$ : sinks of flow F

When a module is isolated, all possible inputs of M can be generated, and M has complete controllability. In this case, M can receive the maximum amount of information, which is  $C(X_M)$ . However, when M is in flow F, the maximum amount of information that M can receive from F is  $T(S_F, X_M)$ . Controllability of M in flow F is computed as:

$$CO(M) = \frac{T(S_F, X_M)}{C(X_M)}$$

Similarly, when a module is isolated, all its output can be observed, and the total amount of information being received from its outputs is represented as  $C(Y_M)$ . However, when M is in flow F, the amount of information that is transferred from output of M to the sinks of F is  $T(Y_M, O_F)$ . This amount does not exceed  $C(Y_M)$ . Observability is calculated using the following formula:

$$OB(M) = \frac{T(Y_M, O_F)}{C(Y_M)}$$

Testability of M in flow F is the product of its controllability and observability.

$$TE(M) = CO(M) \times OB(M)$$

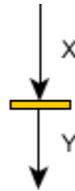
Controllability, observability, and testability values obtained from these formulas range from 0 to 1: the higher value of the testability measure, the higher of the testability of the module of the flow.

### *Calculating transinformation*

For both controllability and observability measure formulas, transinformation between flow sources and module inputs as well as transinformation between module outputs and flow sinks are needed. Therefore, an effective method for obtaining this transinformation is essential for testability measures.

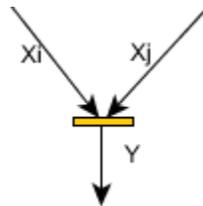
Below are characteristics of channel capacity and transinformation: (44)

- Propagation rules through the transitions:
  - o Simple transition:



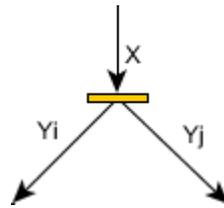
$$C(Y) = C(X)$$

- o Junction transition



$$C(Y) = \min [C(X_i, X_j), n_Y] , \text{ where } n_Y \text{ is the number of bits used to represent } Y$$

- Distribution transition



$$C(Y_i) = \min [C(X), n_i]$$

$$C(Y_j) = \min [C(X), n_j]$$

$$C(Y_i, Y_j) = \min [C(X), n_i + n_j]$$

- Property of transinformation:

For module  $M$  with input  $X_M$ , output  $Y_M$  and any variable  $Z$  in the ITG,

then

$$T(Z; Y_M) \leq \min[T(Z; X_M), T(X_M, Y_M)]$$

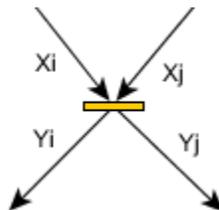
Because  $T(X_M, Y_M) \leq C(Y_M)$ :

$$T(Z; Y_M) \leq \min[T(Z; X_M), C(Y_M)]$$

Considering a junction transition with two incoming arcs  $X_i, X_j$  and one outgoing arc  $Y$  as a module, the following property results:

$$T(Z; Y) \leq \min[T(Z; X_i, X_j), C(Y)]$$

If it has fan-ins and fan-outs



$$T(Z; Y_i, Y_j) \leq \min[T(Z; X_i, X_j), C(Y_i) + C(Y_j)]$$

If there are many fan-ins, the exact computations of transinformation are quite complex as the information at the inputs of a fan-in can be less than the sum of the information available on each fan-in link because the information on each fan-in may be dependent; for example, some of it may come from the same fan-out. Moreover, the exact calculation of testability measures as defined above is difficult; it is of the same order of complexity as the generation of complete test cases, contradicting the goal of this research to find a preliminary assessment of testability.

To reduce complexity, transinformation  $T$  will be calculated approximately by replacing all inequalities in the above formulas with equalities. The resulting transinformation value is called the upper bounded transinformation  $T'$ . Dammak proposed a method for calculating all upper bounded transinformation values in ITG using a min-cut, max-flow algorithm (43). The testability measure formula with  $T$  replaced by  $T'$  is called the upper bounded testability.

## CHAPTER THREE

### THEORY EXPOSITION IN DETAIL

#### Testability measurement for AADL models

Unlike the AADL declarative model, the AADL instance model conveys only enough information to instantiate the system, representing the runtime architecture of the system. It also represents the architecture of the system after integration. Because a testability measure is meaningful in the context of system integration, this study examined the testability of components in the AADL instance model only.

A system can be fully or partially specified. The instantiating process begins with its implementation specification, and then each of its subcomponents is instantiated. This process is done recursively until the lowest level of system instance hierarchy is reached when no further subcomponent implementation is found. In a fully specified system, the application components, e.g. software parts, are modeled to the level of a thread or even to subprogram calls within a thread (12). While a fully specified system provides the most precise estimation of component testability, early in the development process, a partially specified system is also useful for giving a quick overview of system software testability. For both a fully or partially specified system, testability for the software components at the lowest level of the system instance hierarchy is measured.

#### Connection instances in the AADL instance model

A connection instance in a fully specified system model is a connection between two thread instances, a thread instance and a device instance, or a thread instance and a

process instance. These connection instances are actual data or control flows between system instance components. Fully specified system connection instances, referred to as semantics connections, at least one thread is included. In a partial specified system, a connection instance is expanded through the system hierarchy until no more implementation is specified. Therefore, these connections may be connections between system component instances or process component instances. Although these are not semantics connections, they are essential for analysis.

AADL declarative specifications are blueprints for building system instance models. A connection instance is equivalent to several connection declarations in an AADL declarative model. For example, a semantic connection between two threads of two processes is instantiated from at least three connection declarations: one from the source thread to its process, one from the source process to the destination process, and one from the destination process to its thread subcomponent. The following figure taken from (18) provides an example explaining this concept.

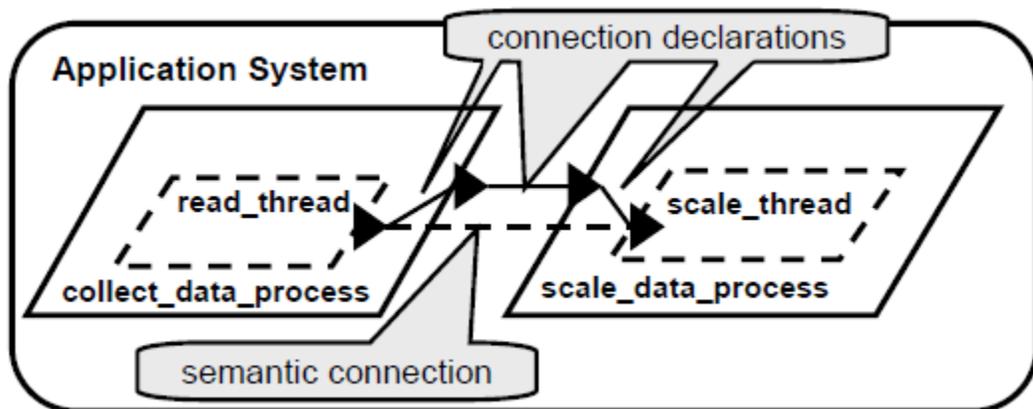


Figure 9 - semantic connection and connection declarations (12)

There are three types of connection instances: mode transition connection instances, port connection instances, and access connection instances (45). Because this research focuses on how data are transferred through the system, it examines port and access connection instances.

#### *Port connection instances*

Port connections are explicit relationships between ports and port groups of AADL components for transferring data and events. The three types of ports are data ports, event ports, event-data ports, all of which have corresponding port connections. Port connection instances are instantiated from port connection declarations to represent connections between two components at the lowest level in the system instance hierarchy. The two ends of the port connection instances are referred to as feature (port) instances.

Data pass through connections between data ports and event-data ports. An out data port connects to only one in data port; the relationship between the two is represented as 1:n, meaning that the data port connections allows “fan-out” only. To accept both “fan-in” and “fan-out,” an event-data port connection is needed. Data can be transferred sequentially from many out event-data ports to one event-data port by controlling the events.

The following figures shows the possible data ports and data event port connections as well as the ITGs representing how data are passed through the inputs to the main component to its output. Incoming data connection ports are represented in ITG by an AND mode transition only while incoming data event connections are shown in both AND and OR modes.

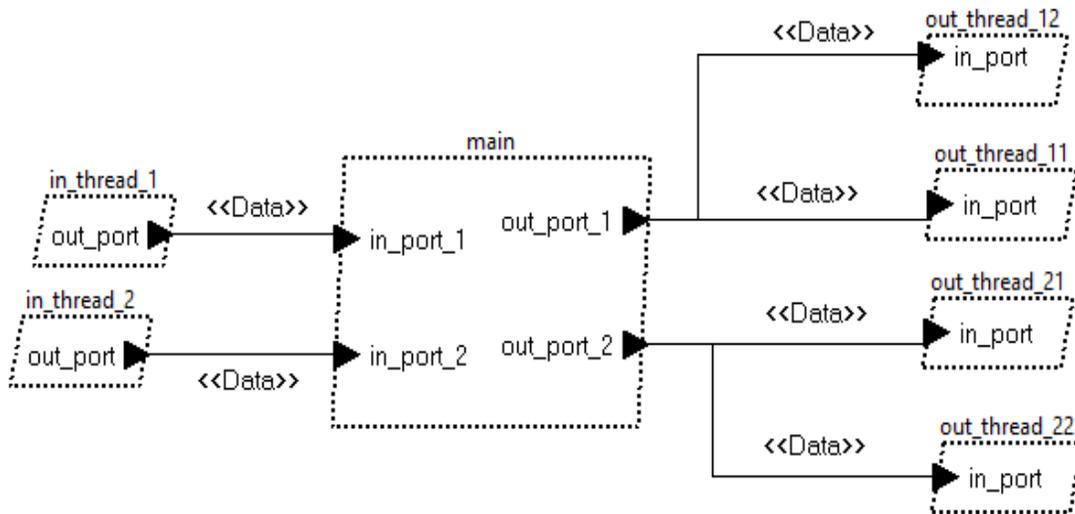


Figure 10 -typical data port connections

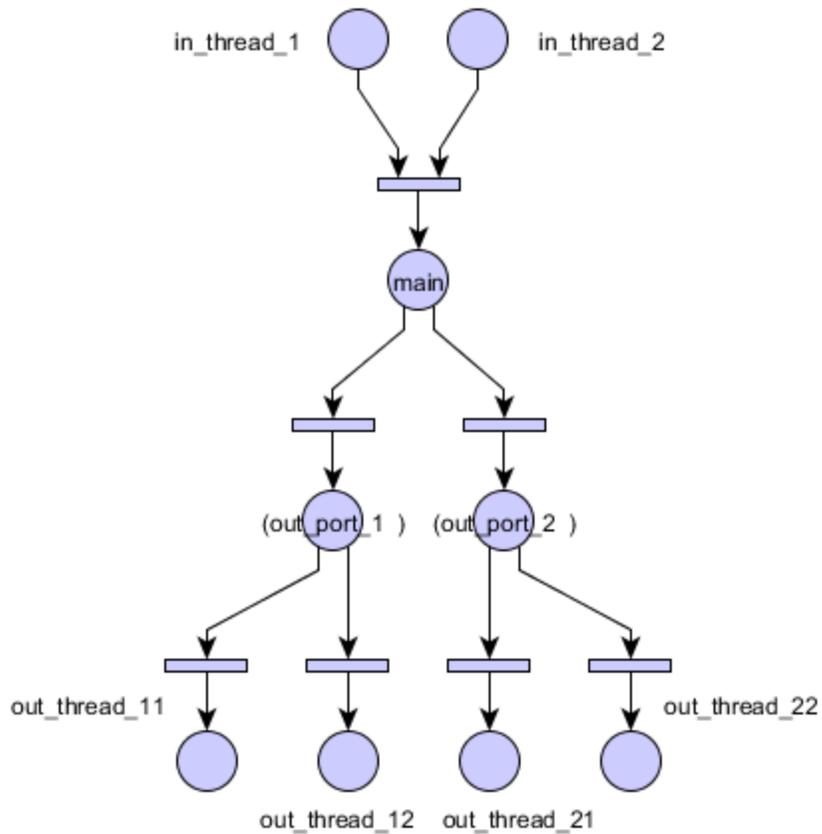


Figure 11 - ITG of the piece of system in figure 10

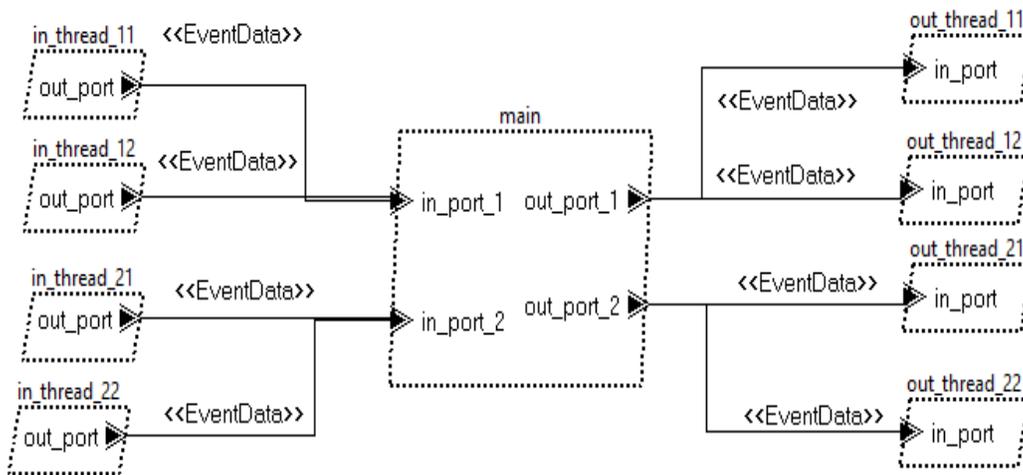


Figure 12 - typical event data port connections

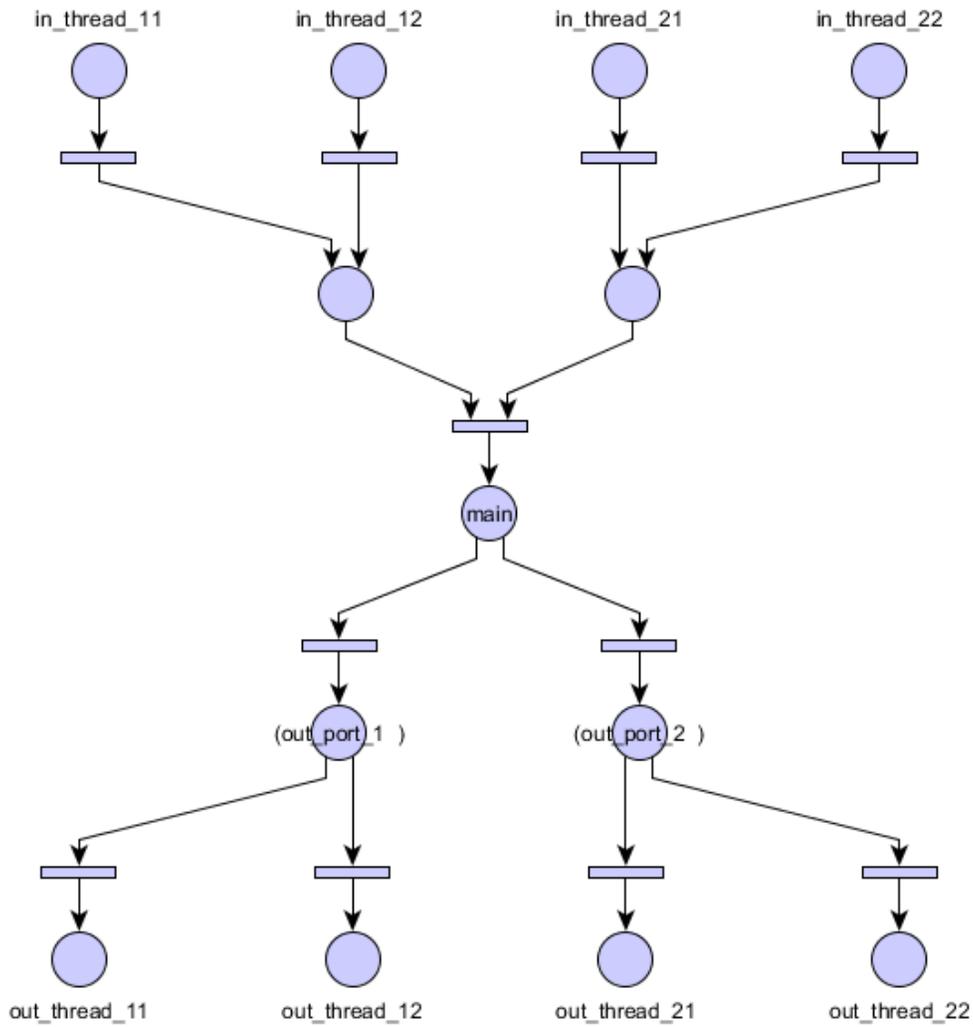


Figure 13 - ITG of the piece of system in figure 12

*Access connection instance*

Component access connections allow multiple components access to shared data and shared buses. A data access connection provides a path from shared data or an access data port to a required access data port. Similar to port connection instance, an access connection instance is instantiated by expanding the port connections through the system

hierarchy so that the access connection instance connects the shared data with the required data access port instance of a leaf component in the system instance hierarchy. In an access connection instance, data pass in either direction beginning from the shared data instance and moving to the required data access port instance of a component instance or vice versa depending on the type of permission given to the component requiring access. If the permission is read\_only, data are passed from the shared data to the component. On other hand, if the permission is write\_only, data are passed from the component to the shared data and if the permission is read\_write, data can be passed in either direction.

#### Converting the AADL instance model into ITG

An efficient method for converting the AADL instance model into an ITG disregarding the type of connection instances is needed. For easy processing when building an ITG for this model, both data instance and required data access of a component are considered to be feature instances of that component and an access connection instance is treated as a port connection instance. The exception is the direction of the connection, which is the direction the data are passing as discussed above.

In (40), the author proposed the final specific model of an electronics device as an ITG modeling the origins of the information transferred as the inputs of the device and the destinations as the outputs. This model was used to analyze the inputs and outputs of such a device in an electronics board based on its interaction with its adjacent devices. This thesis adapts this model to use for converting the AADL instance model to an ITG.

The final specific model, an ITG, shows how information is transferred between components in a system instance and between components and system primary inputs and outputs. Its sources represent the origins of information that the component receives and the sinks represent where the post-processing information produced by the component is sent. In a final specific model, modules and transitions are added to represent how data are transferred from the sources to the sinks, with the module representing the component itself.

To build a final specific model for a component, functional inputs and functional outputs of the component are analyzed, a process involving the concepts of the input element, input group, input hypergroup, input supergroup, output element, output group.

Those concepts are defined as below in the context of one component:

- Input element: the set of all connections to the component originating from the same source component.
- Input group: a set of all input elements with the same set of destination ports.
- Input hypergroup: a set of all input groups containing overlapping source ports
- Input supergroup: a set of disjointed input groups in an input hypergroup so that their information is equal to the the information of the input hypergroup.

Based on these definitions, the functional input of a component is a junction of all its input hypergroups, and an input hypergroup is the attribution of all its input supergroups. An input supergroup is a junction of all input groups, and an input group is an attribution of all its input groups.

- Output element: a set of all output connections with the same destination component.
- Output group: a set of output elements with the same source ports.

Functional inputs analysis of a component begins by converting the functional input of a module into the junction of its input hypergroups. Each input hypergroup is then converted to the attribution of its input supergroups. Next each input supergroup is converted into the junction of its input groups, and finally, each input group is converted into the attribution of its input elements.

Functional outputs analysis is conducted by simply converting the functional outputs of a component into a selection of output groups. Both the functional input and output analyses result in the final specific model for a component. The following example focuses on building the final specific model for a process component referred to as “main.”

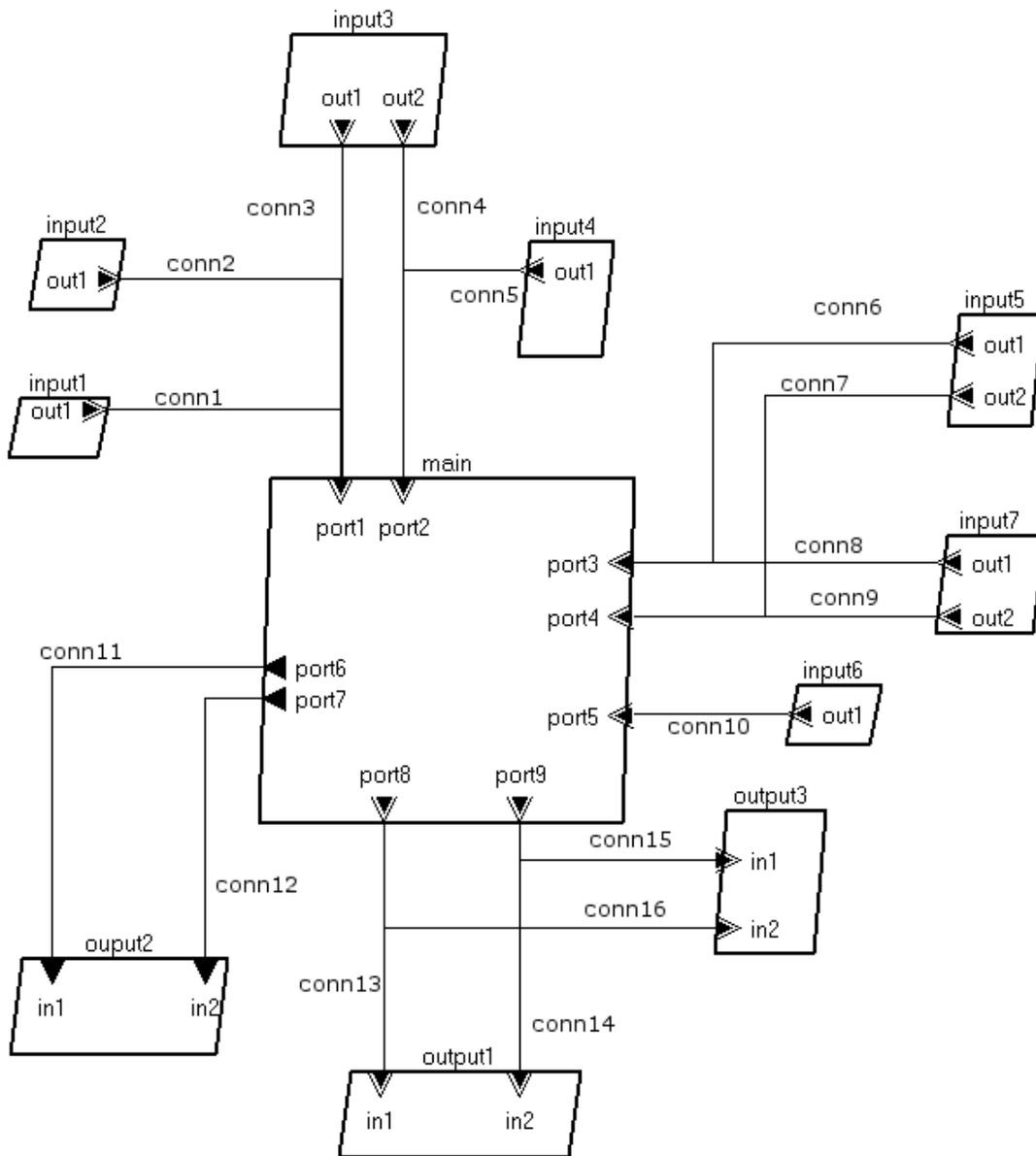


Figure 14 - sample AADL model to be converted to ITG

The process main has 10 incoming connections originating from 7 processes and 6 outgoing connections to 3 different processes. The analysis of its functional inputs and outputs indicates

- 6 input elements:

$$IE1 = \{\text{conn1}\}$$

$$IE2 = \{\text{conn2}\}$$

$$IE3 = \{\text{conn3}, \text{conn4}\}$$

$$IE4 = \{\text{conn5}\}$$

$$IE5 = \{\text{conn6}, \text{conn7}\}$$

$$IE6 = \{\text{conn8}, \text{conn9}\}$$

$$IE7 = \{\text{conn10}\}$$

- 5 Input groups

$$IG1 = \{IE1, IE2\}$$

$$IG2 = \{IE3\}$$

$$IG3 = \{IE4\}$$

$$IG4 = \{IE5, IE6\}$$

$$IG5 = \{IE7\}$$

- 3 Input hypergroups

$$IHG1 = \{IG1, IG2, IG3\}$$

$$IHG2 = \{IG4\}$$

$$IHG3 = \{IG5\}$$

- 4 Input supergroups

$$IHG1: ISG1 = \{IG1, IG3\}, ISG2 = \{IG2\}$$

$$IHG2: ISG3 = \{IG4\}$$

$$IHG3: ISG4 = \{IG5\}$$

- 3 Output Elements

$$OE1 = \{\text{conn11}, \text{conn12}\}$$

$$OE2 = \{\text{conn13}, \text{conn14}\}$$

$$OE3 = \{\text{conn15}, \text{conn16}\}$$

- 2 Output groups

$$OG1 = \{OE1\}$$

$$OG2 = \{OE2, OE3\}$$

The final specific model for the component main is a combination of the functional input analysis and functional output analysis:

$$\begin{aligned} FI &= IHG1 * IHG2 * IHG3 \\ &= (ISG1 + ISG2) * ISG3 * ISG4 \\ &= ((IG1 * IG3) + IG2) * IG4 * IG5 \\ &= ((IE1 + IE2) * IE4) + IE3 * (IE5 + IE6) * IE7 \\ FO &= OG1 + OG2 \end{aligned}$$

The figure below shows the resulting final specific model for the process main.

The ITG of the entire AADL system instance is built by concatenating the final specific models of all the component instances. This process is accomplished by merging the output groups of each final specific model with the corresponding input elements.

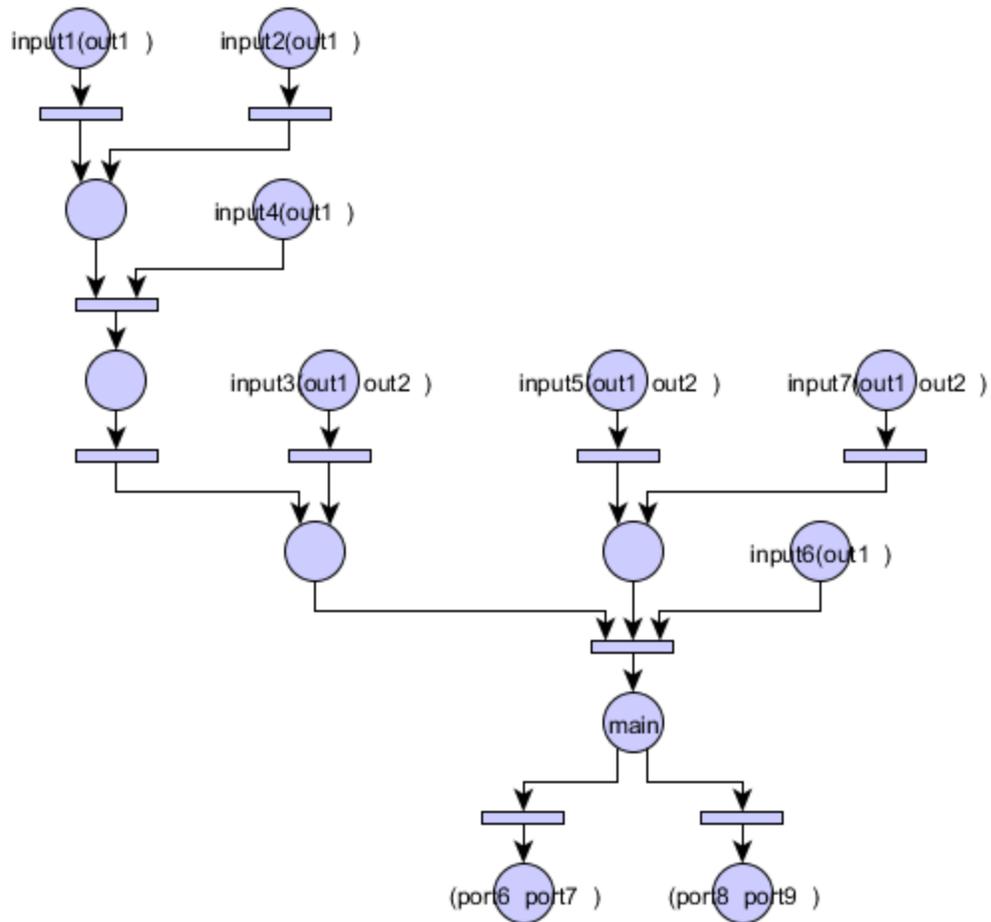


Figure 15 - result final specific model for “main” process from above AADL model

### Calculating the upper bounded testability

After the ITG for the AADL instance model is determined, all its flows are found and the upper bounded testability for each component instance is calculated. A flow F is a sub-network of the ITG if

- A place (source, sink or module) belongs to F, then F admits at least one upstream transition and a downstream transition in the ITG
- A transition is in F, then every place adjacent to it in the ITG also belongs to F

- Regardless of the position of a module  $x$  in  $F$ , there exists in  $F$  a path (in the sense of graph theory) from a source to a sink that goes through the module  $x$ .

All the flows are obtained by considering successively all the sinks of the system.

The upper bounded testability measure involves estimating the upper bounded transinformation. The original information transfer graph of each flow needs to be transformed into a form appropriate for the application of the min-cut, max-flow algorithm to compute the transinformation. A detailed discussion of the feasibility of this method can be found in (43). Here only the implementation is shown. The ITG transformation includes three steps: converting each module which has module capacity into ones with no module capacity information, adding a supersource, and adding a supersink.

In the first step, module  $M$  with capacity  $C(M)$  becomes 2 modules  $M'$  and  $M''$ , both of which have no knowledge of module capacity. The original module capacity becomes the line capacity in the arc from  $M'$  to  $M''$ . All connections coming to  $M$  have  $M'$  as their destination, while all outgoing connections of  $M$  have  $M''$  as their source instead of  $M$ . Now all transitions, modules, sources, and sinks have identical roles because they are simply vertexes of a graph.

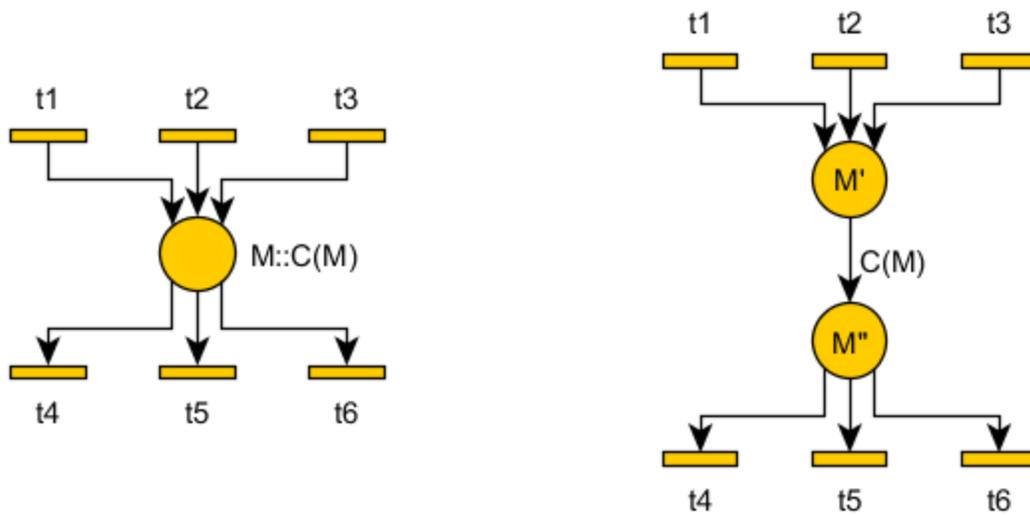


Figure 16 - a module before and after transformation

In the second step, if there is more than one source, a supersource SS is added and connected to all other sources by arcs with infinite line capacity. If this is not the case, SS is the only source of the flow.

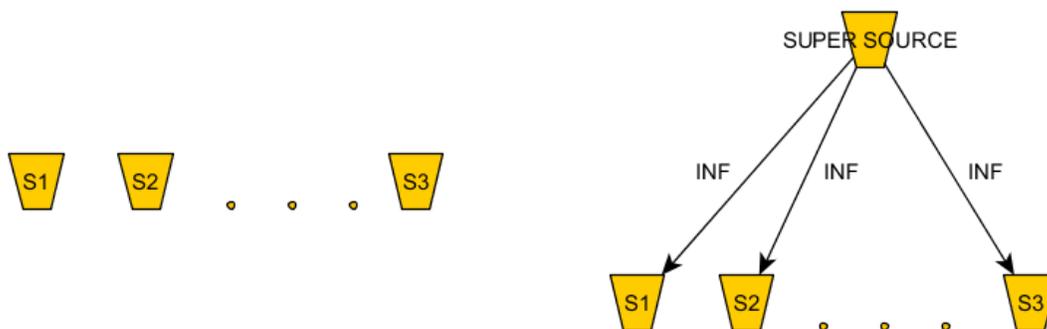


Figure 17 - super sources with infinite line capacity outputs added to existing ITG

If there is more than one sink, a supersink SO is added and connected to all sinks by infinite line capacity arcs. Otherwise, SO is the only sink of the flow.

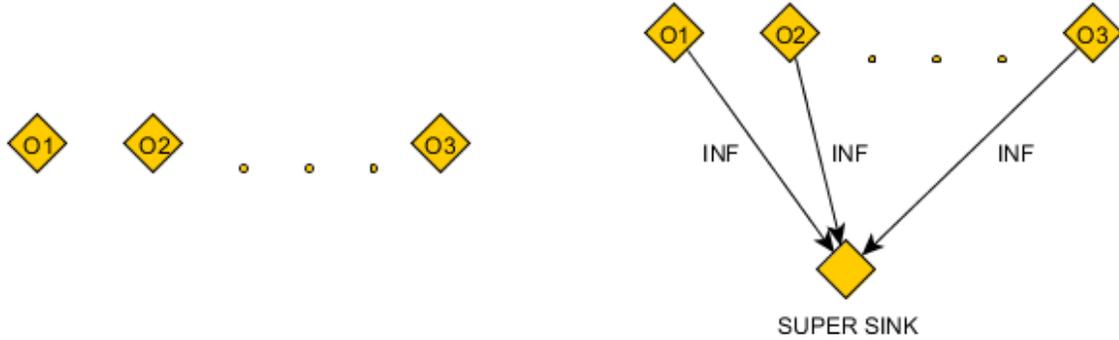


Figure 18 - super sink with infinite line capacity inputs added to existing ITG

Once the ITG for each flow is transformed, transinformation between any module and source as well as between any module and sink can be computed. The variables in flow  $F$  are denoted as follows:

- $X_M$ : inputs of module  $M$
- $Y_M$ : outputs of module  $M$

To compute transinformation  $T'(SS, XM)$ , a fictional arc  $U_r$  having infinite line capacity from  $M'$  to the super sink is added; the min-cut, max-flow algorithm is then applied to compute the maximum flow from the supersink to the supersource. The transformation between the sources and  $T'(SS, XM) =$  the value of the flow through  $U_r$ .

This calculation represents the maximum information that can be transferred from all sources to the module. Connecting the module directly to the supersink bypasses all other intermediate nodes. As a result, the amount of information at the supersink is same as the module  $M$  received from the sources. This method for computing the transinformation  $T'(SS, XM)$  is different from that in (43) as it does not require reversing all the flow edges, making it faster and simpler.

Transinformation  $T'(Y_M, SO)$  is computed by adding fictional arc  $U_r$  from the super- source to  $M'$ , then applying the min-cut, max-flow algorithm. The value of  $T'(Y_M, SO)$  is the value of the flow through  $U_r$ . Similarly, this process calculates the maximum amount of information that can be sent from a module to all the flow sinks. Adding the fictional arc from the supersource to  $M'$  allows for the sending of the maximum amount of information to the input of the module. As a result, the flow value for  $U_r$  is the transinformation between module  $M$  and super sink  $SO$ .

After obtaining the transinformation value between a module and all the sources, controllability of the module is computed as

$$CO(M) = \frac{T'(SS, X_M)}{C(X_M)}$$

where  $C(X_M)$  is the sum of the line capacities of all arcs coming to  $M'$

Similarly, observability of a module is computed after obtaining the transinformation value between the module and the flow sinks:

$$OB(M) = \frac{C(Y_M)}{T'(Y_M, SO)}$$

where  $C(Y_M)$  is  $C(M)$

Testability is computed as the product of the observability and controllability values

$$TE(M) = CO(M) \times OB(M)$$

### Channel capacity calculation

A testability measure requires knowledge of the channel capacity, e.g. the module capacity and the line capacity. The line capacity is obtained based on the type of data connections between components; for example Boolean connections have a capacity of 1 bit while integer connections may have a capacity of 32 bits. The module capacity can be obtained through statistics or by exact calculation. These two methods require the execution of a module or require module functionality provided by a truth table. (17). However, at the architecture level, a component implementation might yet be unknown. Therefore, a more efficient way to estimate the module capacity is needed. In addition to a testability measure, this thesis also proposes a method for measuring the module capacity of system software. These module capacity values are useful when a new system is built based on existing systems such as system of systems architecture or system product lines.

If there is no detailed information about the implementation of a component, the approximate module capacity can be approximated by summing the line capacities of all the output connections of the component. However, component implementation provides a more precise estimation of module capacity; the more detailed it is, the more precise the module capacity estimation.

By definition, module capacity is the maximum transinformation between its inputs and outputs over all possible distributions of its inputs. The exact computation of the transinformation value is complex and not necessary at the architecture level. Therefore, to estimate module capacity, the upper bounded transinformation is calculated

instead of the exact transinformation calculation. The same method used to calculate testability described previously is applied here. First, an ITG is built for the component, considering all inputs of the component as sources and all outputs as sinks. A supersource and supersink are then added to the graph. Next, the ITG is transformed such that the min-cut, max-flow algorithm can be applied to calculate the maximum flow from the supersource to the supersink. The upper bounded transinformation between the sources and sinks is the value of maximum flow in the graph. This value is also the estimated value of module capacity.

## CHAPTER FOUR

### EXPERIMENTAL RESULTS

#### Defining the new property set

Because testability measurement requires information about sources, sinks and channel capacities, a new AADL property set is defined as the following:

```
property set testability is
  LineCapacity: aadlreal applies to (connections);
  ModuleCapacity: aadlreal applies to (system, process, device, thread);
  isSource: aadlboolean applies to (system, process, device, thread);
  isSink: aadlboolean applies to (system, process, device, thread);
end testability;
```

**Figure 19 - testability property set definition**

Source components, which are fully controlled so that all possible inputs to test the system can be generated, can be another system, an external device or even a process or thread. Similarly, a sink can be another system, device, process, or thread that receives output from the system being tested; thus, it provides the ability to fully observe the outputs from the system.

The channel capacity property is optional for each component. If not specified, by default the channel capacity is defined as the maximum value. For simplification, in this thesis channel capacity, e.g. line capacity and module capacity, is considered a real value, the unit of which can be implicitly understand as a bit, byte, or MB.

#### Example 1

This chapter analyzes two examples to show how the proposed plugin works. It begins with following AADL specifications:

```

system board
end board;

system implementation board.impl
subcomponents
  reg1: process reg1;
  reg2: process reg2;
  incr: process incr;
  decr: process decr;
  REG: process REG;
  input: device input;
  input2: device input2;
  input3: device input3;
  output: device output;
connections
  event data port input.D03 -> REG.E03 {
    testability::ChannelCapacity => 4.0;
  };
  event data port input.D47 -> REG.E47 {
    testability::ChannelCapacity => 4.0;
  };
  event data port reg2.Q03 -> REG.E03 {
    testability::ChannelCapacity => 4.0;
  };
  event data port reg1.Q03 -> REG.E47{
    testability::ChannelCapacity => 4.0;
  };
  data port REG.S03 -> decr.E03{
    testability::ChannelCapacity => 4.0;
  };
  data port REG.S47 -> incr.E03{
    testability::ChannelCapacity => 4.0;
  };
  data port REG.S03 -> output.B03{
    testability::ChannelCapacity => 4.0;
  };
  data port REG.S47 -> output.B47{
    testability::ChannelCapacity => 4.0;
  };
  data port decr.S03 -> reg2.D03{
    testability::ChannelCapacity => 4.0;
  };
  event data port incr.S03 -> reg1.D03{
    testability::ChannelCapacity => 4.0;
  };
  data port input2.R -> reg1.A{
    testability::ChannelCapacity => 1.0;
  };
  data port input2.R -> reg2.A{
    testability::ChannelCapacity => 1.0;
  };
  event data port input3.I03 -> reg1.D03{

```

Figure 20 - Sample model (part 1)

```

        testability::ChannelCapacity => 4.0;
    };
    properties none ;
end board.impl;

process reg1
    features
        D03: in event data port;
        Q03: out event data port;
        A: in data port;
    end reg1;

process reg2
    features
        D03: in data port;
        Q03: out event data port;
        A: in data port;
    properties
        testability::ModuleCapacity => 1.7;
    end reg2;

process decr
    features
        E03: in data port;
        S03: out data port;
    properties
        testability::ModuleCapacity => 3.2;
    end decr;

process incr
    features
        E03: in data port;
        S03: out event data port;
    end incr;

process REG
    features
        E03: in event data port;
        E47: in event data port;
        S03: out data port;
        S47: out data port;
    end REG;

device input
    features
        D03: out event data port;
        D47: out event data port;
    properties
        testability::isSource => true;
    end input;

```

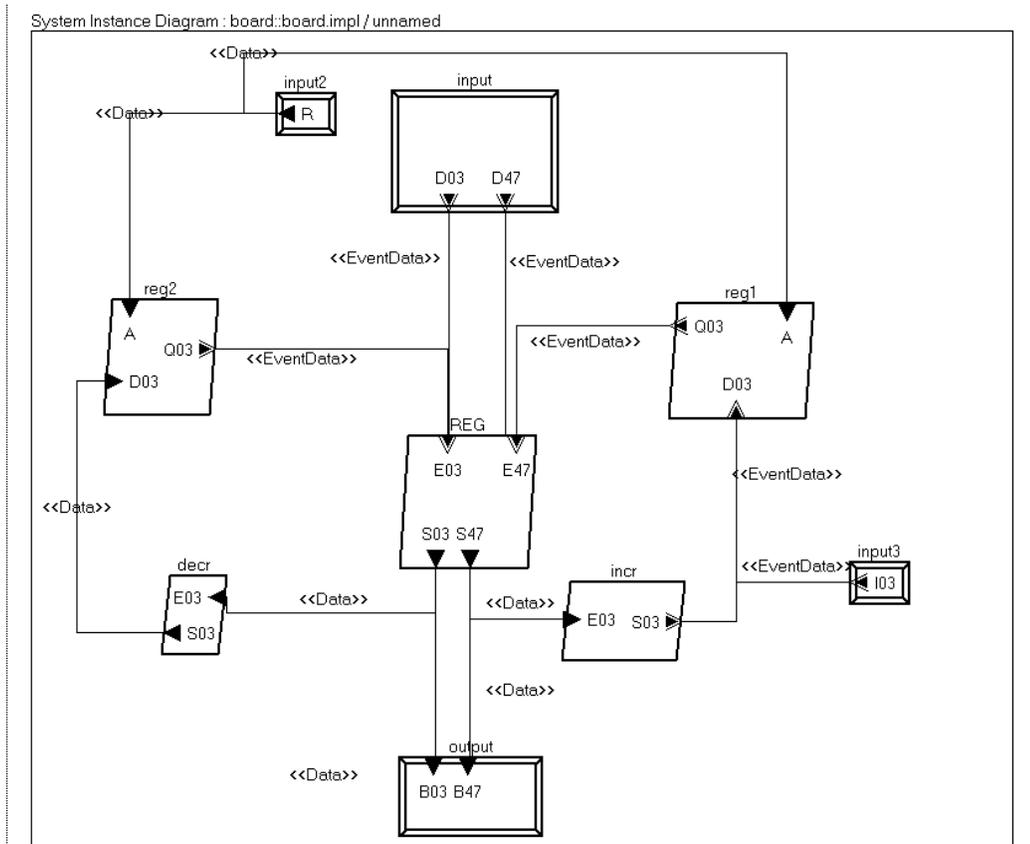
Figure 21 - Sample model (part 2)

```
device input2
  features
    R: out data port;
  properties
    testability::isSource => true;
end input2;

device input3
  features
    I03: out event data port;
  properties
    testability::isSource => true;
end input3;

device output
  features
    B03: in data port;
    B47: in data port;
  properties
    testability::isSink => true;
end output;
```

Figure 22 - Sample model (part 3)



**Figure 23 - The AADL graphical representation of above AADL model**

In the first step, all components in the system implementation are analyzed for their input and output data flows. The final specific model for each is then built, with the line capacity being written on each arc. Each module is denoted in the following form:

`<module_name>[::module_capacity]`

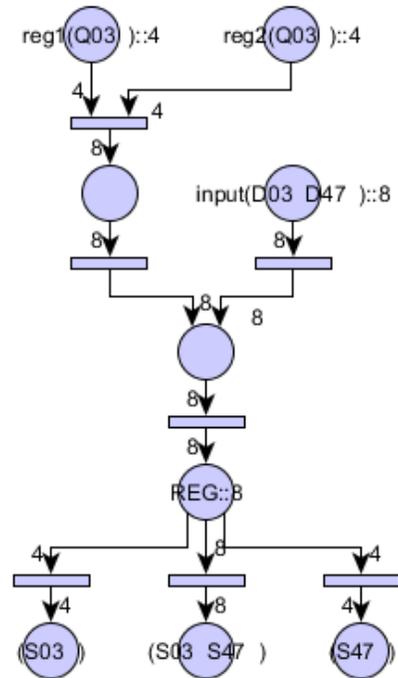
`module_name = [component instance][feature instance | shared data instance]`

For example, `reg1(Q03)::4` means port Q03 of component reg1 which has capacity of 4.

REG:

Data can be injected to REG's inputs from ports D03 and D47 of "input" device (8 bytes in total) or from Q03 port of reg1 (4 bytes) and Q03 port of reg2 (4 bytes).

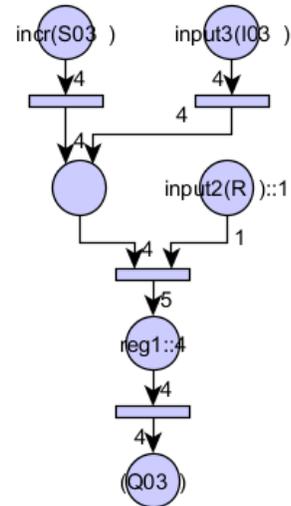
Output data of REG is sent out via port S03 or S47 or both S03 and S47

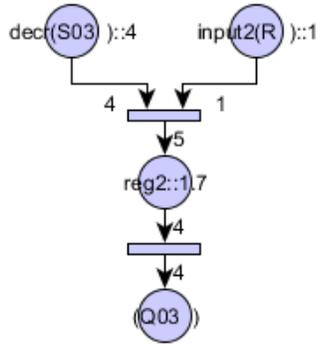
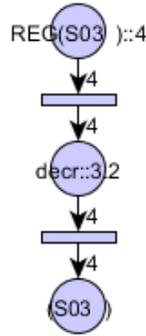
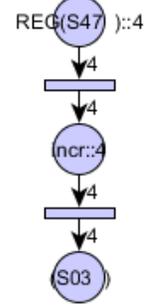
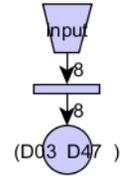


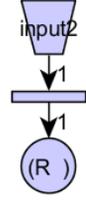
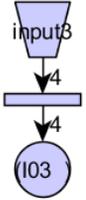
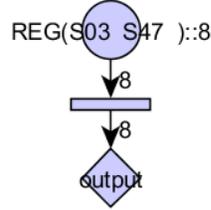
reg1:

Inputs of reg1 can be controlled by (port R of input2 and port S03 of incr) or port I03 of input3.

Output of reg1 is sent out via port Q03 only



<p><u>reg2</u></p> <p>Input data of reg2 is sent from port S03 (4 bytes) of decr and port R (1 byte) of input2. Reg2 has module capacity of 1.7 and it has only one output element Q03.</p>	
<p><u>Decr</u></p> <p>Decr receives data sent from port S03 of REG and sends data out via port S03. Decr's module capacity is 3.2 bytes</p>	
<p><u>Incr</u></p> <p>Incr receives data sent from port S47 of REG and sends data out via port S03</p>	
<p><u>Input</u></p> <p>“Input” is a source, sending data out via port D03 and D47</p>	

<p><u>Input 2</u></p> <p>Input 2 is a source that sends only 1 byte through port R</p>	
<p><u>Input 3</u></p> <p>Input3 is a source that sends 4 bytes out via port I03</p>	
<p><u>Output</u></p> <p>Output is a sink that receives 8 bytes of data from port S03 and S47 of REG</p>	

These final specific models are then concatenated to produce the ITG for the system as shown in Figure 24. The 3 flows in Figure 25 are derived from the ITG.

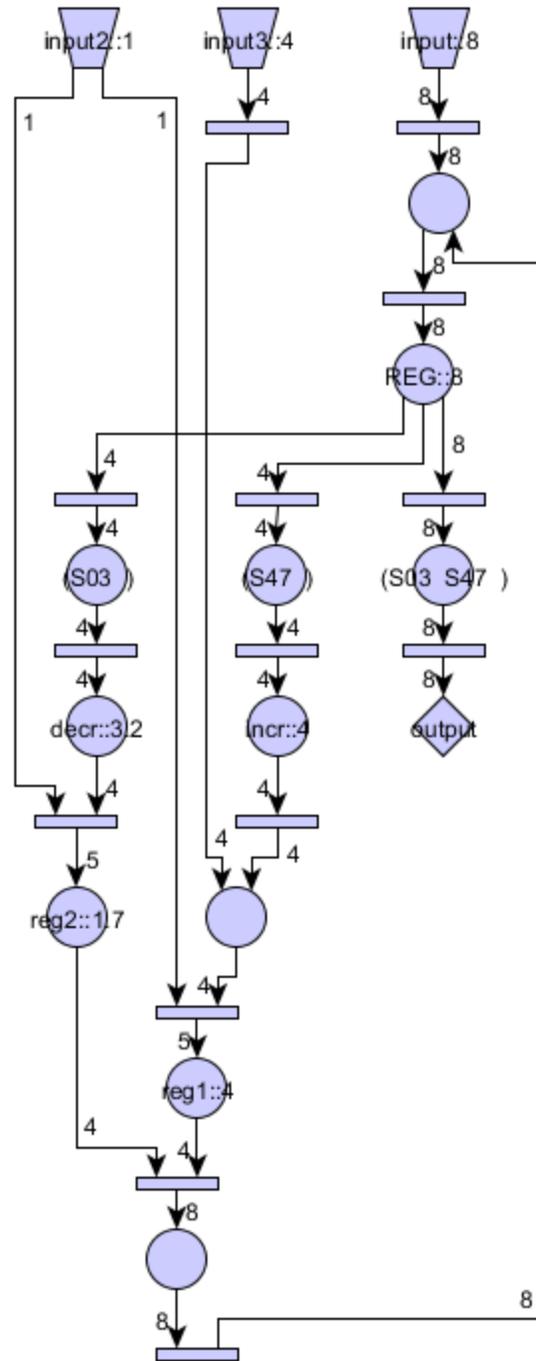


Figure 24 - ITG of the example system

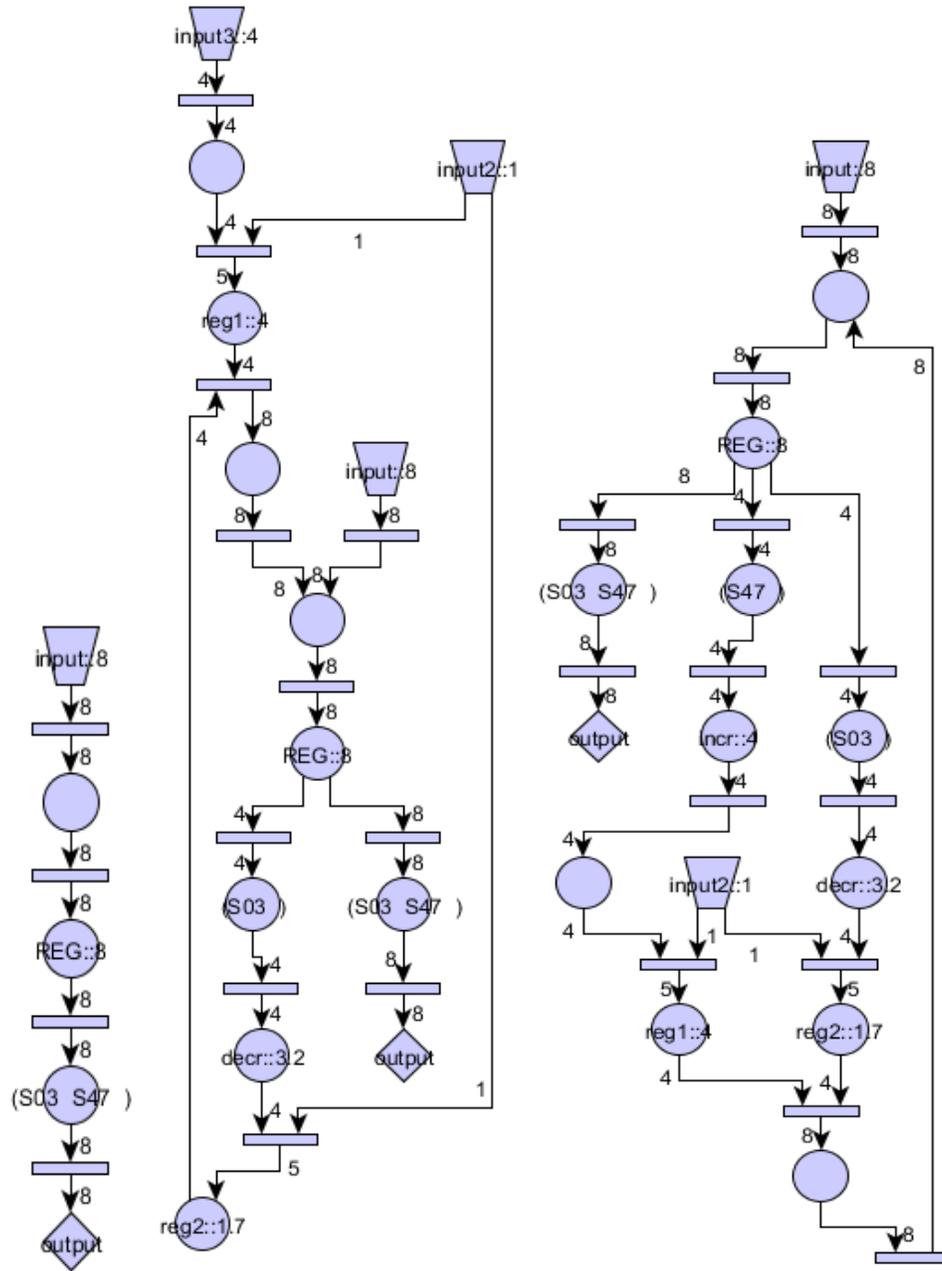


Figure 25 - three generated flows numbered 1-3 from left to right

In flow 1, all input data can be sent to output through REG. Therefore, controllability, observability, and testability of REG is 1. Flow 2 and flow 3 must be transformed into a form appropriate for the application of the min-cut, max flow algorithm. Figure 23 shows this transformation. The module capacities of reg2 and decr

are specified in the given AADL models, while the module capacities of the other components are derived from the line capacity of its outputs. The testability calculation for *decr* in flow 2 and *reg2* in flow 3 are described in detail below.

To calculate the controllability of *reg2* in flow 3, this formula is used:

$$CO(reg2) = \frac{T'(SS, C_X(reg2))}{C_X(reg2)}$$

where  $T'(SS, C_X(reg2))$  is the transinformation between all the sources of flow 3 and inputs of *reg2* and  $C_X(reg2)$  is the sum of the channel capacity of all the incoming connections of *reg2*. Here,  $C_X(reg2) = 5$ .

$T'(SS, C_X(reg2))$  is calculated by:

- Connecting all the sources of flow 3 to the supersource.
- Adding a fiction edge with infinite line capacity to connect *reg2* to the supersink.
- Applying the Ford-Fulkerson algorithm (46) to find the max flow value from the supersource to the supersink.
- The value of the flow on the fiction edge is  $T'(SS, C_X(reg2))$

Figure 24 exemplifies this process for the calculation of  $T'(SS, C_X(reg2))$  on flow 3. The pair of numbers (x/y) on each edge shows the flow value and the line capacity.

Based on this calculation,  $T'(SS, C_X(reg2)) = 4.2$ . Therefore,

$$CO(reg2) = 4.2 / 5 = 0.84$$



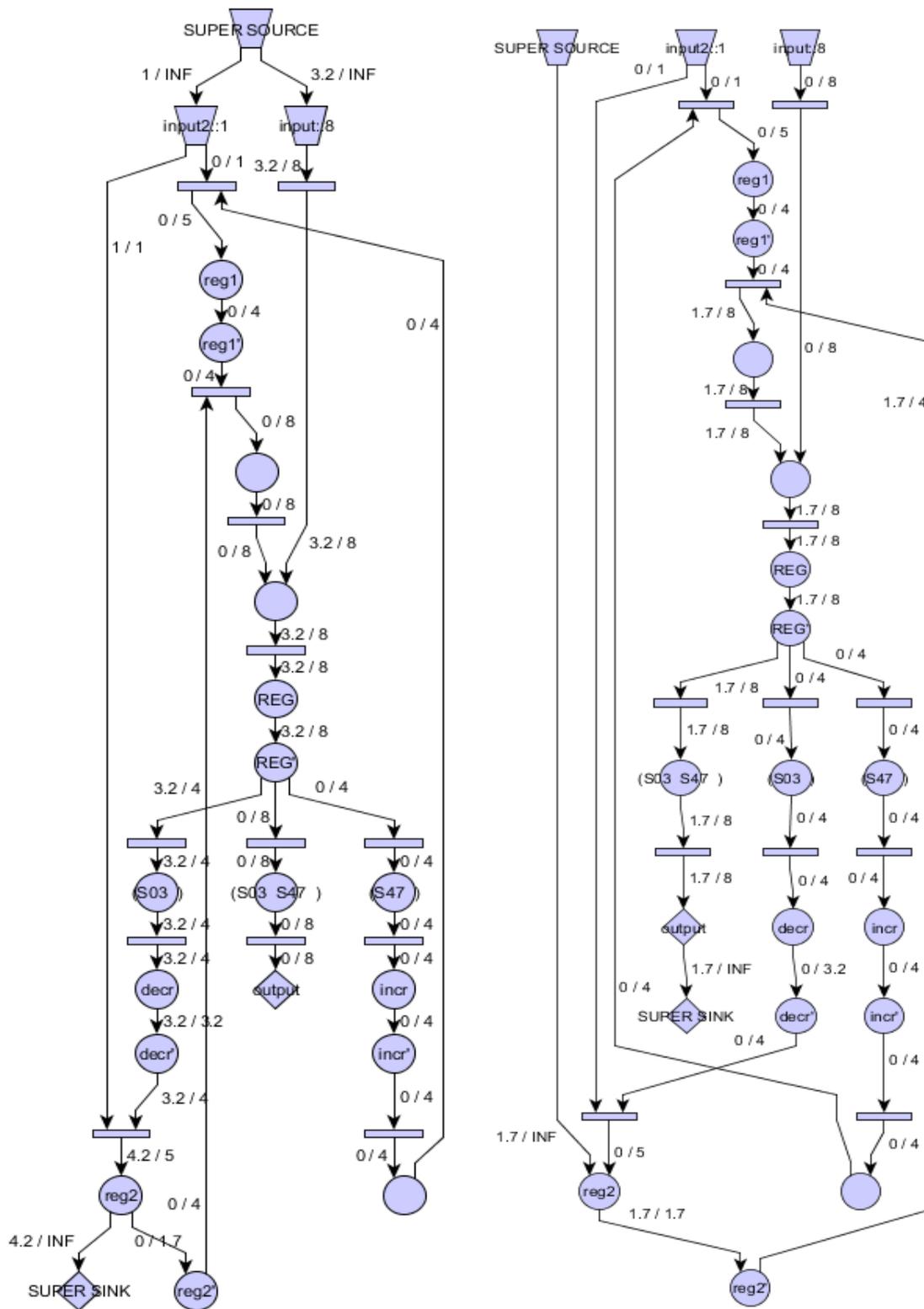


Figure 27 - demonstration of measuring  $T'(SS, reg2)$  (left) and  $T'(reg2, SO)$  (right) on flow 3

To calculate the observability of reg2 in flow 3, the following formula is used:

$$OB(REG) = \frac{T'(C_Y(reg2), SO)}{C_Y(reg2)}$$

where  $T'(C_Y(reg2), SO)$  is the transinformation between all the sinks in the flow and the outputs of REG and  $C_Y(reg2)$  is the module capacity of reg2.

$$C_Y(reg2) = 1.7$$

$T'(C_Y(reg2), SO)$  is calculated by

- Connecting all the sinks to the supersink.
- Adding a fiction edge with a infinite line capacity to connect the supersource to REG
- Applying the Ford-Fulkerson algorithm to find the max flow value from the supersource to the supersink.

The value of the flow on the fiction edge is  $T'(C_Y(reg2), SO)$ . Figure 24 illustrates the calculation of  $T'(C_Y(reg2), SO)$  on flow 3.

$T'(C_Y(reg2), SO) = 1.7$ . Therefore,

$$OB(reg2) = 1.7 / 1.7 = 1$$

As a result,  $TE (reg2) = CO(reg2) \times OB(reg2) = 0.84$

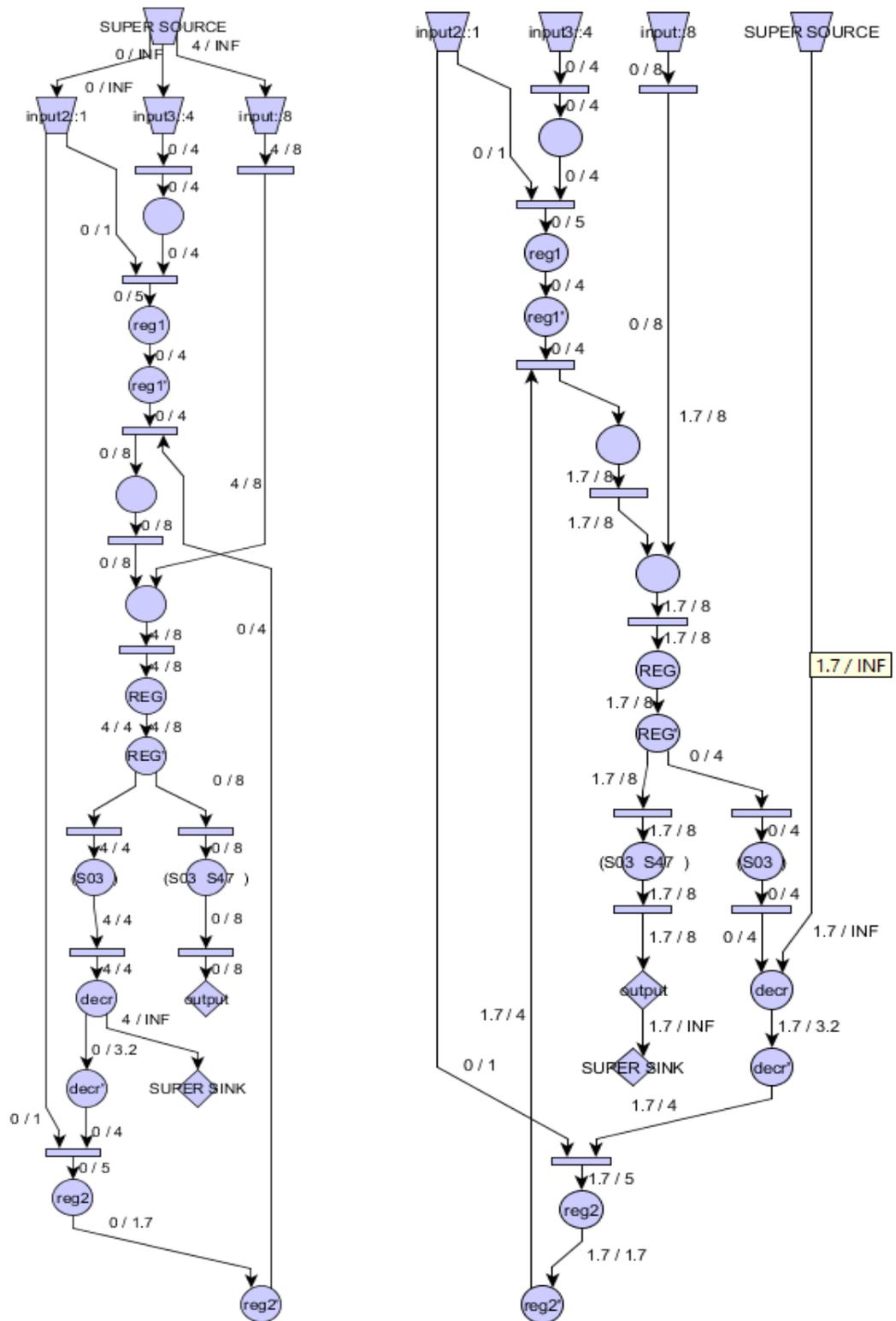


Figure 28 - demonstration of measuring  $T'(SS, decr)$  (left) and  $T'(decr, SO)$  (right) on flow 2

Similarly, calculating the controllability of decr in flow 2 uses the following formula:

$$CO(REG) = \frac{T'(SS, C_X(\text{decr}))}{C_X(\text{decr})}$$

- $T(SS, C_X(\text{decr})) = 4$
- While  $C_X(\text{decr}) = 4$
- Therefore,  $CO(\text{decr}) = 4 / 4 = 1$

To calculate the observability of reg2 in flow 2, the formula below is used:

$$OB(\text{decr}) = \frac{T'(C_Y(\text{decr}), SO)}{C_Y(\text{decr})}$$

- $T'(C_Y(\text{decr}), SO) = 1.7$
- While  $C_Y(\text{decr}) = 3.2$
- Therefore,  $OB(\text{decr}) = 1.7 / 3.2 = 0.53125$

As a result,  $TE(\text{decr}) = 0.53125$  in flow 2.

The results of the testability measure for each flow are shown in following tables

**Table 1 – Controllability /observability /testability measure for flow 1**

Flow 1 information:		
=====		
-----		
Controllabilities:	Observabilities:	Testabilities:
-----		
CO(REG) = 1.00000	OB(REG) = 1.00000	TE(REG) = 1.00000

**Table 2 - Controllability /observability /testability measure for flow 2**

Flow 2 information:		
=====		
-----		
Controllabilities:	Observabilities:	Testabilities:
-----		
CO(reg1) = 1.00000	OB(reg1) = 1.00000	TE(reg1) = 1.00000
CO(REG) = 1.00000	OB(REG) = 1.00000	TE(REG) = 1.00000
CO(decr) = 1.00000	OB(decr) = 0.53125	TE(decr) = 0.53125
CO(reg2) = 0.84000	OB(reg2) = 1.00000	TE(reg2) = 0.84000

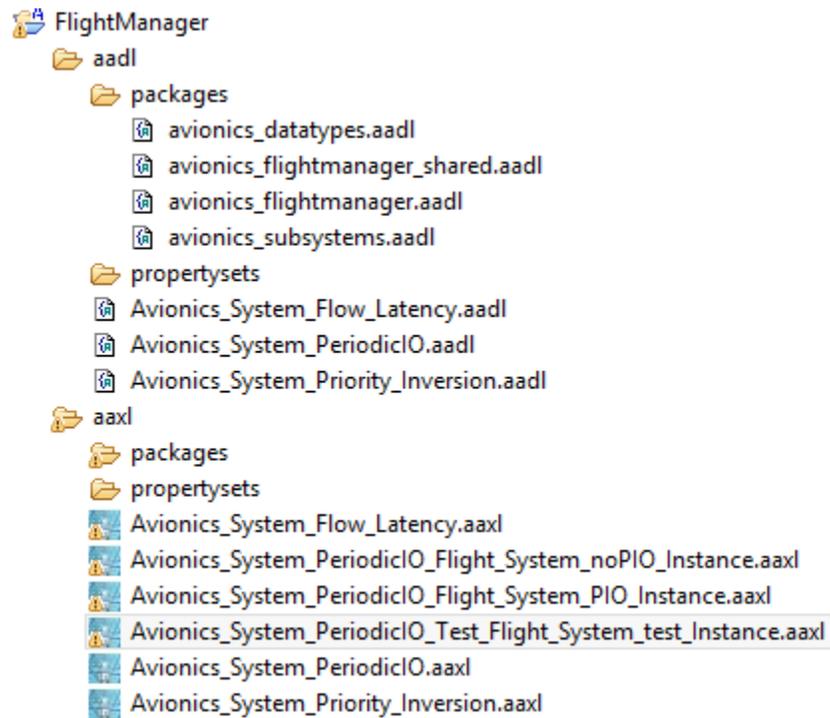
**Table 3 - Controllability /observability /testability measure for flow 3**

Flow 3 information:		
=====		
-----		
Controllabilities:	Observabilities:	Testabilities:
-----		
CO(REG) = 1.00000	OB(REG) = 1.00000	TE(REG) = 1.00000
CO(incr) = 1.00000	OB(incr) = 1.00000	TE(incr) = 1.00000
CO(reg1) = 1.00000	OB(reg1) = 1.00000	TE(reg1) = 1.00000
CO(decr) = 1.00000	OB(decr) = 0.53125	TE(decr) = 0.53125
CO(reg2) = 0.84000	OB(reg2) = 1.00000	TE(reg2) = 0.84000

If the threshold for good testability is set at 0.7 or above, incr and reg1 exhibit perfect testability and reg2 acceptable testability according to these results. However, decr has a poor testability measure because it lacks observability. According to the graphical representation of the system, the testing results can be observed from 8 bytes of outputs generated from REG. These outputs loop back to the inputs of decr and pass through reg2 before returning again through REG and to the outputs. This loop makes the decr outputs difficult to transfer to the primary output of the system. The solution in this case is to increase decr observability by adding observation points either directly at the outputs of decr or somewhere near them.

## Example 2

In this example, testability of an avionics system model taken from (47) is measured. This system model includes two implementations, one for periodic IO and the other for non-periodic IO. Figure 29 shows the full model in OSATE while figure 30 illustrates the system instance of the avionics system with periodic IO under testing. Figure 31 shows the ITG converted from the system instance. Calculation results are in Table 4.



**Figure 29 - Full system model**

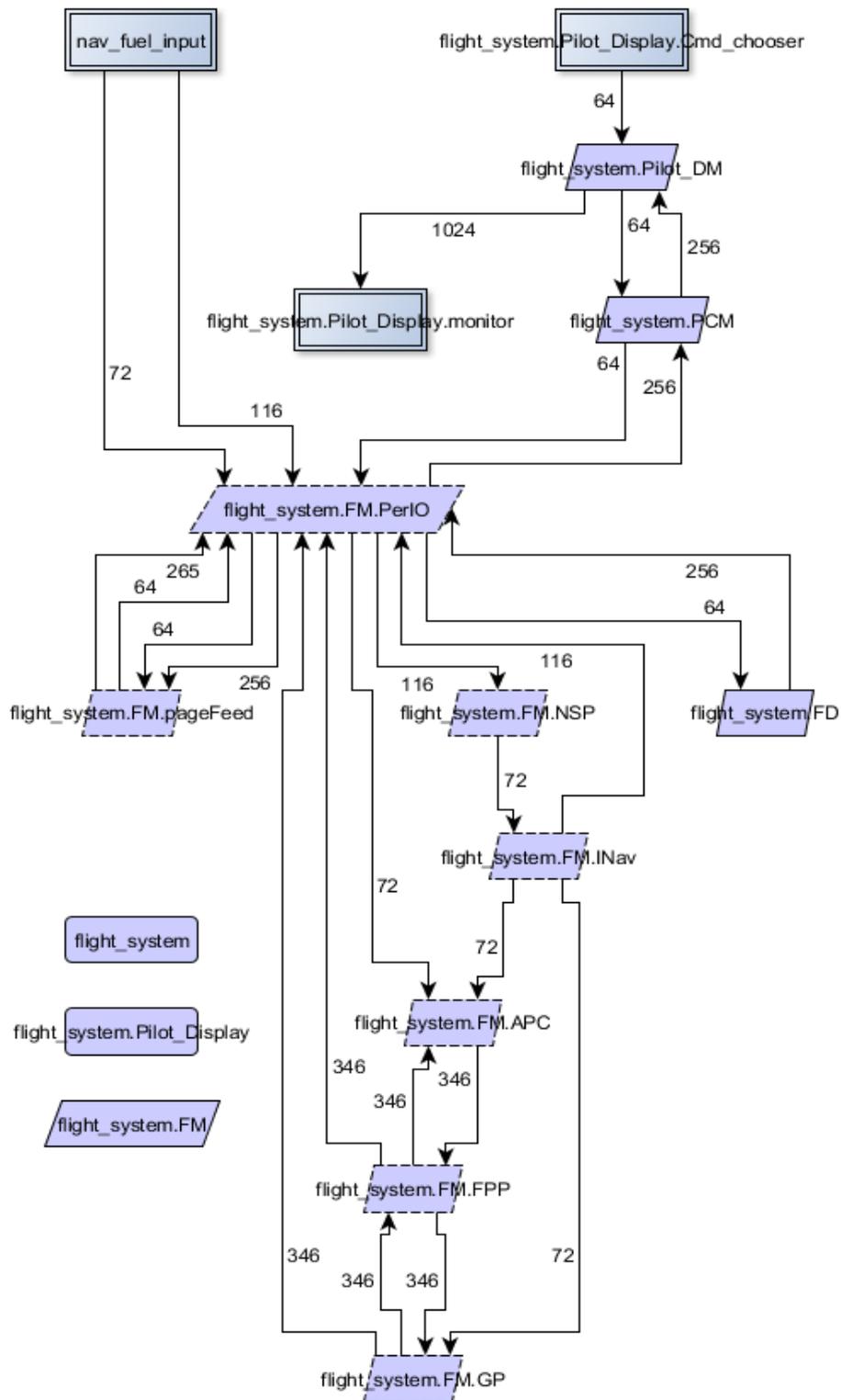


Figure 30 - Avionics\_system\_periodic\_IO under test

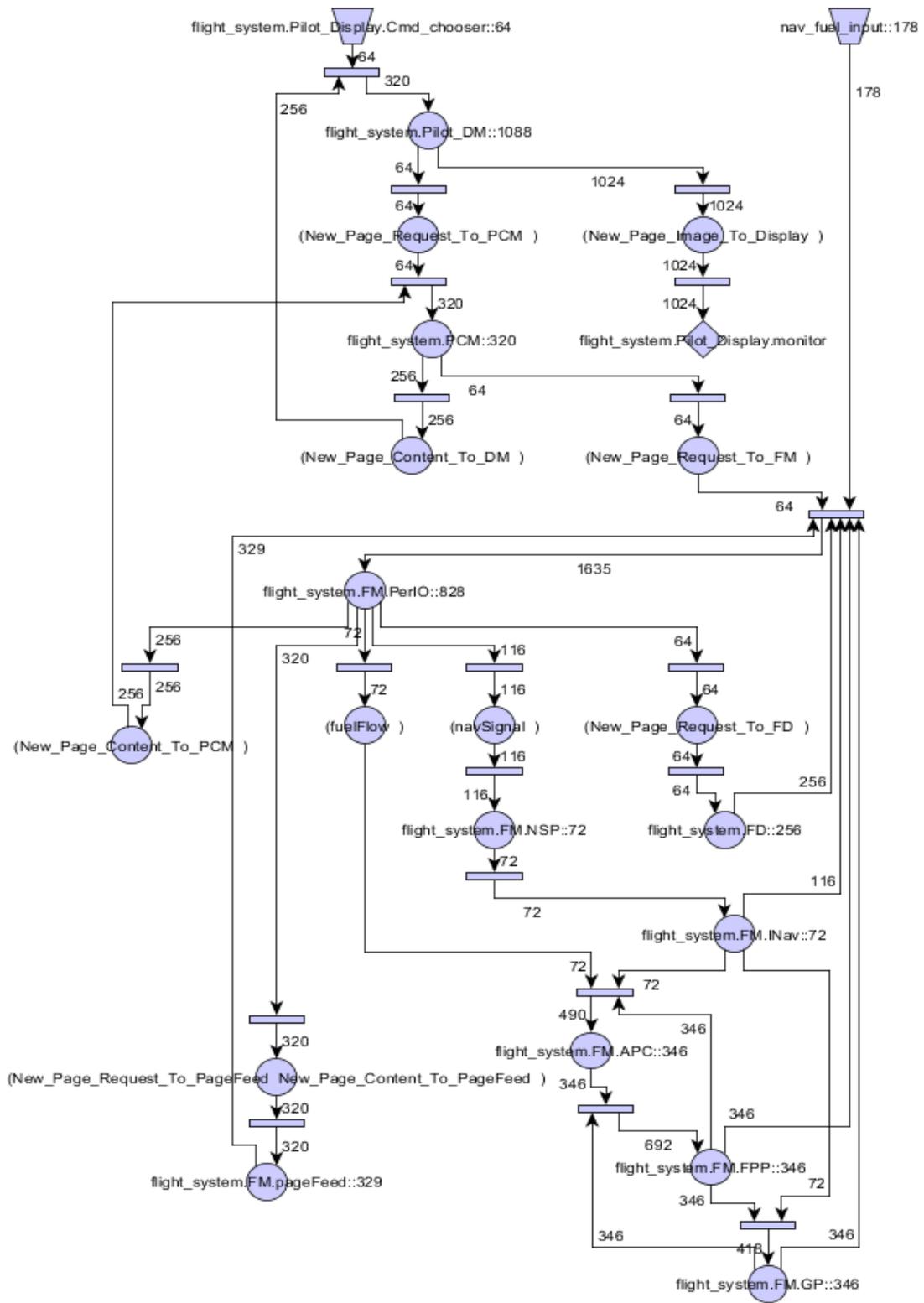


Figure 31 - ITG for the system instance

Component	Controllabilities	Observabilities	Testabilities
flight_system.Pilot_DM	0.76250	0.94118	0.71765
flight_system.PCM	0.76250	0.80000	0.61000
flight_system.FM.PerIO	0.14905	0.30918	0.04608
flight_system.FM.INav	1.00000	1.00000	1.00000
flight_system.FM.GP	0.34450	0.73988	0.25489
flight_system.FM.APC	0.29388	0.73988	0.21744
flight_system.FM.FPP	0.20809	0.73988	0.15396
flight_system.FM.NSP	1.00000	1.00000	1.00000
flight_system.FM.pageFeed	0.76250	0.77812	0.59331
flight_system.FM.FD	1.00000	1.00000	1.00000

**Table 4 - Calculation results for example 2**

The calculation results in Figure 11 suggest if a threshold for good testability is 0.5, the components PerIO, GP, APC and FPP exhibit poor testability primarily because they have poor controllability. This means that testing using only default primary input, e.g. the command menu on the pilot display, navigation data and fuel flow information, is not sufficient for achieving confidence in testing this system. Other inputs are needed in order to test this system.

## CHAPTER FIVE

### RELATED WORKS

#### Testability analysis for data flow software

The most significant research related to this thesis is the method for measuring testability as controllability and observability for data flow of software/hardware proposed by (17,43,44). The method was realized through such tools as SATAN and CATA. This concept, initiated in hardware testing (2), was first used in hardware testing, specifically to analyze the VHDL description of electronic circuits using SATAN (39). These tools also exhibited a potential as a testability measure for software and co-designed systems. In addition, it was successful in analyzing and locating low testability components in critical system such as the AIRBUS A320 aircraft flight management system (5).

Several researchers took advantage of this testability measurement method, including (48)(49) (42)(38). The first two (48)(49) applied this method in analyzing the testability of data flow design in reactive systems, basing it on SATAN and data-flow design approach environments such as SCADE, SLIDEX and SIMULINK. The last two (42)(38) developed a method to measure testability at the source code level by transforming source code into data-flow representation using Static Single Assignment (SSA).

The benefits of these studies are their practical applications. Another benefit is that this method was proposed along with the testing strategies based on the data obtained on flows such as all-path, multiple clue, start-small, and start big. However, the testability

measure tools and the design tools were developed separately. To use the tools, design documents first have to be converted to ITG, which is then fed as input into the testability tools. For example, Le Traon (5) takes the design documents from SAO (or CAS), a design technique developed by AEROSPATIALE, and Huy Do Vu (48) takes the designs from SCADE, SLIDEX and SIMULINK, converts them to ITG, and then feeds them to SATAN to measure testability. Using several tools for the same purpose leads to compatibility issues between the design and measurement tools. Moreover, the design environments that they are based on focus more on the component design level, e.g. at the AND, OR, SWITCH level rather than the architecture level. In addition, these design tools lack non-functional properties analysis, meaning they are not effective for early analysis at the architecture level.

AADL is a standard that is widely used today, in part because it focuses on early analysis at the architecture level. Given its extensible capability, developing a method to measure component testability in AADL models is potentially beneficial. By supporting Model Based Development, the testability measure extension along with other AADL analysis will support the entire system development life cycle.

#### Framework for model-based testing of AADL models

Y Dong (37) proposed a framework based on statistical methods to test an AADL model using the Markov Chain. From the AADL Error models, an expected Markov chain is generated. In addition, an actual Markov chain (AMC) is extracted from the AADL model. From the test pool, n-test-cases are drawn and fed into AMC, resulting in the actual probability of mode transition. This probability value is then compared to the

one drawn from EMC describing the expected behavior of the system software. The difference between the two probabilities is manually evaluated to verify whether errors exist in the AMC. Although this framework supports the testing of AADL models, its purpose is different from that proposed in this thesis. This framework is helpful later in the development process while the aim of the AADL testability plugin proposed in this thesis is early analysis testability of the AADL model.

#### Test case generation based on AADL system component modes

X Lu (36,37) proposed a method supporting testing by generating test cases based on AADL models. It builds a Component Test Model (CTM) to depict system component modes and mode transitions. The CTM is then converted into a test tree to help testers generate test cases to check mode transitions. The author developed an OSATE plugin, the AADL Testing Case Assistant Generation Tool (ATCAG), based on this method. Similar to (37), this method is meaningful in late phases of the software development life cycle only.

## CHAPTER SIX

### CONCLUSIONS AND FUTURE WORKS

Testability, an important software quality attribute, is desired for the ease of software testing. Early analysis of the testability of the system architecture to improve system component testability is essential to reduce development cost and risks. The goal of this thesis was to develop a testability measurement method for AADL models and a plugin for OSATE, an open source AADL tool. AADL is a standard language for architecture description and analysis. It, along with OSATE, is widely used to specify and analyze critical systems such as avionics, automotive, and medical. They support such architecture analysis as performance, latency, and security. Additional analytical capability can be incorporated into them because of the extensible capability of the language. However, currently, there is no plugin for a testability measure. A testability measure based on controllability and observability has been successfully used previously in data flow software, hardware and co-designed systems. This thesis takes advantage of this method and adapts it to AADL.

A testability measure plugin was successfully built and experimental results demonstrating how this approach works are included in this thesis. This approach promises to help architects quickly gain insight about system components testability, about where faults can be hidden, and how to improve the system testability. Its results can lead to early modifications to system specifications, helping to help reduce cost in later development phases after the system has been implemented and integrated.

However, further research is needed. This testability approach is based primarily on data flows; further work on control flows is need for a more complete overview of system testability. The plugin developed here is used for the software components of the AADL specification only. Further work could extend the plugin to measure execution platform testability. In addition, changes can be made to the plugin to enhance it, for example, having line capacity determined automatically based on connection data types.

## REFERENCES

- (1) Bass L, Clements P, Kazman R. Software architecture in practice. : Addison-Wesley Longman Publishing Co., Inc.; 2003.
- (2) From hardware to software testability. Test Conference, 1995. Proceedings., International: IEEE; 1995.
- (3) Nikora AP, Some RR, Tamir Y. Increasing software testability with standard access and control interfaces. 2003.
- (4) Binder RV. Design for testability in object-oriented systems. Commun ACM 1994;37(9):87-101.
- (5) Analyzing testability on data flow designs. Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on: IEEE; 2000.
- (6) Baudry B, Traon YL. Measuring design testability of a UML class diagram. Information and software technology 2005;47(13):859-879.
- (7) Voas JM. Factors that affect software testability. 1991.
- (8) Freedman RS. Testability of software components. Software Engineering, IEEE Transactions on 1991;17(6):553-564.
- (9) Mulo E. Design for testability in software systems. 2007.
- (10) Bertolino A, Strigini L. On the use of testability measures for dependability assessment. Software Engineering, IEEE Transactions on 1996;22(2):97-108.
- (11) Im K. Debugging Techniques for Locating Defects in Software Architectures 2011.
- (12) Feiler PH, Gluch DP, Hudak JJ. The architecture analysis & design language (AADL): An introduction 2006.
- (13) Open Source AADL Tool Environment (OSATE). AADL Workshop, Paris; 2004.
- (14) Voas J, Morell L, Miller K. Using dynamic sensitivity analysis to assess testability. Using dynamic sensitivity analysis to assess testability 1991.
- (15) Voas JM, Miller KW. Semantic metrics for software testability. J Syst Software 1993;20(3):207-216.

- (16) Testability measurement and software dependencies. Proceedings of the 12th International Workshop on Software Measurement; 2002.
- (17) Testability measurements for data flow designs. Software Metrics Symposium, 1997. Proceedings., Fourth International: IEEE; 1997.
- (18) The SAE Architecture Analysis & Design Language (AADL) a standard for engineering performance critical systems. Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE: IEEE; 2006.
- (19) System Engineering Approaches for Performance Critical Avionics Embedded Computer Systems Using the Architecture Analysis and Design Language (AADL). ANNUAL FORUM PROCEEDINGS-AMERICAN HELICOPTER SOCIETY: AMERICAN HELICOPTER SOCIETY, INC; 2007.
- (20) Rolland JF, Bodeveix JP, Filali M, Chemouil D, Dave T. AADL modes for space software. Data Systems In Aerospace (DASIA), ESA Publications 2008.
- (21) AADL-Modelling of Plug&Play Weapon System Architecture. AADL Workshop, Paris; 2004.
- (22) Shiraishi S. An AADL-based approach to variability modeling of automotive control systems. Model Driven Engineering Languages and Systems 2010:346-360.
- (23) Hugues J. AADL for Cyber-Physical Systems: Semantics and beyond, validate what's next. 2011.
- (24) Gupta SKS. A tool for designing high-confidence implantable biosensor networks for medical monitoring. ACM SIGBED Review 2009;6(2):2.
- (25) Model-based architecture analysis for wireless healthcare. Proc. ACM MobiHoc Workshop on Pervasive Wireless Healthcare (MobileHealth'11); 2011.
- (26) Methods and Tools for Embedded Distributed System Scheduling and Schedulability Analysis. AADL Workshop, Paris; 2005.
- (27) Farail P, Gauffillet P. COTRE as an AADL profile. Architecture Description Languages 2005:167-179.
- (28) Aircraft integration real-time simulator modeling with AADL for architecture tradeoffs. Proceedings of the Conference on Design, Automation and Test in Europe: European Design and Automation Association; 2009.

- (29) Exploring the design space of IMA system architectures. Digital Avionics Systems Conference (DASC), 2010 IEEE/AIAA 29th: IEEE; 2010.
- (30) Statezni D. Analyzable and Reconfigurable AADL Specifications for IMA System Integration. Rapport technique, Rockwell-Collins 2004.
- (31) Feiler P, Lewis B, Vestal S, Colbert E. An overview of the SAE architecture analysis & design language (AADL) standard: a basis for model-based architecture-driven embedded systems engineering. Architecture Description Languages 2005:3-15.
- (32) An Overview of the Architecture Analysis & Design Language (AADL) Error Model Annex. AADL Workshop; 2005.
- (33) The AADL behaviour annex--experiments and roadmap. Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on: IEEE; 2007.
- (34) Niz Dd. Coupler Contracts Annex. January 2007.
- (35) SAE, inventor. AnonymousSAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex. . 2006-06-19 .
- (36) Research of embedded software testing method based on AADL modes. Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on: IEEE; 2011.
- (37) A Model-Based Testing for AADL Model of Embedded Software. Quality Software, 2009. QSIC'09. 9th International Conference on: IEEE; 2009.
- (38) Binh NT, Delaunay M, Robach C. Testability Analysis of Data-Flow Software. 2005.
- (39) Robach C, Malecha P, Michel G. CATA: A Computer-Aided Test Analysis System. Design & Test of Computers, IEEE 1984;1(2):68-79.
- (40) Robach C, Wodey P. Linking design and test tools: an implementation. Industrial Electronics, IEEE Transactions on 1989;36(2):286-295.
- (41) Towards a unified approach to the testability of co-designed systems. Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on: IEEE; 1995.

- (42) Testability analysis applied to embedded data-flow software. Quality Software, 2003. Proceedings. Third International Conference on: IEEE; 2003.
- (43) Dammak A. Etude de mesures de testabilité de systèmes logiques 1985.
- (44) Information based testability measures. Proceedings of Silicon Design Conference; 1986.
- (45) Feiler P. AADL Meta Model & XML/XMI. 2005.
- (46) Ford D, Fulkerson DR. Flows in networks. : Princeton university press; 2010.
- (47) CMU S. SAE AADL Example models.
- (48) Automatic testability analysis for data-flow designs of reactive systems. Testability Assessment, 2004. IWoTA 2004. Proceedings. First International Workshop on: IEEE; 2004.
- (49) Do H, Delaunay M, Robach C. MaC: A Testability Analysis Tool for Reactive Real-Time Systems. ERCIM news 2004;58:26.