

**VALIDATION OF GUIDANCE CONTROL SOFTWARE REQUIREMENTS
SPECIFICATION FOR RELIABILITY AND FAULT-TOLERANCE**

By

Hye Yeon Kim

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

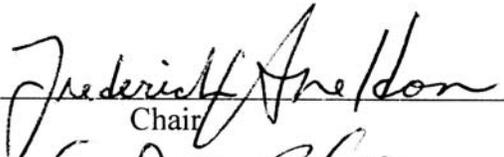
WASHINGTON STATE UNIVERSITY

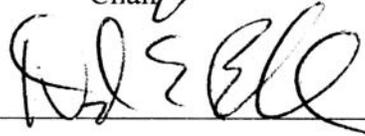
School of Electrical Engineering and Computer Science

May 2002

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of
Hye Yeon Kim find it satisfactory and recommend that it be accepted.


Chair





© Copyright 2002

Hye Yeon Kim*

All Rights Reserved.

* Contact information : hyekim@ieee.org

ACKNOWLEDGMENT

Most of all, I would like to thank my Lord Jesus for all of his love and support throughout my graduate study. Without his strengthening and encouragement, this thesis would not exist.

I would like to express sincere appreciation to my chair Dr. Sheldon's dedication for this research and its publications. His academic guidance and encouragement enabled this research to be successful. I would like to thank my committee member Dr. Bakken and Dr. Dang. Their comments and supports to this research were invaluable.

I thank my parents, Dal Hwan Kim and In Wook Lee, for their lifetime dedication for my education and my sister, Diane Hye Kyoung Kim, for her support. I am grateful to my dearest friends, Dean and Mary Guenther, for their love and care throughout my study at Washington State University. I would also like to thank to my friends Dr. and Mrs. Johnson, and Elly Soeryapranata for their advice and support. I am thankful for the time and effort my friend, Mary Baker, spent for editing this thesis.

I would like to express my sincere appreciation to i-logix for extending the Statemate licenses until the completion of this thesis. I am very grateful for their gift.

**VALIDATION OF GUIDANCE CONTROL SOFTWARE REQUIREMENTS
SPECIFICATION FOR RELIABILITY AND FAULT-TOLERANCE**

ABSTRACT

by Hye Yeon Kim, M.S.
Washington State University
May 2002

Chair: Frederick T. Sheldon

Many critical control systems are developed using CASE tools. Validation for such systems is largely based on simulation and testing. Current software engineering research has sought to develop theory, methods, and tools based on mechanized formal methods that will provide increased assurance for such applications. In addition to the previous fact, the present software engineering research focuses on allowing earlier error detection of overlooked cases, more complete testing using model checking to examine all reachable states, and full verification of critical properties using an automated theorem prover to undertake formal verification. This case study was performed for validating the integrity of a software requirements specification (SRS) for Guidance Control Software (GCS) in terms of reliability and fault-tolerance. A verification of the extracted parts of the GCS Specification is provided as a result. Two modeling formalisms were used to evaluate the SRS and to determine strategies for avoiding design defects and system failures. Z was used first to detect and remove ambiguity from a portion of the Natural Language based (NL-based) GCS SRS. Next, Statecharts, Activity-charts, and Module charts were constructed to visualize the Z description and make it executable. Using executable models, the system behavior was assessed under normal and abnormal conditions. Faults were seeded into

the model, an executable specification, to probe how the system would perform. Missing or incorrectly specified requirements were found during the process. In this way, the integrity of the SRS was assessed. The significance of this approach is discussed by comparing this approach with similar studies and possible approaches for achieving fault tolerance. This approach is envisioned to be useful in a more general sense as a means to avoid incompleteness and inconsistencies along with dynamic behavioral analysis useful in avoiding major design flaws. The iteration between these two formalisms gives pertinent analysis of a problem – i.e., operational errors between states, functional defects, etc.

LIST OF PUBLICATIONS

- Refereed international conferences:
 1. Frederick T. Sheldon, Hye Yeon Kim, and Zhihe Zhou. " A Case Study: Validation of the Guidance Control Software Requirements for Completeness, Consistency, and Fault Tolerance," in Proceedings of IEEE 2001 Pacific Rim International Symposium on Dependable Computing.
 2. Frederick T. Sheldon and Hye Yeon Kim. "Validation of Guidance Control Software Requirements Specification for Reliability and Fault-Tolerance," in 2002 Proceedings of Annual Reliability and Maintainability Symposium. IEEE.
- Refereed workshop:
 3. Frederick T. Sheldon and Hye Yeon Kim. "Software Requirements Specification and Analysis Using Zed and Statecharts," IEEE 3rd Workshop on Formal Descriptions and Software Reliability, 2000.
- Journal papers in preparation for submission.
 4. Hye Yeon Kim and Frederick T. Sheldon. "Evaluating Software Requirements for Completeness, Consistency and Fault-Tolerance," Submission in May 2002, IEEE Transactions on Reliability.
 5. Hye Yeon Kim and Frederick T. Sheldon. "Testing Software Requirements with Z and Statecharts model," Submission in May 2002, Special issue of the Requirements Engineering Journal. UMIST.

TABLE OF CONTENTS

ACKNOWLEDGMENT.....	iv
ABSTRACT.....	v
LIST OF PUBLICATIONS.....	vii
LIST OF TABLES.....	xi
LIST OF FIGURES.....	xii
CHAPTER	
1. INTRODUCTION.....	1
1.1. Problem definition.....	2
1.2. Motivation.....	3
1.3. Definitions.....	5
1.4. Organization.....	6
2. RELATED RESEARCH.....	7
2.1. Formal Methods.....	7
2.2. SRS Analysis/Evaluation/Assessment Studies.....	8
2.3. Related Case Studies with Z.....	10
2.4. Contribution from this Case Study.....	12
3. GCS REQUIREMENTS SPECIFICATION.....	14
3.1. Specification Excerpt.....	16
4. METHODOLOGY.....	19
4.1. Application Phase.....	19
4.2. Applied Methods.....	19
4.2.1 Z (Zed).....	21

4.2.2	Statecharts	23
4.2.3	Specification Tests	24
4.2.4	Fault Injection	25
4.3.	Application Example	26
4.3.1	Z Specification	29
4.3.2	Statecharts	32
4.3.3	Tests	35
4.3.4	Fault Injection	38
4.3.5	Reformulated Requirements	40
5.	RESULTS	42
5.1.	Z (Zed)	42
5.1.1	Global Constants and Functions	43
5.1.2	ARSP Module	52
5.1.3	CP Module	55
5.1.4	GP Module	65
5.1.5	RECLP Module	68
5.1.6	GCS Schema	70
5.2.	Executable Models	72
5.2.1	Module Chart	73
5.2.2	Activity Charts	74
5.2.3	Statecharts	75
5.3.	Specification Test Results	78
5.3.1	Test Results	78

5.3.2	Fault Injection Results/Discussion.....	79
6.	CONCLUSIONS.....	81
	Bibliography	83
APPENDIX	A.....	86

LIST OF TABLES

Table 1.	Cost of software defects [5, 6]	4
Table 2	Functional unit schedule [25].....	17
Table 3.	ARSP specification simulation result	36
Table 4.	ARSP specification test input and output	38
Table 5.	Detailed testing results – Case 1 example.....	38
Table 6.	Detailed fault injection results – Case 1 example.....	40
Table 7.	Fault injection simulation result.....	40
Table 8.	GCS excerpt high-level activity/state charts simulation result	79

LIST OF FIGURES

Figure 1.	A typical terminal descent trajectory [25]	15
Figure 2.	Velocity-altitude contour [25].....	16
Figure 3.	GCS system architecture.....	18
Figure 4.	NL-based specification for AR_COUNTER [25].....	20
Figure 5.	Form of axiomatic definition	22
Figure 6.	Form of a schema.....	22
Figure 7.	Free type notation	22
Figure 8.	Translation example from NL-based to statecharts	27
Figure 9.	ARSP_RESOURCE schema.....	30
Figure 10.	ARSP schema.....	31
Figure 11.	ARSP activity-chart	33
Figure 12.	INIT statechart	33
Figure 13.	ALTIMETER statechart.....	35
Figure 14.	Array definition with sequences	43
Figure 15.	Proof formula of the nmatrix function	44
Figure 16.	Z/EVES proof window	44
Figure 17.	Z/EVES specification window.....	45
Figure 18.	Proof formula of the rmatrix function.....	46
Figure 19.	T_any free type definition.....	46
Figure 20.	Global variable definition	46
Figure 21.	GUIDANCE_STATE_1 schema	47

Figure 22.	GUIDANCE_STATE_2 schema	49
Figure 23.	GUIDANCE_STATE schema	49
Figure 24.	EXTERNAL schema	50
Figure 25.	RUN_PARAMETERS schema.....	51
Figure 26.	SENSOR_OUTPUT schema	52
Figure 27.	ARSP_RESOURCE schema	53
Figure 28.	ARSP_FUNCTION schema	53
Figure 29.	ARSP schema.....	54
Figure 30.	CP_RESOURCE_1 schema.....	56
Figure 31.	CP_RESOURCE_2 schema.....	57
Figure 32.	CP_RESOURCE schema.....	58
Figure 33.	CP_PREP_MASK1 schema.....	58
Figure 34.	CP_MASK schema	62
Figure 35.	CP_FUCNTION schema	63
Figure 36.	CP schema.....	64
Figure 37.	GP_FUCNTION schema	65
Figure 38.	GP_1 schema	66
Figure 39.	GP schema	67
Figure 40.	A module chart of the GCS excerpt.....	73
Figure 41.	Actual module chart of the GCS excerpt	74
Figure 42.	GCS activity chart.....	75
Figure 43.	GCS_CONTROL statechart.....	76
Figure 44.	SUBFRAME1 statechart.....	76

Figure 45.	SUBFRAME2 statechart.....	77
Figure 46.	SUBFRAME3 statechart.....	77
Figure 47.	ARSP activity chart.....	87
Figure 48.	ARSP_CONTROL state chart	88

DEDICATION

This thesis is dedicated to my Lord Jesus Christ who
provided me all of his best.

CHAPTER ONE

INTRODUCTION

The trend of using software in embedded real-time systems and the fact that requirements for such software are often complex and therefore difficult to understand necessitate increasing employment of formal methods (FMs) in the software requirements specification and verification. Requirements validation is concerned with checking the requirements document for consistency, completeness and accuracy [1]. The main problem of requirements validation is that there is no existing document which can be a basis for the validation. A design or a program can be validated against the specification. However, there is no way to demonstrate that a requirements specification is correct with respect to some other system representation [1]. Therefore, requirements validation really means ensuring that the requirements specification represents a clear description of the system for design and implementation and is a final check that the requirements meet stakeholder needs. Validating such a document is the final stage of the requirements engineering. In this thesis, a requirements specification of a typical example for embedded real-time software - the Viking Mars Lander Guidance Control Software - is validated.

The particular notations that are selected to express requirements or designs can have a very important impact on the construction time, correctness, efficiency, and maintainability of the target system. One desirable property for these notations is that they be precise and unambiguous, so that clients and implementers can agree on the required behaviors and observe them in operation. The notation should be possible to state and prove properties of a system

before it is built; then, if the system is constructed according to the specifications, it may be guaranteed to exhibit certain properties and behaviors. This implies that the selected notation is not only formally defined but is also amenable to mathematical/logical manipulation. Observation of behaviors is particularly convenient if the specification language is executable. Executable specifications are also useful for clarifying and refining requirements and designs [2].

The term ‘formal methods’ applies to a variety of methods that are used to ensure correctness of the software, and their common characteristic being a mathematical foundation that make it possible to prove correctness of software in the mathematical sense. The approach chosen combines a model-based formal method (FM) which uses the mathematical theory of sets, propositional and predicate logic with a state-based diagrammatic formalism to visualize and simulate the specification (including fault injection). The first case correctness proof was employed while the behavior of executable specifications was gauged through visualization and simulation in the second case. This thesis covers a case study that is conducted to validate a software requirements specification in terms of reliability and fault-tolerance using Z and Statecharts.

1.1. Problem definition

Critical systems, such as safety-critical systems, mission-critical systems, and business-critical systems [3], demand rigorously engineered software. A failure in the control software of such systems can be disastrous. However, it is difficult to create a reliable software specification because such control software tends to be highly complex. To avoid problems in the latter development phases and reduce life-cycle costs, it is crucial to ensure that the specification be reliable. Moreover, such control software is required to tolerate failure because the system is typically cost/safety critical and operate in very harsh environments.

Practically, no system is absolutely fault free. There are plenty of catastrophic failures to substantiate this [4]. The probability of system failure decreases in accordance with a cautious specification and design process. However, the more complex the system, the more difficult it is to achieve high integrity and fault tolerance.

The typical SRS is highly dependent on natural language. Natural language (NL)-based specifications are often subject to multiple interpretations. Even when such specifications are developed systematically, it is difficult to ensure their integrity without some form of correctness checking. Generally, correctness checking obligates the use of a mathematically based requirements specification language (RSL). Such languages are notoriously difficult to understand, and minimally require a proficient level of knowledge in discrete mathematics and/or some formal logic system. This poses a serious concern to industry because many different classes of requirements exist, and different stakeholders typically signify various ways of looking at the problem. Thus, in regards to the requirements specification, a multi-perspective analysis is important, as there is no single correct way to analyze system requirements [3]. The usefulness of the requirements specification diminishes by not being understandable to the diverse set of stakeholders.

1.2. Motivation

The Software development starts from specifying the requirements of the software. A Software Requirements Specification (SRS) describes what the software must do. Naturally, the SRS takes the core role as the descriptive documentation at every phase of the software development cycle. Therefore, it is required to make sure the SRS contains correct and complete information for the system. For that reason, employing a verification technique is necessary for the specification to provide some support of prototyping, correctness proofs, elaboration of test data, and failure

detection. Moreover, early detection of failures and incorrectly specified requirements can reduce the amount of money and effort for the corrective work. Table 1 shows roughly the relationship between correction costs and the development phase when the defect was discovered, and detecting problems earlier, rather than later, provides significant rewards. To avoid problems in the latter development phases and reduce the software life-cycle costs, it is crucial to ensure that the specification be reliable.

Table 1. Cost of software defects [5, 6]

When defect is detected	Typical Costs of Correction
Requirements Specification	\$100-\$1,000
Coding/Unit Testing	\$1,000 or more
System Testing	\$7,000-\$8,000
Acceptance Testing	\$1,000-\$100,000
After Implementation	Up to millions of dollars

Most problems can be traced to the requirements specification typically due to the ambiguity [7]. Formal methods unambiguously define the requirements of software with respect to its specification. They are the primary way to have a rigorous definition of correctness of the system requirements. The decision to use formal specifications mainly depends on the criticality of the component, in term of severity of fault consequences and of the complexity of its requirements or of its development [8].

Z is a formal requirements specification language that uses set theory as the basic building blocks of complex data structures combined with first-order predicate logic. A specification written in Z is a mixture of formal, mathematical statements and informal explanatory text [9]. Z was used in this case study to clarify intentions, identify assumptions, introduce precision and explain correctness in light of ambiguous statements found in the NL-based SRS.

Considering the complexity of the control system, it is hard to prove the specification is complete and consistent without any automated tool support. There are several tools supporting the Z specification proof/refinement. However, they do not provide any improvement on the understandability of the requirements for a set of diverse stakeholders. As mentioned before, mathematically specified requirements are not easy to comprehend. Even though a Z specification has informal explanatory descriptions, it is mathematically represented. By describing the requirements as a set of graphical models, one can improve understandability of the requirements better than either the detailed NL descriptions or the mathematical representations. Statecharts were chosen to model the Z specifications because they provide a means of visualization and a means to test the specification.

1.3. Definitions

Reliability, as applied to the software requirements specifications, means: (1) is the specification correct, unambiguous, complete, and consistent; (2) can the specification be trusted to the extent that design and implementation can commence while minimizing the risk of costly errors; and (3) how can the specification be defined to prevent the propagation of errors into the downstream activities?

The completeness of a specification is defined as a *lack of ambiguity* in the implementation. The specification is incomplete if the system behavior is not specified precisely because the required behavior for some events or conditions is omitted or is subject to more than one interpretation [10]. Consistency, the presence of a lack of ambiguity in requirements, means the specification is free from conflicting requirements and undesired nondeterminism [11].

Typically, fault-tolerance is considered as a implementation methodology that provides for (1) explicit or implicit error detection for all fault conditions, and (2) backup routines for

continued service to critical functions in case errors arise during operation of the primary software [8]. For the software requirements specification, it can be defined as (1) subsistence of specified requirements to detect explicit or implicit errors for all fault conditions, and (2) presence of specified requirements that supports the system robustness, software diversity, and temporal redundancy for continuing service of critical system functions in case of failure.

1.4. Organization

The next chapter of this thesis provides an introduction of formal methods and case studies that were conducted to evaluate software requirements specifications. Chapter 3 presents a brief description of the GCS requirements including a specification excerpt. Chapter 4 describes the methodology used in this thesis and a simple application example to show how the method was implied for evaluation of GCS SRS excerpt introduced in chapter 3. Chapter 5 presents the Z and Statecharts models were built based on the abstracted information from the GCS SRS and symbolic simulation results (i.e., testing results, fault injection results). Chapter 6 concludes analysis results of this case study and chapter 7 finishes this thesis with a plan for the future studies.

CHAPTER TWO

RELATED RESEARCH

In this section, several categories of analysis methods are introduced for the safety/mission critical system software requirements. In addition, a number of researches are presented which are sought to find a way to verify software requirements specifications for critical systems for consistency and completeness. Last, numerous related case studies are introduced that are conducted using Z and other formal methods for the benefit of visualization and/or dynamical assessment.

2.1. Formal Methods

Formal methods are a collection of techniques rather than a single technology to apply on specifying a software system. The sole objective for using formal methods is to provide a way to eliminate inconsistency, incompleteness, and ambiguities. Because the formal methods have an underlying mathematical basis, it provides valid analysis of a system better than ad hoc reviews. There are several classes of distinguishable formal specification techniques (also called as formal methods). They are model-oriented specifications, property-oriented specifications, and operational specifications [12].

In the property-oriented approaches, known as constructive techniques, one declares a name list of functions and properties. Infinite numbers of models are represented by this method because infinite numbers of functions can be provided for each of the previously declared names. Among the models, only a few of them satisfy the required properties. The software is correct with respect to a specification if it provides all the declared function names and defines a model

that satisfies the specification [12]. These approaches provide notations that can depict a series of data, and use equations to describe the system behaviors rather than building a model. These property-oriented approaches can be broken into algebraic and axiomatic specifications [13]. The algebraic specification describes a system as an algebra that is consisted with a set of data and a number of functions over this set [14]. The axiomatic specification has its origin in the early work on program verification. It uses first-order predicate logic in pre- and post-conditions to specify operations [13].

In the model-oriented approach, known as declarative techniques, one builds a unique model from a choice of built-in data structures and construction primitives that the specification language offers [12]. This approach provides a direct way of describing system behaviors. The system is specified in terms of mathematical structures such as sets, sequences, tuples, and maps [13]. It defines the correctness based on the model behaviors whether it meets the specified functionality [12]. Vienna Development Method (VDM), B, and Z are fit into this category.

The Operational/Executable specification is another category of formal specification techniques. It provides sets of actions that describe the sequence of the system behavior and computational formulas that describe the performance calculation. Petri nets, process algebra, and state/activity charts in the STATEMATE² environment [2] are considered to be in this category [12].

2.2. SRS Analysis/Evaluation/Assessment Studies

There have been numerous studies with the goal of improving the integrity, identifying defects, and removing ambiguities (completeness and consistency). Fabbrini et al came up with an automatic evaluation method called “Quality Analyzer of Requirements Specification (QuARS)”

to evaluate the quality of SRS. They defined testability, completeness, understandability, and consistency as properties of a high quality SRS [15]. The QuARS tool was employed to parse requirement sentences written in natural language (NL) and point out potential source of errors in the SRS. This is a linguistic, informal evaluation approach rather than a formal method. This approach shows informal methods can expose some of errors in SRS. Authors claim this approach is applicable to any domain of software based on the tool's ability of the customization of dictionaries. However, this approach did not provide any clear quantitative or qualitative measure of quality for an SRS.

Heitmeyer et al., used Software Cost Reduction (SCR) tabular notation to expose inconsistencies in software requirements specifications. The SCR method is applied to expose a safety violation in a safety-critical software requirements specification. They used "Two Pushbutton" abstraction method to reduce infinite system state space into a sizable one because the enormous state space of specifications of practical software usually renders direct analysis impractical [16]. Two redundant specifications were used to represent the required behavior of a system. In this step, Petri-net and TRIO specification logic were employed. They analyzed reduced size of SRS with Spin and a simulator which they developed to support the SCR method. This approach is quite complicated to understand and apply for highly complex systems due to the involvement of too many formalism and tools.

Heimdahl and Leveson used Requirements State Machine Language (RSML) to verify requirements specifications for completeness and consistency [17]. RSML is a state-based requirements specification language suitable for the specification of reactive systems. It includes several features developed by Harel for Statecharts: superstates, AND decomposition, broadcast communication, and conditional connectives. Superstates mean a state that represents groups of

² STATEMATE Magnum – product of i-Logix, was used to conduct the research for this thesis.

states. A superstate may have a number of substates. Such groupings reduce the number of transitions by allowing transitions between superstates rather than explicit transitions between all substates. AND decomposition (also known as orthogonal product) means that several state machines are allowed to be combined into one state. Those state machines are combined as parallel state machines. When the AND decomposed state is entered, each of the parallel state machines are entered. Exiting from any of those parallel state machines makes all state machines be exited. This use of parallel state machines reduces the size of the specification. In RSML, the transitions are represented as relationships between states (i.e., hierarchical, next-state mappings). The functional framework defined in [17] to check the model against every possible input, to find conflicting requirements, to verify whether the model is deterministic. They used textual representation based simulator developed for RSML to execute the requirements specification. This approach shows several advantages such as the global reachability graph is not required to perform analysis just for a part of a system.

2.3. Related Case Studies with Z

There have been numerous studies conducted that combine a Z specification with some formal method or design notation. A hybrid formal method called PZ-nets is suggested by Xudong He. PZ-nets combine Petri nets and Z notations [18]. PZ-nets provide a unified formal model for specifying the overall system structure, control flow, data types and functionality. Sequential, concurrent and distributed systems are modeled using a valuable set of complementary compositional analysis techniques. However, modular and hierarchical facilities are needed to effectively apply this approach to large systems.

Hierons, Sadeghipour, and Singh present a hybrid specification language μ SZ [19]. The language uses Statecharts to describe the dynamical system behavior and Z to describe the data

and data transformations. In μ SZ, Statecharts define sequencing while Z is used to define the data and operations. Their data abstraction technique uses information derived from the Z specifications to produce an Extended Finite State Machine (EFSM) defined by the Statecharts. The EFSM features that can be utilized during test generation. These features help solve the test automation problem of setting up the initial state and checking the final state of each test. Both the dynamic behavior specified in Statecharts and the individual operations are checked using these features.

Bussow and Weber present a mixed method consisting of Z notations and Statecharts [20]. Each method was applied to a separate part of the system. Z was used in defining the data structures and transformations. Statecharts were used in representing the overall system and the reactive behavior. The Z notations were type checked with the ESZ type-checker but the statecharts semantics were not fully formalized. In addition, several other case studies utilized Z for defining data while Statecharts were used as a behavioral description method [21-23].

Castello developed a framework for the automatic generation of the layout of statecharts from a database that contains information abstracted from a SRS [24]. He developed a tool to generate statecharts layout and generated Z schemas from the statecharts layout. He abstracted data from the SRS to generate a database, and then he used the tool to create statecharts from the info which derived from the database entry. The statecharts were translated by one to one match into Z schemas. His Z schemas were exact replica of the statecharts. In the other words, the Z schema was the text version of the statecharts. Both, the method and the criteria of SRS abstraction for the database entry, were not explained in [24].

He claims that it is necessary to translate the statecharts into Z to validate the correctness of the statecharts layout. To evaluate the correctness of the statecharts layout, he generated an exact

replica of the statecharts layout in Z schema using the tool. However, the statecharts are considered as a formal method [2], which is verifiable by using several off-the-shelf products that support statecharts simulations and model checking without translating into Z.

How this approach can validate SRS which is usually involved with infinite state systems when it is formed into statecharts? Statecharts do not allow representing infinite state systems. To develop statecharts, one should find a way to reduce the system states into a discrete number. This issue was not addressed in the paper.

2.4. Contribution from this Case Study

A clear distinction of this approach as compared to others is that Z and Statecharts are not concurrently used to evaluate separate parts of the SRS. The part of the NL-based SRS is translated completely into Z and then the Z specification was translated into Statecharts. The Z specification is type checked and proved for correctness using Z/EVES with reduction/refinement before translating it into statecharts. State/Activity charts in Statemate environment are tested for consistency and completeness using simulations and the model checking. By injecting faults into the statecharts model, the transformed SRS is evaluated for fault-tolerance. Details of the tests and faults injections are described in chapter 4.

Z and statecharts do have different precision for revealing flaws inheriting in a SRS. In general, it is believed that Z has more ability to define data types and Statecharts are better to support to check the state transition over Z [21-23]. When one uses conjoined methods like other case studies, the consistency between the joined methods cannot be tested or verified with any technique. By transforming the specifications with one method at a time, the consistency can be assured for the transformed specification with the method. For example, the consistency of Z specification is verifiable using the type-checking and proofs features in the Z/EVES and the

consistency and completeness of the Statecharts model are verifiable using the model checker and simulations in the Statemate environment. Refinement between these two different formalisms gives a more in-depth understanding of requirements, and reveals different flaws in SRS.

Next, the usefulness of this approach is evaluated by applying it to some critical parts of the SRS.

CHAPTER THREE

GCS REQUIREMENTS SPECIFICATION

The Guidance and Control Software (GCS) principally provides control during the terminal phase of descent for the Viking Mars Lander. The lander has three accelerometers, one Doppler radar with four beams, one altimeter radar, two temperature sensors, three gyroscopes, three pairs of roll engines, three axial thrust engines, one parachute release actuator, and a touch down sensor. After initialization, the GCS starts sensing the vehicle altitude. When a predefined engine ignition altitude is sensed, the GCS begins guidance and control of the vehicle. The purpose of this software is to maintain the vehicle along a predetermined velocity-altitude contour. Descent continues along this contour until a predefined engine shut off altitude is reached or touchdown is sensed.

The initialization of the GCS starts the sensing of vehicle altitude. When the altimeter radar senses a predefined engine ignition altitude, the GCS starts controlling descent of the vehicle. The axial and roll engines are ignited; while the axial engines are warming up, the parachute remains connected to the vehicle. During this engine warm-up phase, the aerodynamics of the parachute dictate the trajectory (see Figure 1) followed by the vehicle. Vehicle altitude is maintained by firing the engines in a throttled-down condition. Once the main engines become hot, the parachute is released and the GCS performs an altitude correction maneuver and then follows a controlled acceleration descent until a predetermined velocity-altitude contour is crossed (see Figure 2). The GCS then attempts to maintain the descent of the vehicle along this predetermined velocity-altitude contour. The vehicle descends along this contour until a

predefined engine shut off altitude is reached or touchdown is sensed. After all engines are shut off, the vehicle free-falls to the surface.

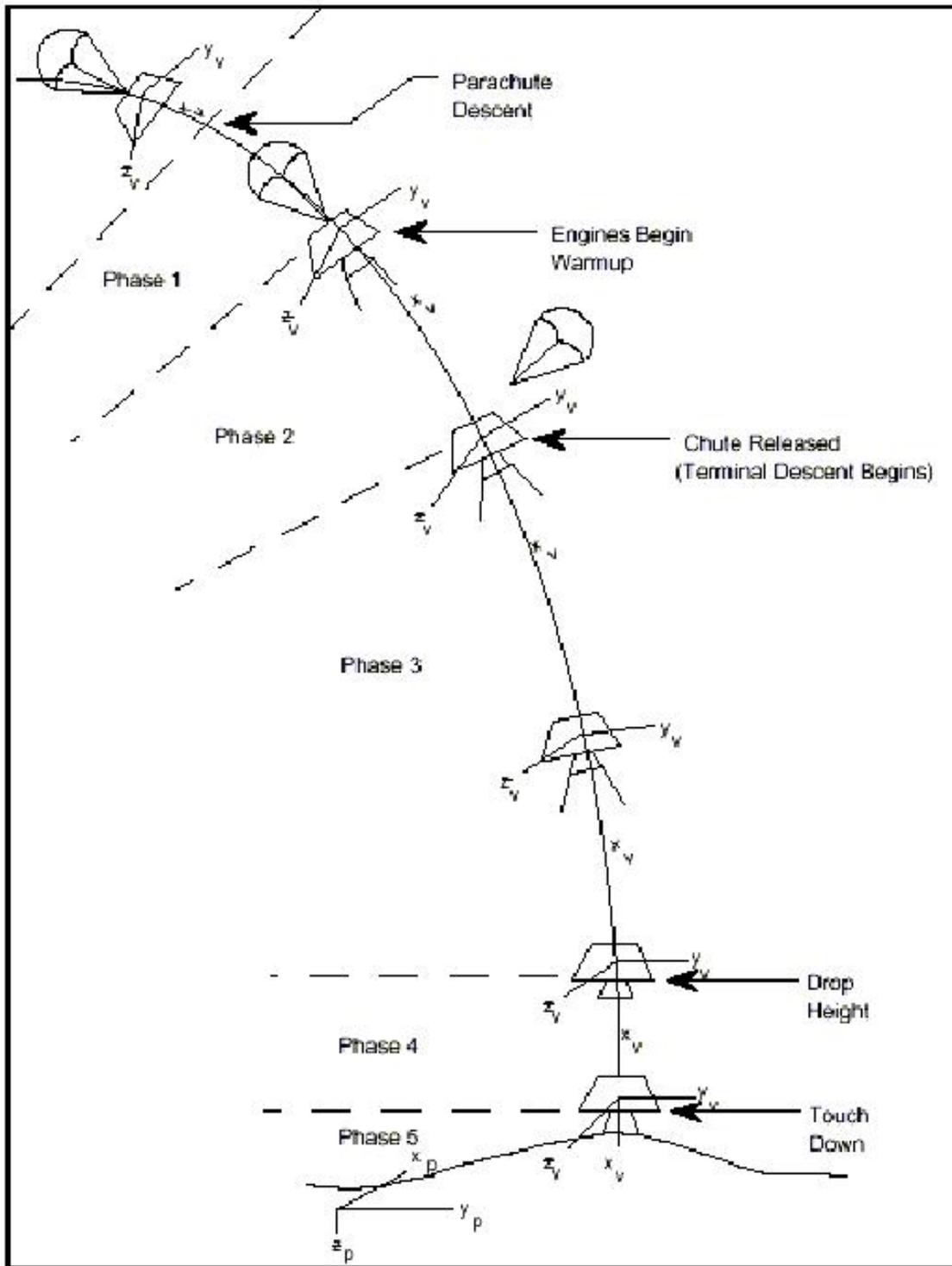


Figure 1. A typical terminal descent trajectory [25]

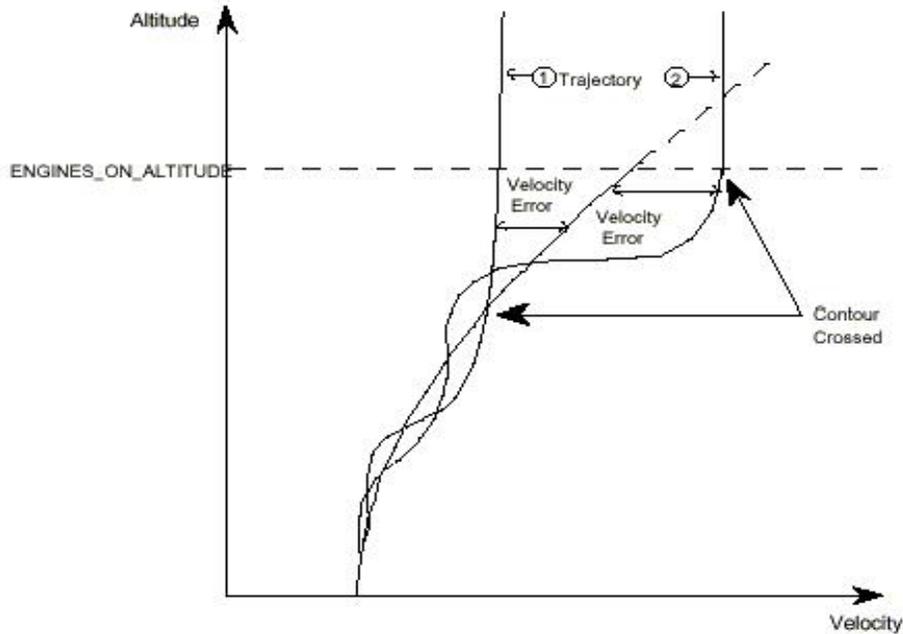


Figure 2. Velocity-altitude contour [25]

3.1. Specification Excerpt

Figure 3 shows the overall system architecture of the GCS software. The circled parts are the subunits consisting of the partial specification for this case study. The partial specification that was examined includes one sensor processing unit, one actuator unit, and the two core subunits of the GCS system (circled units in Figure 3). All other subunits are ignored in this case study except the data stores. Control and data flows between the excerpted modules are the same as they are represented in the Module chart (Figure 40).

The choice of parts for the specification excerpt is made based on its run-time schedule (Table 2). The GCS has predetermined running time frame that is consists of three subframes. Each subframe has specific submodules to run. The partial specification under this study is consists of one submodule from each subframe and a submodule that runs every subframe. ARSP (Altimeter Radar Sensor Processing) is running in the first subframe, GP (Guidance Processing) is running in the second subframe, and RECLP (Roll Engine Control Law

Processing) is running in the third subframe. CP (Communication Processing) is running in every subframe. In SRS, CP is specified as the last submodules to run for every subframe. The order of the submodules in the same subframe is not declared except CP. All of these submodules share the control of some common variables. The modification of those variables is verified by the testing. The detailed testing method is described in chapter 4.2.

Table 2 Functional unit schedule [25]

SCHEDULING	
Sensor Processing Subframe (Subframe 1)	
ARSP	1
ASP	1
GSP	1
TDLRSP	1
TDSP	5
TSP	2
CP	1
Guidance Processing Subframe (Subframe 2)	
GP	1
CP	1
Control Law Processing Subframe (Subframe 3)	
AECLP	1
CRCP	5
RECLP	1
CP	1

The ARSP (Altimeter Radar Sensor Processing) is a sensor processing submodule of the GCS. This functional unit reads the altimeter counter provided by the altimeter radar sensor and converts the data into a measure of distance to the surface of Mars. The CP (Communication Processing) is a submodule that converts the sensed data into a data packet appropriate for radio transformation. The data packets are relayed back to the orbiting platform for later analysis. The GP (Guidance Processing) is the core-processing submodule of the GCS. This module gathers the information from the entire sensor processing subunits and the previous computational results. Then, it manages the vehicle's state during the descent by controlling the actuators. The

RECLP (Roll Engine Control Law Processing) is an actuator unit that computes the value settings for three roll engine. The roll engine value settings are calculated to fix the difference between the vehicle's measured values during operation and the designated trajectory values.

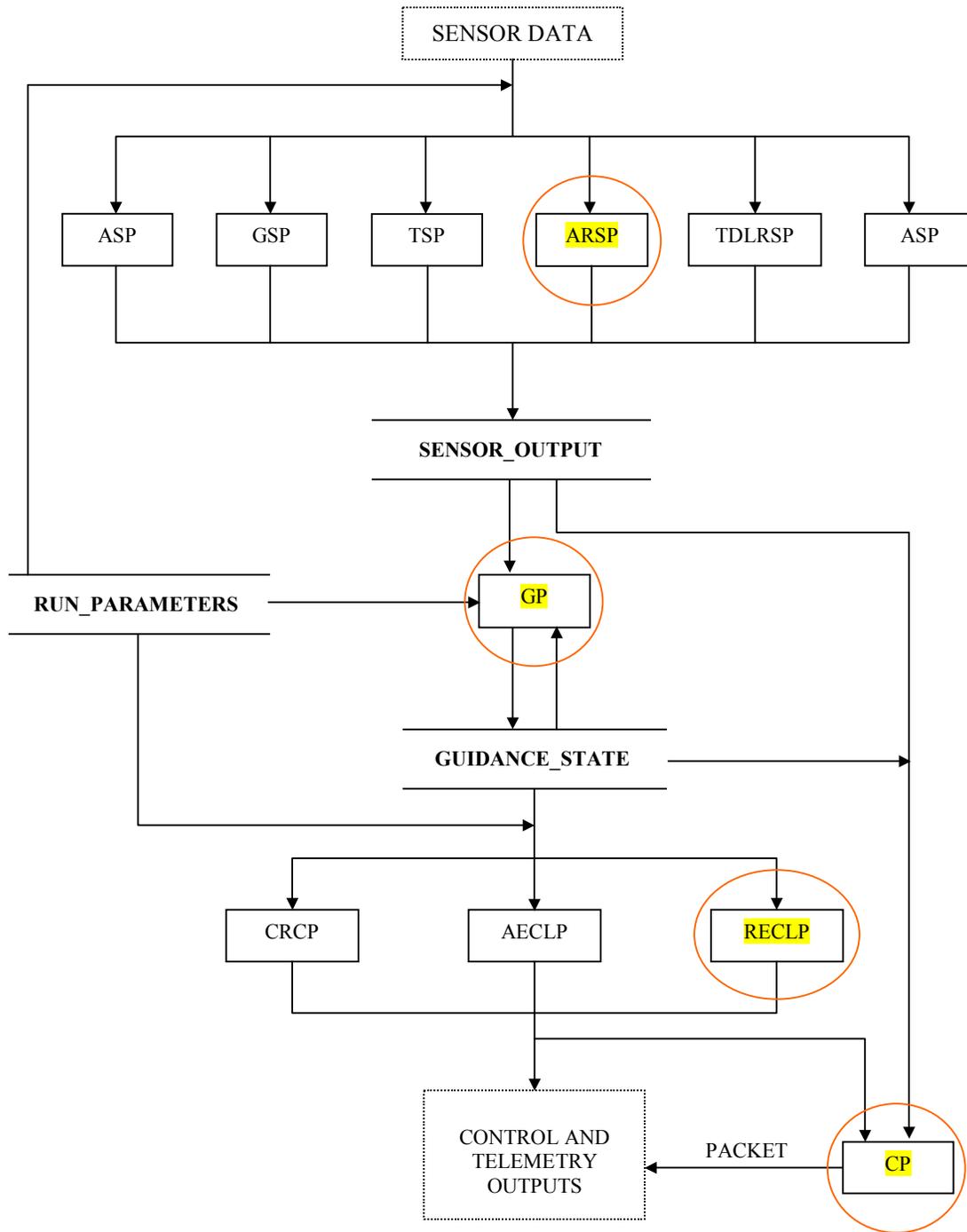


Figure 3. GCS system architecture

CHAPTER FOUR

METHODOLOGY

4.1. Application Phase

In requirement engineering, Formal Methods are used in the process of the requirements validation. It is the last step of the requirement specification phase of the typical software development cycle. At the requirement validation phase, one must evaluate the final version of the SRS credibility.

4.2. Applied Methods

The employed approach for this case study is a two-step process using Z/Statecharts. First, the NL-based GCS requirements specification is transformed using the Z notation. Z is used to clarify ambiguous statements found in the SRS. For example, AR_COUNTER is specified in two sections (Figure 4) in NL-based GCS.

The statements in Processing Unit (left column in Figure 4) describe AR_COUNTER modification and values. One can conclude that AR_COUNTER is increasing after the radar pulse is transmitted from the first two sentences. This means that the AR_COUNTER value is a positive number when the radar pulse is transmitted whether an echo is arrived or not. It conflicts with the last sentence that states the AR_COUNTER will contain sixteen one bits that representing a negative one according to the definition in data dictionary (right column in Figure 4).

Processing Unit	Data Dictionary
A digital counter (AR_COUNTER) is started as the radar pulse is transmitted. The counter increments AR_FREQUENCY times per second. If an echo is received, the lower order fifteen bits of AR_COUNTER contain the pulse count, and the sign bit will contain the value zero. If an echo is not received, AR_COUNTER will contain sixteen one bits.	NAME: AR_COUNTER DESCRIPTION: counter containing elapsed time since transmission of radar pulse USED IN: ARSP UNITS: Cycles RANGE: [-1, 2 ¹⁵ -1] DATA TYPE: Integer*2 ATTRIBUTE: data DATA STORE LOCATION: EXTERNAL ACCURACY: N/A

Figure 4. NL-based specification for AR_COUNTER [25]

Z is used because it provides a concrete way to transform the requirements into state-based models using the schematic structuring facilities. The transformation elucidates assumptions and provides mechanisms for refining abstract specifications into concrete ones for clarifying data and functional definitions. Z Schemas are abstracted from the GCS SRS. This compositional process helped to clarify ambiguities. Second, the Schemas are transformed into Statecharts/Activity-charts and symbolically executed to assess the model's behavior based on the GCS-specified mission profile. A clear distinction of this approach with other approaches is that Z is not used as a conjunct³ method of Statecharts. The SRS was translated into Z completely and then translated into Statecharts.

Developing Statecharts/Activity charts from the Z schema is not a direct transformation process. It requires in-depth knowledge of Z model because Z allows specifying infinite (countably infinite) numbers of the system state. To specify a model that involves with infinite system states into statecharts, one should refine the number of states into the discrete numbers.

³ This means that Z and Statecharts are used to specify different part of the same specification (i.e., Z for data specification and Statecharts for system behavior).

The development of statecharts and activity charts in Statemate environment from Z specification is an iterative process. The refinement process involves Activities and States specified in Statemate tool first. Simulations are performed to verify that the statecharts do not have any nondeterministic state/activity transitions. After checking inconsistency for data items are included in the Statemate model, all the data and transition conditions specified in the statecharts model. Simulations are performed again to verify the second transformation did not effect the overall model transitions. In this second simulation process, some improperly defined function/data items in Z were found. Some function/data items were correct in ranges and types in both Z and Statecharts; however, they were generating incorrect output during simulations. This error information is used for the refinement of the Z schemas into more accurate terms.

After the simulation, faults are injected into state/activity charts in STATEMATE environment. It is done by changing state variable values while running simulation. The output from simulation with injected faults is compared with expected output. The expected output values are obtained by calculation based on the formula given in SRS. This fault-injected simulation enables one to evaluate system ability for coping unexpected system failure.

4.2.1 Z (Zed)

Z is classified as a model-based specification language that is equipped with an underlying theory that enables nondeterminism to be removed mechanically from abstract formulations to result in specifications that are more concrete. In combination with natural language, it can be used to produce a formal specification [26].

Axiom is one of the ways of defining global object in Z. It consists of two parts: declaration and predicate in Figure 5. The predicate constrains upon objects introduced in the declaration.

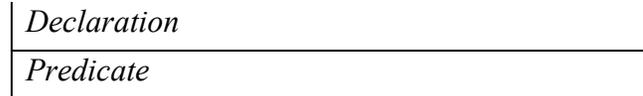


Figure 5. Form of axiomatic definition

Schema's are the main structuring mechanism used to create patterns and objects. The Schema notation is used to model system states and operations. A schema consists of two parts: a declaration of variables; and a predicate constraining their values. The name of a schema is optional, however, it is more convenient to give a name to be referred in other schemas.



Figure 6. Form of a schema

Free type can be used to define new types. Free types are similar to the enumerated types provided by many programming languages [27].

$$Free_type_name ::= constants \mid constructor \langle\langle source \rangle\rangle$$

Figure 7. Free type notation

Figure 7 introduces a collection of constants, one for each element of the set *source*. Constructor is an injective function whose target is the set *Free_type_name*. Consistency of free type can only be validated when each of the constructions (i.e., the set *source*) is involved with Cartesian products, finite power sets, finite functions, and finite sequences [26].

In this case study, the state of the system and the relationship between the ARSP and the state of various components are explained. The production of such a specification helps one to understand requirements, clarify intentions, and identify assumptions and explain correctness. These facilities are useful and essential in clarifying ambiguities and solidifying one's understanding of the requirements.

4.2.2 Statecharts

Statecharts, a state-based formal diagrammatic language, constitute a visual formalism for describing states and transitions in a modular fashion, enabling cluster orthogonality (i.e., concurrency) and refinement, and supporting the capability for moving between levels of abstraction. The kernel of the approach is the extension of conventional state diagrams by AND/OR decomposition of states together with inter-level transitions, and a broadcast mechanism for communication between concurrent components. The two essential ideas enabling this extension are the provision for depth (level) of abstraction and the notation of orthogonality. In other words, Statecharts = State-diagrams + depth + orthogonality + broadcast-communication [28].

Statecharts (using STATEMATE) provide a way to specify complex reactive systems both in terms of how objects communicate and collaborate and how they conduct their own internal behavior. Together, Activity-charts and Statecharts are used to describe the system functional building blocks, activities, and the data that flows between them. These languages are highly diagrammatic in nature, constituting full-fledged visual formalisms, complete with rigorous semantics providing an intuitive and concrete representation for inspecting and checking for conflicts [29]. The Activity-charts and Statecharts are used to specify conceptual system models for symbolic simulation. Using the simulation method, assumptions were verified, faults were injected, and hidden errors were identified that represent inconsistencies or incompleteness in the specification.

In this case study, a GCS project was created within the Statemate environments. Graphical editors were used to create Module chart, Statecharts and Activity-charts. Once the graphical forms were characterized, state transition conditions and data items were defined. These items and/or conditions trigger activities and state transitions that occur within the Statemate model

based on definitions within the “data dictionary” and/or the “data bank browser.” The Activity-chart and Statecharts reflect all variables/conditions defined in the Z formulation. During simulation, various color changes and simulation monitor help to show the sequence of state changes that occur to validate the system according to its specified structure (based on Schema signatures) and constraints (based on Schema predicates). Initial (and current) values and conditions were changed while at the same time rerunning and/or resuming the simulation in the process of verifying assumptions against the Statecharts specification. In this way, the Statecharts model was executed and tested.

4.2.3 Specification Tests

The statecharts model is tested in two different ways in the Statemate environments. First, the state/activity charts are tested as finite state machines. In this way, the consistency and the completeness of those charts are verified. This testing involves with state transition conditions and the triggers of activities. Next, the functionality of the statecharts model is tested using simulations. The actual outputs (values that are generated by the state/activity charts simulations) are compared with the expected output (values that are calculated based on the equations given in the SRS and defined in the Z schemas). Using graphical simulation, both of the tests are performed.

4.2.3.1 Finite State Machine Approach

Bogdanv and Holcombe discusses about how they tested a statecharts model of the aircraft control system in [30]. They extended a method to examine finite state machine for the statecharts model. It is reasonable to extend their method to this case study because the statecharts are a state-based specification language.

The test cases are generated to evaluate whether the statecharts model is behaviorally equivalent to the NL-based GCS specification. In other words, every activity and state transitions are performed as described in the SRS. This testing assures that there are no absorbing states/activities in the statecharts model.

4.2.3.2 Data Item approach

In the data item approach, the state/activity charts are treated like a software program. The test cases are generated to evaluate whether the statecharts model produces the correct outputs. The input and output values are determined based on the information from the data dictionary and the equations given by the SRS. The expected outputs are calculated from a pre-selected set of inputs. This test assures that there are no inconsistent or unspecified operations with in the SRS.

4.2.4 Fault Injection

Fault injection is a technique used to observe how a software system behaves under experimentally-controlled, anomalous circumstances. Voas et al., define that system anomalies are caused by either faulty codes or corrupted inputs otherwise combinations of both. They take the approach of injecting anomalies rather than injecting faults in the software program [31]. In this case study, the evaluation with abnormal inputs and outputs are covered by the test using the data item approach (Chapter 4.3.3.2).

In this process, the system state transitions and incorrect output due to the abnormal system failures are focused to analyze. While running the simulation, either the system variables were altered or state transitions are redirected. Test cases were generated based on the functionality and significance of the failure. The fault injection is not performed to the submodule that would not cause critical system failure. In this way, the SRS is evaluated for the fault-tolerance.

4.3. Application Example

The example shown in this section is the part of publications that describes a prototypical case study using same validation methods as used in this thesis [32, 33]. This section shows the only one submodule of the GCS SRS transformation via Z and Statecharts.

The Altitude Radar Sensor Processing (ARSP) module specification showing inputs, outputs, and subsystem processing descriptions was chosen for the purpose of this application example. The SRS provides a data dictionary with variable definitions, type, and units, and a brief description of variables and functions. The NL-based module specification was abstracted into Z, while variable names, operations (i.e., functionality), dependency and scope were preserved. Figure 8 provides an example using the FRAME_COUNTER input variable that illustrates the complete translation from Z to Statecharts. The top box in the Figure 8 represents the NL-based SRS. The box in the middle of the Figure 8 represents the Z Specification while the bottom box shows a part of the Statecharts model of ARSP submodule. In the NL-based SRS, the FRAME_COUNTER is defined as an integer with range $[1, 2^{31}-1]$. In Z, the FRAME_COUNTER is declared as a set of natural numbers in the signature part, and the range of the variable is defined in the predicate part (lower half of the schema). The Statechart representation of the FRAME_COUNTER variable is presented with the direction of data transfer from EXTERNAL into the ARSP Module. Its type and value range are defined in the Statemate data dictionary.

NL-Based SRS

Module Specification

Data Dictionary

INPUT

AR_ALTITUDE	AR_COUNTER
AR_FREQUENCY	AR_STATUS
FRAME_COUNTER	K_ALT

OUTPUT

AR_ALTITUDE	AR_STATUS
K_ALT	

PROCESS:

It is only necessary that this functional module ...

NAME: **FRAME_COUNTER**

DESCRIPTION: Counter containing the number of the present frame

USED IN: AECLP, ARSP, CP, GP, TDLRSP

UNITS: none

RANGE: $[1, 2^{31}-1]$

DATA TYPE: Integer*4

ATTRIBUTE: data

DATA STORE LOCATION: EXTERNAL

ACCURACY: N/A

Z Specification

```

_____ ARSP_RESOURCE _____
① FRAME_COUNTER? : N
② AR_FREQUENCY? : R
③ AR_COUNTER? : Z
④ K_ALT_1, K_ALT_2, K_ALT_3, K_ALT_4, K_ALT_NEW: {0,1}
⑤ AR_ALTITUDE_1, AR_ALTITUDE_2, AR_ALTITUDE_3, AR_ALTITUDE_4,
  AR_ALTITUDE_NEW: R
⑥ AR_STATUS_1, AR_STATUS_2, AR_STATUS_3, AR_STATUS_4,
  AR_STATUS_NEW: {0,1}
⑦ K_ALT: K_ALT_NEW × K_ALT_1 × K_ALT_2 × K_ALT_3 × K_ALT_4
⑧ AR_STATUS: AR_STATUS_NEW × AR_STATUS_1 × AR_STATUS_2 ×
  AR_STATUS_3 × AR_STATUS_4
⑨ AR_ALTITUDE: AR_ALTITUDE_NEW × AR_ALTITUDE_1 × AR_ALTITUDE_2 ×
  AR_ALTITUDE_3 × AR_ALTITUDE_4
⑩ AR_COUNTER? ∈ -1..32767
⑪ AR_FREQUENCY? ∈ 1..2450000000
⑫ FRAME_COUNTER? ∈ 1..2147483647
⑬ AR_ALTITUDE_1 == 1..2000 ∧ AR_ALTITUDE_2 == 1..2000 ∧
  AR_ALTITUDE_3 == 1..2000 ∧ AR_ALTITUDE_4 == 1..2000 ∧
  AR_ALTITUDE_NEW == 1..2000
    
```

Statecharts

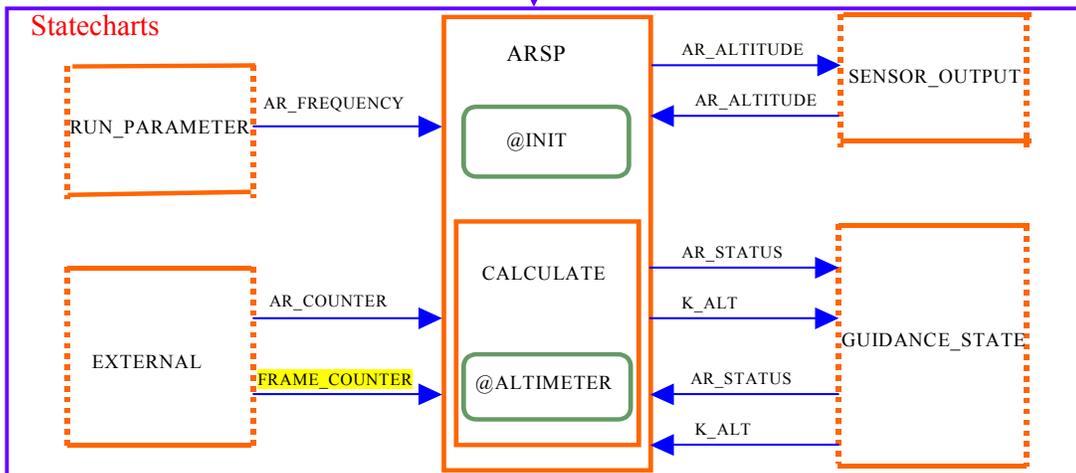


Figure 8. Translation example from NL-based to statecharts

In translating from the NL-based SRS to Z, four different requirements were identified as being ambiguous. The first ambiguous requirement concerns the rotational direction assumed by the use of the term “rotate.” Secondly, an undefined third order polynomial was revealed that is used to estimate the AR_ALTITUDE value. The third issue (i.e., ambiguity) concerns the use of the AR_COUNTER variable for two different purposes, which implies that it has two different types. Finally, there is uncertainty regarding the scope of the AR_COUNTER variable that brings into question which module should use and/or modify this variable.

Given these various issues, two scenarios were considered. The first scenario assumes the AR_COUNTER is updated within the ARSP module while the second scenario does not. Both scenarios were constructed separately and compared to understand how Z could be useful in clarifying ambiguity and avoiding conflicts. In the SRS, the sign bit of AR_COUNTER represents whether the radar echo pulse is received on time. In scenario one, this condition is split off into the Echo variable while in scenario two the Echo variable is not introduced. The Z specification is consistent with the SRS as long as the newly defined Echo variable does not cause a side affect outside of the ARSP module. Accordingly, the Z version of the ARSP specification was defined to account for two separate variables. As the result of the process, the Echo variable was found to be treated as an additional ARSP input, otherwise there is no way to determine if the radar echo pulse has been received. This in turn caused the whole specification to be revised to reflect the principle that mandates decoupling data [3]. Therefore, the interpretation of Scenario One is inconsistent with the SRS.

On the other hand, in Scenario Two (details described in chapter 4.3.1) no additional variables were defined. Only those variables defined in the SRS were specified, and all the requirements specified in ARSP were covered. Therefore, this reformulation of the SRS in Z

was considered complete and consistent. Consequently, Statecharts were developed based on Scenario Two.

4.3.1 Z Specification

The second Z scenario of the ARSP module is described here. The only assumption in this scenario is that the AR_COUNTER value must be updated from outside of the ARSP module and is ready for immediate use. When the AR_COUNTER value is -1 this indicates that the echo of the radar pulse has not yet been received. If the AR_COUNTER value is a positive integer, this means that the echo of the radar pulse arrived at the time indicated by the value of the counter.

The ARSP_RESOURCE schema (Figure 9) defines the ARSP module input and output variables. The FRAME_COUNTER? (Signature [Sig.] ①) is an input variable giving the present frame number and is typed as a natural number. AR_FREQUENCY? (Sig. ②) represents the rate at which the AR_COUNTER? has been incremented and is typed as a real number. The AR_COUNTER? (Sig. ③) is an input variable that is used to determine the AR_ALTITUDE value and its type is an integer. The K_ALT_1, K_ALT_2, K_ALT_3, K_ALT_4, and K_ALT_NEW (Sig. ④) variables are defined as sets of binary elements. The AR_ALTITUDE_1, AR_ALTITUDE_2, AR_ALTITUDE_3, AR_ALTITUDE_4, and AR_ALTITUDE_NEW (Sig. ⑤) are defined as a set of real numbers to represent the altitude that is determined by altimeter radar. AR_STATUS_1, AR_STATUS_2, AR_STATUS_3, AR_STATUS_4, and AR_STATUS_NEW (Sig. ⑥) are defined as binary values that represent health status for the various elements of the altimeter radar. The AR_STATUS, AR_ALTITUDE, and K_ALT (Sig.s ⑦-⑨) arrays hold the previous 4 values of their elements respectively.

The AR_STATUS, AR_ALTITUDE, and K_ALT variables were defined as a 5-element array in the SRS. Z does not have a specific array construct so these variables are designed as 5-element Cartesian products. The array can also be represented as a 5-element sequence. The Cartesian product method was chosen because this composition assumes that any element can be accessed directly without having to search through the sequence. The predicates ❶, ❷, and ❸ represent the variables ranges. The predicate ❹ restricts the values for the sets in the Signature ❺.

ARSP_RESOURCE	
❶	FRAME_COUNTER? : \mathbb{N}
❷	AR_FREQUENCY? : \mathbb{R}
❸	AR_COUNTER? : \mathbb{Z}
❹	K_ALT_1, K_ALT_2, K_ALT_3, K_ALT_4, K_ALT_NEW: {0,1}
❺	AR_ALTITUDE_1, AR_ALTITUDE_2, AR_ALTITUDE_3, AR_ALTITUDE_4, AR_ALTITUDE_NEW: \mathbb{R}
❻	AR_STATUS_1, AR_STATUS_2, AR_STATUS_3, AR_STATUS_4, AR_STATUS_NEW: {0,1}
❼	K_ALT: $K_ALT_NEW \times K_ALT_1 \times K_ALT_2 \times K_ALT_3 \times K_ALT_4$
❽	AR_STATUS: $AR_STATUS_NEW \times AR_STATUS_1 \times AR_STATUS_2 \times AR_STATUS_3 \times AR_STATUS_4$
❾	AR_ALTITUDE: $AR_ALTITUDE_NEW \times AR_ALTITUDE_1 \times AR_ALTITUDE_2 \times AR_ALTITUDE_3 \times AR_ALTITUDE_4$
❶	AR_COUNTER? $\in -1..32767$
❷	AR_FREQUENCY? $\in 1..2450000000$
❸	FRAME_COUNTER? $\in 1..2147483647$
❹	AR_ALTITUDE_1 == 1..2000 \wedge AR_ALTITUDE_2 == 1..2000 \wedge AR_ALTITUDE_3 == 1..2000 \wedge AR_ALTITUDE_4 == 1..2000 \wedge AR_ALTITUDE_NEW == 1..2000

Figure 9. ARSP_RESOURCE schema

The ARSP schema (Figure 10) is the main functional schema of the ARSP module. The ARSP_RESOURCE schema is imported (and is modified) in the Signature ❶. The Altitude_Polynomial function (Sig. ❷) obtains the AR_ALTITUDE as input and estimates the

current altitude by fitting a third-order polynomial to the previous value of the AR_ALTITUDE. AR_STATUS_Update (Sig. ③), K_ALT_Update (Sig. ④), and AR_ALTITUDE_Update (Sig. ⑤) update AR_STATUS, K_ALT, and AR_ALTITUDE array with their _NEW values respectively. The expression “FRAME_COUNTER? mod 2” is used on 7 occasions to determine if the FRAME_COUNTER? is odd or even.

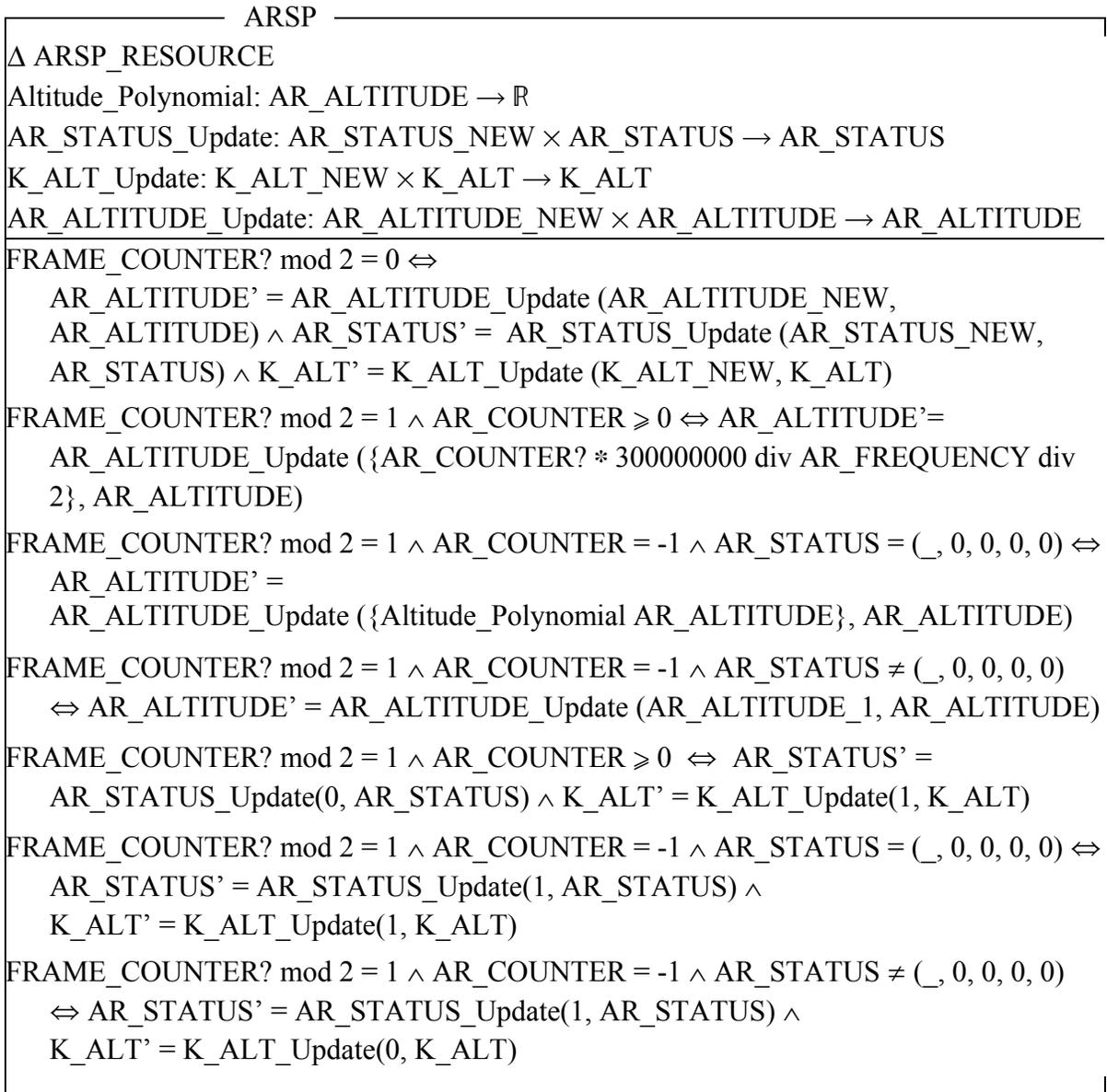


Figure 10. ARSP schema

Predicate ❶ requires that the current AR_ALTITUDE, AR_STATUS, and K_ALT element values be the same as the predecessors when FRAME_COUNTER? is even. Predicate ❷ constraints the AR_ALTITUDE update. The update takes the current value, calculated by the Eq. 1, when FRAME_COUNTER? is odd and AR_COUNTER? is greater than or equal to zero. Predicate ❸ states that the AR_ALTITUDE value is updated (i.e., estimated) by the Altitude_Polynomial function. This is done when FRAME_COUNTER? is odd, AR_COUNTER? is -1, and all the AR_STATUS elements are healthy.

Predicate ❹ requires that the current value in AR_ALTITUDE be the same as the previous values when FRAME_COUNTER? is odd, AR_COUNTER? is -1 and any of the elements in AR_STATUS are not healthy. Predicate ❺ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd and the AR_COUNTER? is -1. Predicate ❻ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd, the AR_COUNTER? is -1, and all of the AR_STATUS elements are healthy. Predicate ❼ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd, AR_COUNTER? is -1, and any of the elements in AR_STATUS is not healthy.

4.3.2 Statecharts

The state/activity charts in Statemate environments of the Z specification is described in this section. The ARSP Activity-chart (Figure 11) shows the data flow between the data stores and the ARSP module. The data flows are directed as specified in the data dictionary of the NL-based SRS. The “@INIT” control state in the ARSP activity chart represents the link to the INIT Statechart. Each activity is allowed to have only one control state. The control state can be a superstate or AND/OR decomposed state.

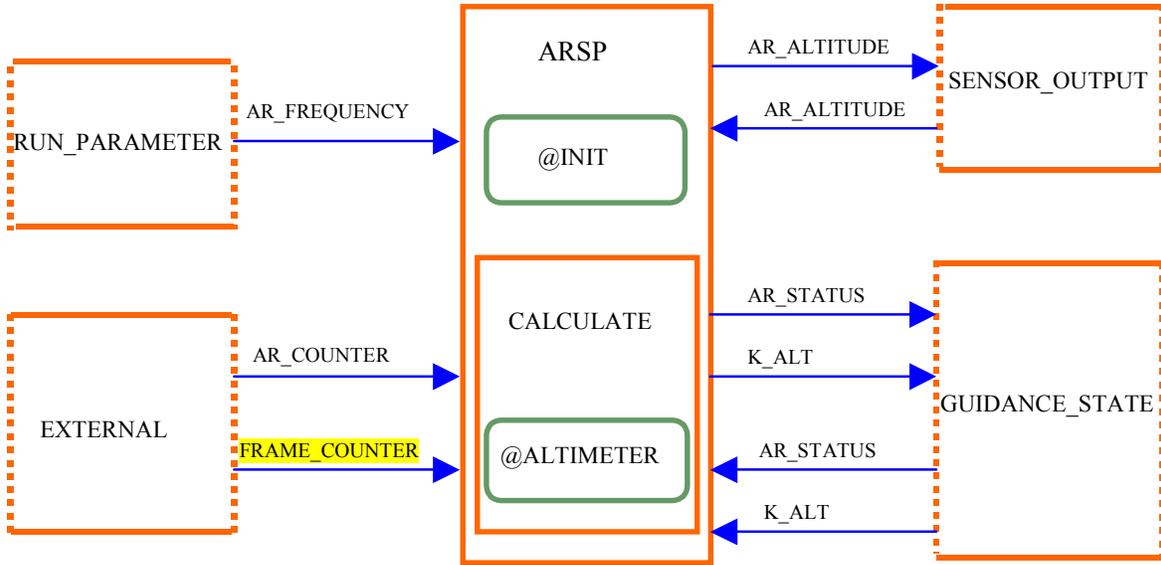


Figure 11. ARSP activity-chart

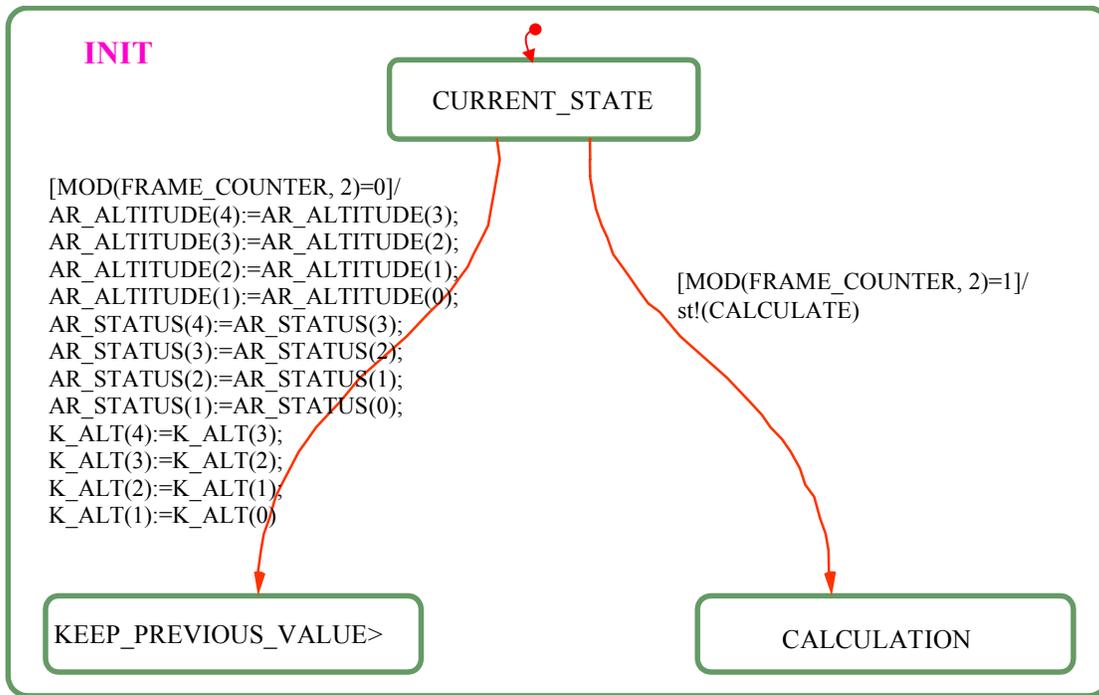


Figure 12. INIT statechart

INIT Statechart (Figure 12) shows the initialization of the ARSP module and a portion of the ARSP operational schema (Figure 10). The default transition activates the CURRENT_STATE

when the ARSP activity of the ARSP activity chart is begun. The transition from the CURRENT_STATE state to KEEP_PREVIOUS_VALUE state describes predicate ❶ of Figure 10. The KEEP_PREVIOUS_VALUE state is one of the module termination states. The termination states are marked with “>” at the end of the state name. The transition from the CURRENT_STATE to the CALCULATION state represents a condition where the value of FRAME_COUNTER is odd which is described as “FRAME_COUNTER mod 2 = 1” in Figure 10.

The Altimeter Statechart (Figure 13) is represented by the “@ALTIMETER” control activity of the ARSP activity chart. The ODD state is activated by the default transition when the CALCULATION activity of the ARSP activity chart is begun. The transition from the ODD state to the ESTIMATE_ALTITUDE state occurs when the AR_COUNTER value is set to -1 and all the elements of the AR_STATUS value are set to “healthy.” When this transition begins the AR_STATUS and K_ALT values will be updated as described by predicate ❸ of Figure 10. The 0 (zero) value of the AR_STATUS means “healthy” which corresponds to the value given in the SRS data dictionary [25]

The transition from the ODD state to the CALCULATE_ALTITUDE state begins when a positive value of the AR_COUNTER is given which is equivalent to predicate ❹ of Figure 10. The transition from the ODD to the KEEP_PREVIOUS state is triggered when the AR_COUNTER value is set to -1 and at least one of the AR_STATUS elements is not healthy. This transition has the same meaning as predicate ❺ in Figure 10. The transition from the ESTIMATE_ALTITUDE state to the DONE state happens when the ESTIMATION_FINISHED event occurs. This process is represented as an *event* because the transaction is described as an undefined third-order polynomial estimation in the SRS. The transaction from the

CALCULATE_ALTITUDE state to the DONE state denotes predicate ② (Figure 10). The transaction from the KEEP_PREVIOUS state to the DONE state denotes the predicate ④ (Figure 10) operation.

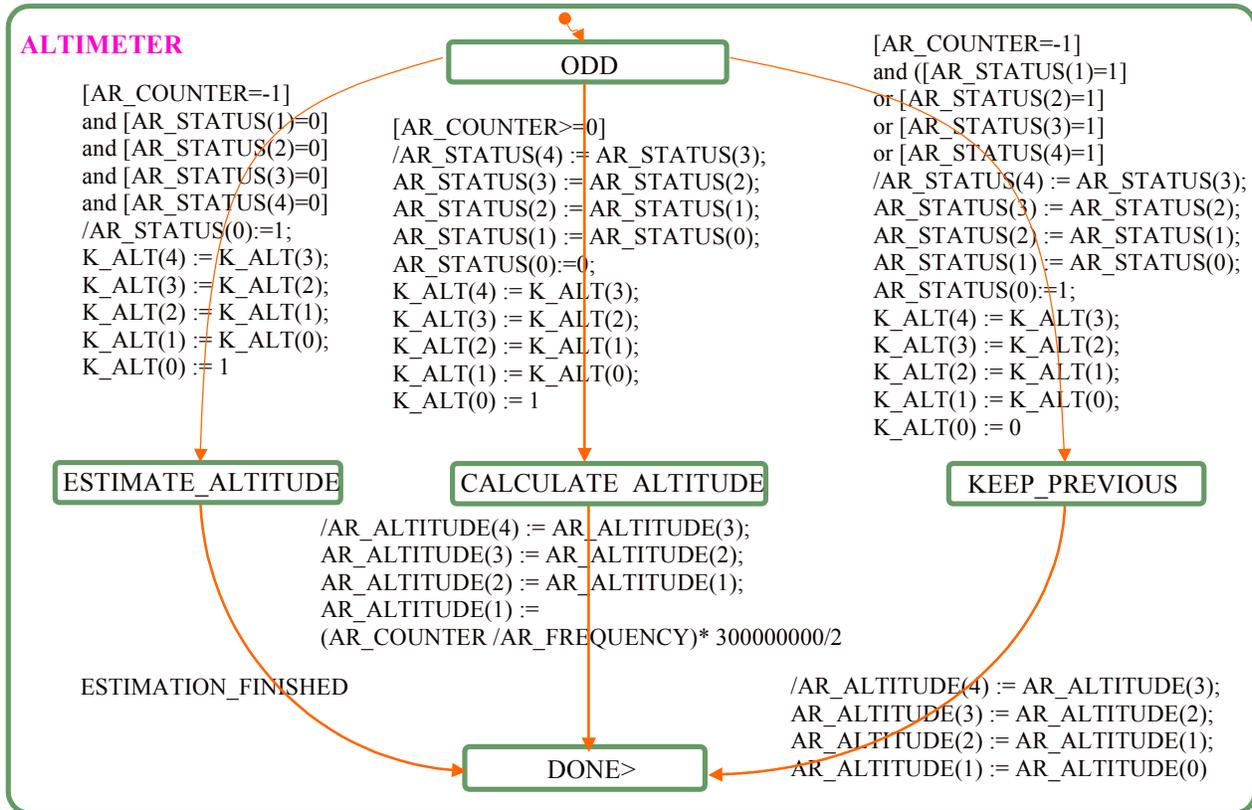


Figure 13. ALTIMETER statechart

4.3.3 Tests

Now the results of this validation effort must be discussed based on a symbolic simulation of the GCS Statechart model. In effect, the ARSP submodule requirements are verified complete and consistent by running the simulation against all of the Activity/Statecharts in the StateMate environments. Two specification tests results are presented in this section.

4.3.3.1 Finite State machine approach

There are four possible paths of activity/state transition in the ARSP statecharts model. *Path 1* represents the ARSP module's reaction when the FRAME_COUNTER is even. *Path 2* is when the updated FRAME_COUNTER is an odd value, the radar echo pulse is *not* yet received, and all the AR_STATUS elements' values are healthy. *Path 3* is when the updated FRAME_COUNTER is an odd value, the radar echo pulse is received, and all the AR_STATUS elements' values are healthy. *Path 4* is when the updated FRAME_COUNTER value is odd, the echo is not arrived, and one or more of the AR_STATUS elements' values are not healthy.

The simulation results in Table 3 shows the order of the activities/states entered for each path. One can conclude that the ARSP statecharts model does not have any absorbing state/activity. Therefore, the statecharts model is consistent.

Table 3. ARSP specification simulation result

Name of Chart	Activity / State Name	Activity/State Transition Paths			
		1	2	3	4
ARSP	ARSP	E ₁	E ₁	E ₁	E ₁
	@INIT	E ₂	E ₂	E ₂	E ₂
	CALCULATE	-	E ₅	E ₅	E ₅
	@ALTIMETER	-	E ₆	E ₆	E ₆
INIT	CURRENT_STATE	E ₃	E ₃	E ₃	E ₃
	KEEP_PREVIOUS_VALUE>	E ₄	-	-	-
	CALCULATION	-	E ₄	E ₄	E ₄
ALTIMETER	ODD	-	E ₇	E ₇	E ₇
	ESTIMATE_ALTITUDE	-	E ₈	-	-
	CALCULATE_ALTITUDE	-	-	E ₈	-
	KEEP_PREVIOUS	-	-	-	E ₈
	DONE>	-	E ₉	E ₉	E ₉

E_i entered in *i*th order, - not activated.

4.3.3.2 Data Item approach

Five test cases (Case 1-5) as shown in Table 4 were defined to execute and test the statecharts. They represent the way the Z specification is visualized and evaluated. Based on the given equations in the NL-based SRS, the input and output values are calculated. The AR_FREQUENCY variable is used to process AR_ALTITUDE value (represented as a state transition from the CALCULATE_ALTITUDE state to the DONE> state in Figure 13). This variable is defined as a real number and has a very large range. For that reason, the AR_FREQUENCY variable is not used as a system state variable in the statecharts model. Instead, its value is fixed as a constant. To calculate the expected output value of AR_ALTITUDE, the AR_FREQUENCY value is fixed at 1,500,000,000 for all test cases. The material presented below shows how each of the conditions was evaluated and this should help to convince the reader that the ARSP subunit is significantly complex (one of six different sensor units used by the GCS).

The values of the ARSP input/output variables are given in Table 4. The contents of the Table 5 represent the highlighted column of the Table 4 in detail. In the test case1, for example, input variables for ARSP submodule are FRAME_COUNTER, AR_STATUS, and AR_COUNTER and their values are 2, “Don’t care”, and -1. “Don’t care” means that the AR_STATUS variable can take any values in its range. The output variables of the ARSP submodule are AR_STATUS, K_ALT, and AR_ALTITUDE. The expected values of each of the output variables are depend on the module inputs and its value before the execution. The expected values of the output variables are calculated outside of the simulation. The “After execution” values (shown in Table 5) of the output variables represent the actual outputs from the statecharts model simulation. The test results are concluded correct when the expected values

and the after execution values match. All of the output values for all the test cases are the same as expected (as shown in Table 4). All of the variables were updated as expected. Therefore, the result of this simulation shows the previous Z specification was developed correctly.

Table 4. ARSP specification test input and output

	Variable	Case 1	Case 2	Case 3	Case 4	Case 5
Input	FRAME_COUNTER	2	2	1	1	3
	AR_STATUS	-	-	[0, 0, 0, 0, 0]	-	[0, 0, 1, 0, 0]
	AR_COUNTER	-1	19900	-1	20000	-1
Expected Output	AR_STATUS	KP	KP	[1, 0, 0, 0, 0]	[0, -, -, -, -]	[1, 0, 0, 1, 0]
	K_ALT	KP	KP	[1, 1, 1, 1, 1]	[1, -, -, -, -]	[0, 1, 1, -, 1]
	AR_ALTITUDE	KP	KP	[* , -, -, -, -]	[2000,-,-,-,-]	KP
Actual Output	AR_STATUS	KP	KP	[1, 0, 0, 0, 0]	[0, -, -, -, -]	[1, 0, 0, 1, 0]
	K_ALT	KP	KP	[1, 1, 1, 1, 1]	[1, -, -, -, -]	[0, 1, 1, -, 1]
	AR_ALTITUDE	KP	KP	[* , -, -, -, -]	[2000,-,-,-,-]	KP

- Don't care, KP Keep Previous value, * An estimated value.

Table 5. Detailed testing results – Case 1 example

	Variable	Case 1		
		Before the execution	Expected values	After the execution
Input	FRAME_COUNTER	2	2	2
	AR_STATUS	-	-	-
	AR_COUNTER	-1	-1	-1
Output	AR_STATUS	[1,0,0,0,0]	[1,1,0,0,0]	[1,1,0,0,0]
	K_ALT	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]
	AR_ALTITUDE	[2000, -, -, -, -]	[2000, 2000, -, -, -]	[2000, 2000, -, -, -]

- Don't care.

4.3.4 Fault Injection

At this step, simulation of the specification is used for discovering hidden faults and their location. To accomplish this, faults are injected into the model to simulate memory corruption (i.e., expected due to the harsh space born lander mission environment.)

Four new issues arose during the fault injection process. (1) Some correct inputs produced incorrect outputs; (2) The Statecharts approach has a better chance of predicting possible faults in the system. (Because the Z specification cannot provide a way of predicting the transitions from state to state i.e., Z is not executable); (3) During the symbolic simulation, some weak points were found where faults were lurking (e.g., errors described in Appendix C in [32]); (4) Consequently, there are many design decisions to be made in the process of developing a model (i.e., specification). Finding the correct formulation is a process of refinement and validation, which was facilitated using this approach combined with symbolic simulation. Some requirements were found to be inconsistent/incomplete because they produced incorrect results.

For example, one can alter a system state variable (i.e., FRAME_COUNTER) at a certain state (i.e., CURRENT_STATE) during the simulation (i.e., for test case 1). Table 6 shows the fault injection results of the FRAME_COUNTER alteration at CURRENT_STATE while running test case 1. The expected values of the output variables are not the same as the actual values of the output due to the state variable change. This means the highlighted **x** mark in the Table 7.

Table 7 shows 120 of the specification testing results using fault injection. The fault injection states are the states defined in the Statecharts model. According to the result table, the “CURRENT_STATE” does not tolerate any of the injected faults. In addition, the fault injection in the CALCULATION and ODD system states produce erroneous outputs. Therefore, one can conclude these three system states are the most vulnerable states.

Table 6. Detailed fault injection results – Case 1 example

	Variable	Case 1		
		Before the execution	Expected values	After the execution
Input	FRAME_COUNTER	2	2	2
	AR_STATUS	-	-	-
	AR_COUNTER	-1	-1	-1
Output	AR_STATUS	[1,0,0,0,0]	[1,1,0,0,0]	[1/0,1,0,0,0]
	K_ALT	[1,1,1,1,1]	[1,1,1,1,1]	[1,1,1,1,1]
	AR_ALTITUDE	[2000, -, -, -, -]	[2000, 2000, -, -, -]	[*, 2000, -, -, -]

- Don't care, * An estimated value.

Table 7. Fault injection simulation result

Fault injected State	Altered state variable														
	FRAME_COUNTER					AR_COUNTER					AR_STATUS				
	Case					Case					Case				
	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
CURRENT_STATE	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
KEEP_PREVIOUS_VALUE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CALCULATION	✓	✓	✓	✓	✓	✓	✓	x	x	x	✓	✓	x	✓	x
ODD	✓	✓	✓	✓	✓	✓	✓	x	x	x	✓	✓	x	✓	x
ESTIMATE_ALTITUDE	✓	✓	✓	✓	✓	✓	✓	N/A	✓	✓	✓	✓	N/A	✓	✓
CALCULATE_ALTITUDE	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓
KEEP_PREVIOUS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DONE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

x incorrect outputs, ✓ no defect, N/A not applicable.

4.3.5 Reformulated Requirements

Based on the simulation results using fault injection, the SRS was discovered incomplete. To remedy the situation, the AR_FREQUENCY value must be bounded to prevent the AR_ALTITUDE value from exceeding its limit. Thus, one of the following conditions should be included: $1 \leq \text{AR_FREQUENCY} \leq \text{AR_COUNTER} * 75000$, or $\text{AR_COUNTER} = -1 \vee (0 \leq \text{AR_COUNTER} \leq \text{AR_FREQUENCY}/75000)$. In other words, one of these two relational expressions must resolve to true.

The result of this analysis revealed that it is possible to construct a complete and consistent specification using this method (Z-to-Statecharts). Ambiguous statements in the NL-based specification were revealed in the refinement process of Z specification using the test results.

The outputs from the modules were examined and shown to be consistent with the expected results by running simulations based on the Statecharts/Activity-charts. All of the state activation/transition paths were in the correct order as expected for all test cases. Moreover, no nondeterministic state transitions were detected for all simulation runs (based on the conditions provided). In this way, the simulation has provided a means for determining the consistency of the requirements.

The output values from the simulation were checked and compared against the expected values, which were calculated based on the NL-based SRS, then found to be valid. After running various simulations using fault injection, several issues were uncovered which indicate that the SRS is incomplete. In addition, some vulnerable states were identified where faults were injected into the system model (i.e., Statecharts) when the model was executed using simulations. The conclusion is that the transformed submodule would not be able to tolerate certain system faults. Therefore, these findings indicate that one can better understand the implications of the system requirements using this approach (Z-Statecharts) to facilitate their specification and analysis.

CHAPTER FIVE

RESULTS

5.1. Z (Zed)

The GCS SRS excerpt was specified in Z. The generic Z has the "?" notation representing an input variable and the "!" notation representing an output variable. The NL-based GCS SRS defines some variables as both input and output. Z does not provide a way to describe this. Therefore, those variables are represented without both "?" and "!" notations.

Z/EVES⁴ is used to verify the correctness and the consistency for the prototypes of the Z specification. There are three reasons why the Z/EVES is used for the prototype rather than the entire Z specification. First, the Z/EVES does not have the capacity of importing the entire schemas in one file. The GCS is a very complex system and the GCS excerpt (in Chapter 3) requires about 18 pages of schema to describe its data and functions. All the schemas and axioms have to be placed in a same file to examine the whole system because they are defined hierarchically. In other words, most of the schemas import previously defined schemas and axioms. Second, most of the specification patterns are repeatedly used to describe functions and data of the GCS excerpt. It is not necessary to examine the same specification patterns over and over. Last, Z/EVES does not provide some data types (i.e., R: real numbers) and functions (i.e., \times : Cross product of metrics). The correctness and consistency of data items using those data types and functions are evaluated while the Statecharts model is tested using simulations.

⁴ Z/EVES is a tool made available by ORA, Canada. It provides theorem proving, domain checking, type checking, precondition calculation, and schema expansion for Z specification.

5.1.1 Global Constants and Functions

Axioms and abbreviations are used to define global constants and functions. The abbreviation ① represents T_n is another name for a sequence of natural numbers.

$$\textcircled{1} T_n ::= \text{seq } \mathbb{N}$$

$$\textcircled{1} \frac{\text{nmatrix}: \mathbb{N} \times \mathbb{N} \rightarrow \text{seq } T_n}{\forall a: T_n; b, m, n: \mathbb{N}; t: \text{seq } T_n \mid m \neq 0 \wedge n \neq 0 \wedge \#a = m \wedge \#t = n \wedge b \in 1..n \cdot t \ b = a \wedge \text{nmatrix}(m, n) = t}$$

The axiom ① defines a global function that returns a matrix that has n rows and m columns.

The dimensions (n and m) of the matrix definition are in reverse order for making reference of the elements more conventional. For example, $[a: \text{seq } T_n \cdot a = \text{nmatrix}(3, 2)]$ creates a 2×3 matrix. When the reference convention of this matrix is as described in the Figure 14.

$\mathbf{a} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix}$	<p style="text-align: center;">When it is defined as a sequence in Z,</p> $\mathbf{a} = \langle a_1, \quad a_2, \quad a_3 \rangle$ $= \langle \langle a_{11}, a_{12} \rangle, \quad \langle a_{21}, a_{22} \rangle, \quad \langle a_{31}, a_{32} \rangle \rangle$	
<p>The row's of \mathbf{a} are referred as</p>	<p>a_1 a_2 a_3</p>	
<p>The each elements of the \mathbf{a} are referred as</p>	<p>$a_{11}, a_{12},$ $a_{21}, a_{22},$ a_{31}, a_{32}</p>	

Figure 14. Array definition with sequences

In the declaration part, the *nmatrix* function is signified to take two natural numbers and it maps those two numbers to a matrix that is represented by a sequence of T_n . The constraint part shows that the element of the return sequence is a non-empty sequence of natural numbers and the return sequence is a non-empty sequence.

Z/EVES is used to prove these definitions are correct. Figure 15 shows the proof formula from the Z/EVES and Figure 16 shows the proof window of the Z/EVES after proving the *nmatrix* function definition. Figure 17 shows the specification window after running the proof.

```

proof of nmatrix$domainCheck
prove by reduce

local nmatrix  $\in \mathbb{N} \times \mathbb{N} \rightarrow \text{seq } T\_n$ 
 $\forall (a \in T\_n \vee b \in \mathbb{N} \vee n \in \mathbb{N} \vee m \in \mathbb{N} \vee t \in \text{seq } T\_n)$ 
 $\Rightarrow (m \neq 0 \vee n \neq 0 \Rightarrow a \in \text{dom } \#)$ 
 $\vee (m \neq 0 \vee n \neq 0 \vee \# a = m \Rightarrow t \in \text{dom } \#)$ 
 $\vee (m \neq 0 \vee n \neq 0 \vee \# a = m \vee \# t = n \vee b \in 1..n$ 
 $\Rightarrow (t,b) \in \text{applies\$to} \vee (t b = a \Rightarrow (m,n) \in \text{dom } \text{local } nmatrix))$ 

```

Figure 15. Proof formula of the nmatrix function

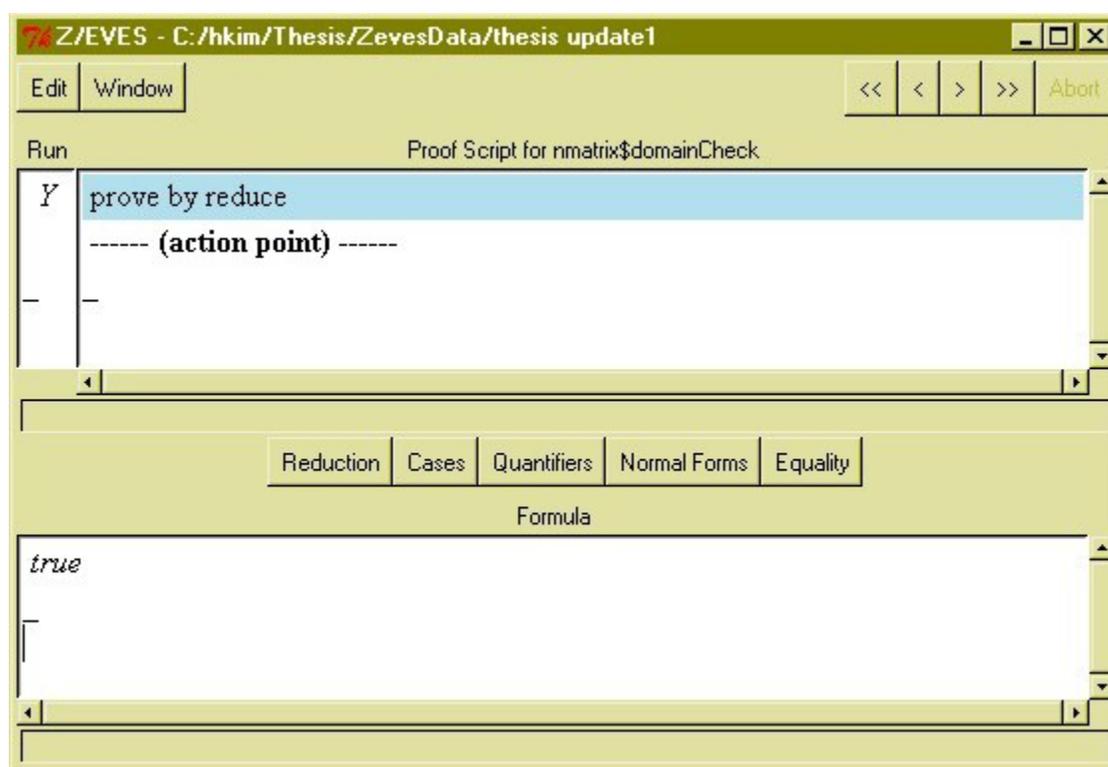


Figure 16. Z/EVES proof window

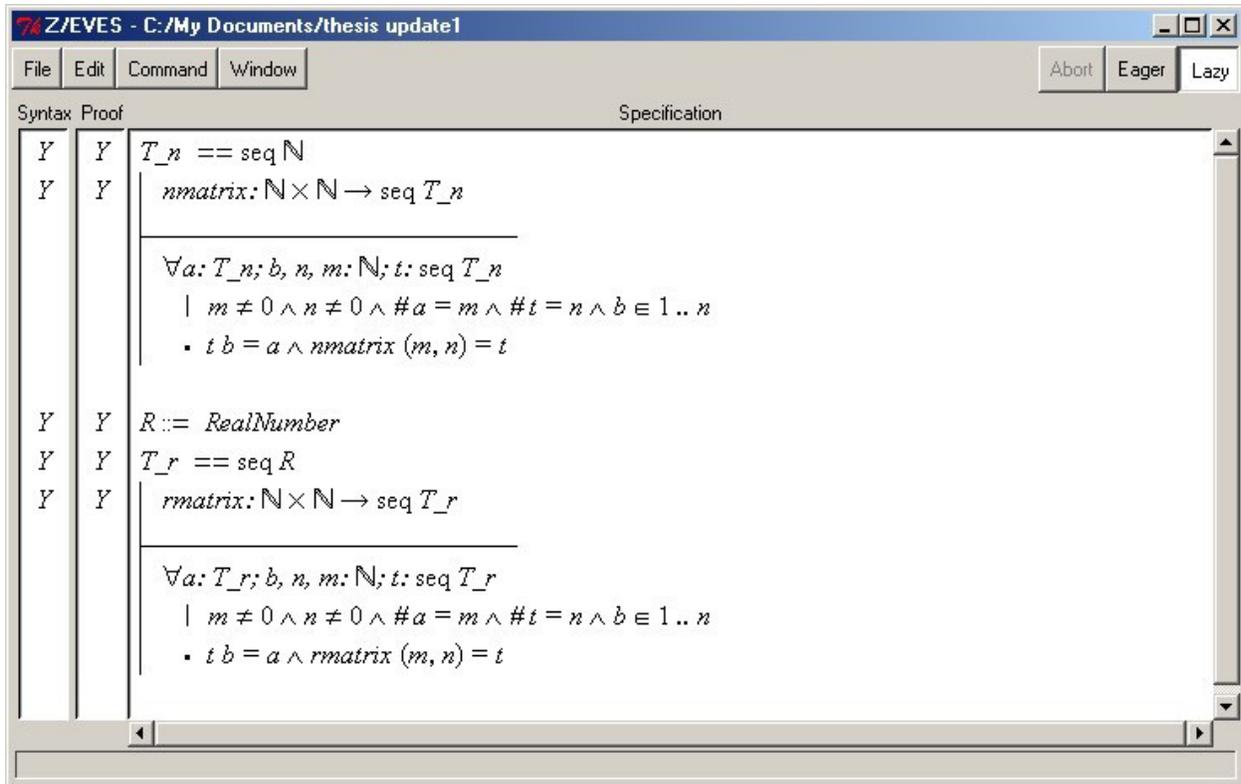


Figure 17. Z/EVES specification window

② $T_r == \text{seq } \mathbb{R}$

②
$$\frac{\text{rmatrix}: \mathbb{N} \times \mathbb{N} \rightarrow \text{seq } T_r}{\forall a: T_r; b, m, n: \mathbb{N}; t: \text{seq } T_r \mid m \neq 0 \wedge n \neq 0 \wedge \#a = m \wedge \#t = n \wedge b \in 1..n \cdot t \text{ b} = a \wedge \text{rmatrix}(m, n) = t}$$

The abbreviation ② makes T_r be another name for the sequence of real numbers. In this case, \mathbb{R} is used as a basic type representing real numbers while it is not provided as a basic type in the Z/EVES; therefore it is defined in the Z/EVES as a free type (in Figure 17).

The axiom ② defines a global function that creates a matrix that has n rows and m columns. In the declaration part, the *rmatrix* function is signified to take two natural numbers and it maps those two numbers to a matrix which represented by a sequence of T_r . The constraint part shows that the elements of the return sequence are sequences of natural numbers and the elements of the return sequence are non-empty sequences.

proof of rmatrix\$domainCheck
 prove by reduce

local $\text{rmatrix} \in \mathbb{N} \times \mathbb{N} \rightarrow \text{seq } T_r$
 $\vee (a \in T_r \vee b \in \mathbb{N} \vee n \in \mathbb{N} \vee m \in \mathbb{N} \vee t \in \text{seq } T_r)$
 $\Rightarrow (m \neq 0 \vee n \neq 0 \Rightarrow a \in \text{dom } \#)$
 $\vee (m \neq 0 \vee n \neq 0 \vee \# a = m \Rightarrow t \in \text{dom } \#)$
 $\vee (m \neq 0 \vee n \neq 0 \vee \# a = m \vee \# t = n \vee b \in 1 .. n$
 $\Rightarrow (t, b) \in \text{applies\$to} \vee (t \ b = a \Rightarrow (m, n) \in \text{dom } \text{local } \text{rmatrix}))$

Figure 18. Proof formula of the rmatrix function

The *nmatrix* function and *rmatrix* functions are the functions that create matrixes with n rows and m columns; one with natural numbers and the other with real numbers respectively. The specification patterns forming these functions are the same except the data type. Furthermore, the proof formula for the *rmatrix* is the same with the proof formula for the *nmatrix* except the data type (Figure 18). Therefore, the *nmatrix* function is considered as the prototype of the functions which form the n rows and m columns of a matrix. In this regard, the prototypes of the declarations and predicates are proved in the same way the *nmatrix* function is proved.

$T_any ::= \text{any1} \langle\langle \text{seq } \mathbb{N} \rangle\rangle | \text{any2} \langle\langle \text{seq } \mathbb{R} \rangle\rangle$

Figure 19. T_any free type definition

Figure 19 shows a definition of a free type that consists of sequences of natural numbers and/or sequences of real numbers.

$\text{INIT_DONE, RUN_DONE} : \mathbb{N}$

Figure 20. Global variable definition

Figure 20 shows global variable definitions. The variable *INIT_DONE* and *RUN_DONE* is specified as natural numbers. They are system control variables of the GCS excerpt.

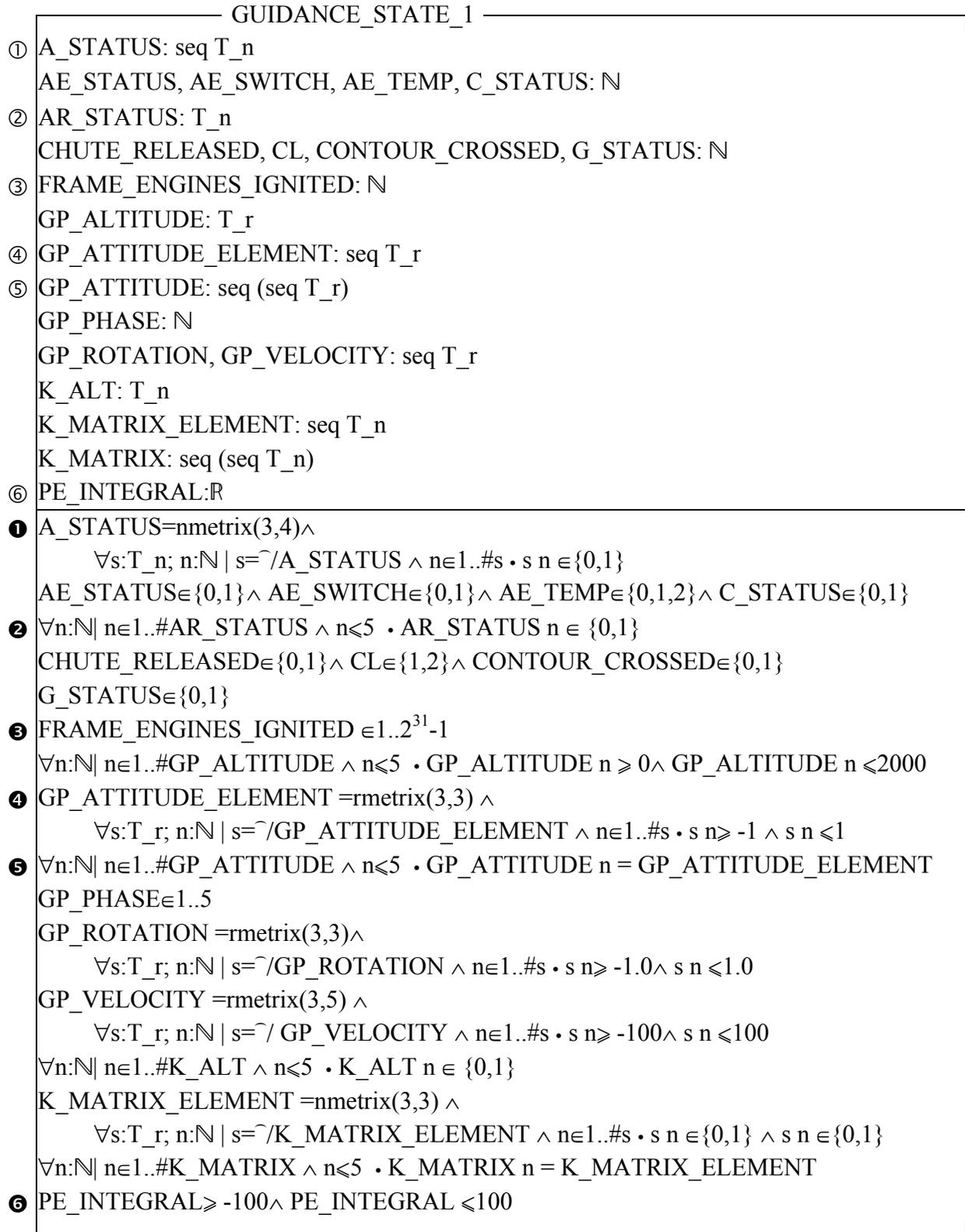


Figure 21. GUIDANCE_STATE_1 schema

The GUIDANCE_STATE_1 (Figure 21) and the GUIDANCE_STATE_2 (Figure 22) schemas are sub-schemas of the GUIDANCE_STATE schema (Figure 23). The GUIDANCE_STATE schema is separated into the two sub-schemas due to the page limit. The GUIDANCE_STATE schema represents the data store GUIDANCE_STATE in the NL-based GCS SRS. The variables used by the GCS excerpt, which are defined in the GUIDANCE_STATE data store, are specified in Figure 21 and Figure 22.

The A_STATUS (Sig. ①) is a variable defined as a natural number matrix. The predicate ❶ constrains the A_STATUS as a 4×3 matrix of which the elements are either 1 or 0. All the 2-dimensional matrix variables are defined in the same way A_STATUS is defined. The AR_STATUS (Sig. ②) is a variable defined as a sequence of natural numbers. The predicate ❷ constrains the sequence to be consists of five elements with the set of possible values $\{0, 1\}$. The AR_STATUS variable represents all the finite sequence variables with natural/real number elements. The FRAME_ENGINES_IGNITED is defined as a natural number variable. The value of the FRAME_ENGINES_IGNITED is restricted to numbers between 1 and $2^{31}-1$ (in the predicate ❸). Any variable has a natural number value defined in the same way.

The signature ④ declares a matrix of real numbers, and the constraints for the matrix are specified in the predicate ❹. The matrix (GP_ALTITUDE_ELEMENT) has 3×3 elements and the value of the elements for the matrix range from -1 to 1. The GP_ALTITUDE is a sequence of matrixes of real numbers (Sig. ⑤). In the predicate ❺, the GP_ALTITUDE is restricted to having five GP_ALTITUDE_ELEMENT variables as its elements. The specification pattern of the GP_ALTITUDE shows how the 3-dimensional matrix variable is defined in the Z specification. The PE_INTEGRAL variable (Sig. ⑥) is a real number range between -100 and 100. The predicate ❻ shows how to restrain the value range of the real number variable. All of

the variables used in the GCS excerpt have one (and only one) of the followings; a single-value natural number, a single-value real number, a finite sequence (i.e., 1-dimensional array/matrix), a 2-dimensional matrix, a 3-dimensional matrix. The variable definitions different from these are explained where they appear.

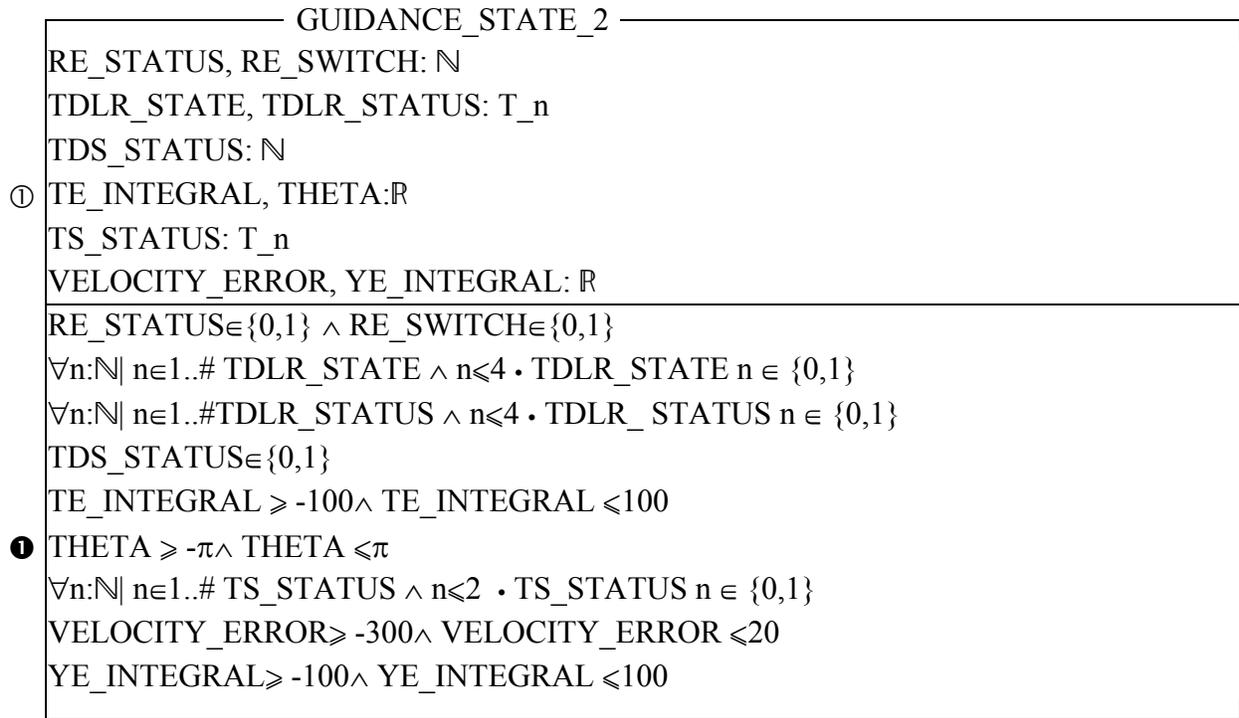


Figure 22. GUIDANCE_STATE_2 schema

In signature ①, the variable THETA is declared as a real number. The unit of this variable is the “radian” according to the NL-based GCS SRS. The symbol “ π ” is not part of the Z notations; however, it is used with the respect to the NL-based GCS SRS. Moreover, there is no other way to declare the value of π in real numbers other than using the symbol π .



Figure 23. GUIDANCE_STATE schema

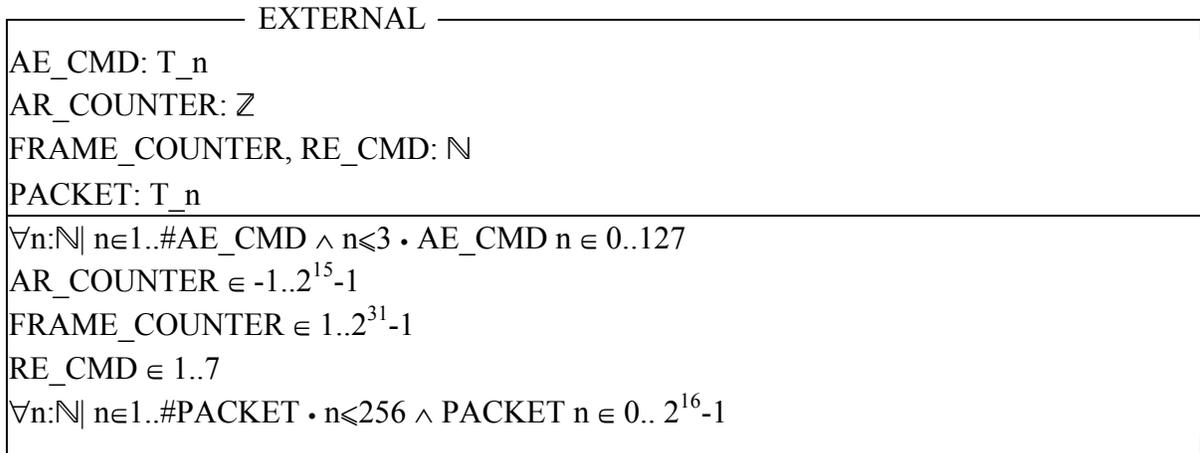


Figure 24. EXTERNAL schema

The variables in the EXTERNAL data store in the GCS SRS are specified in Figure 24. The AR_COUNTER variable (Sig. ①) is an integer of which value is limited in the predicate ① to any integer between -1 and $2^{15}-1$. The PACKET variable (Sig. ②) is declared as a sequence of natural numbers while it is defined as an array of 2-byte integers. The units and the value range of the PACKET elements are not declared in the NL-based GCS SRS; however the number of the elements is specified as 256. The PACKET is the actual data packet the GCS transmit to the orbiter. Its size and contents are determined flexibly by the Communications Processing (CP) submodule of the GCS. Therefore, the PACKET is restrained in the predicate ② to have the maximum of 256 elements of which values fit into 2 byte size. This variable has a type conflict when it is used in the CP. The further discussions about the conflict are in the chapter 5.1.3.

The RUN_PARAMETERS schema (Figure 25) specifies the variables which are used by the GCS excerpt and defined in the RUN_PARAMETERS data store of the NL-based GCS SRS.

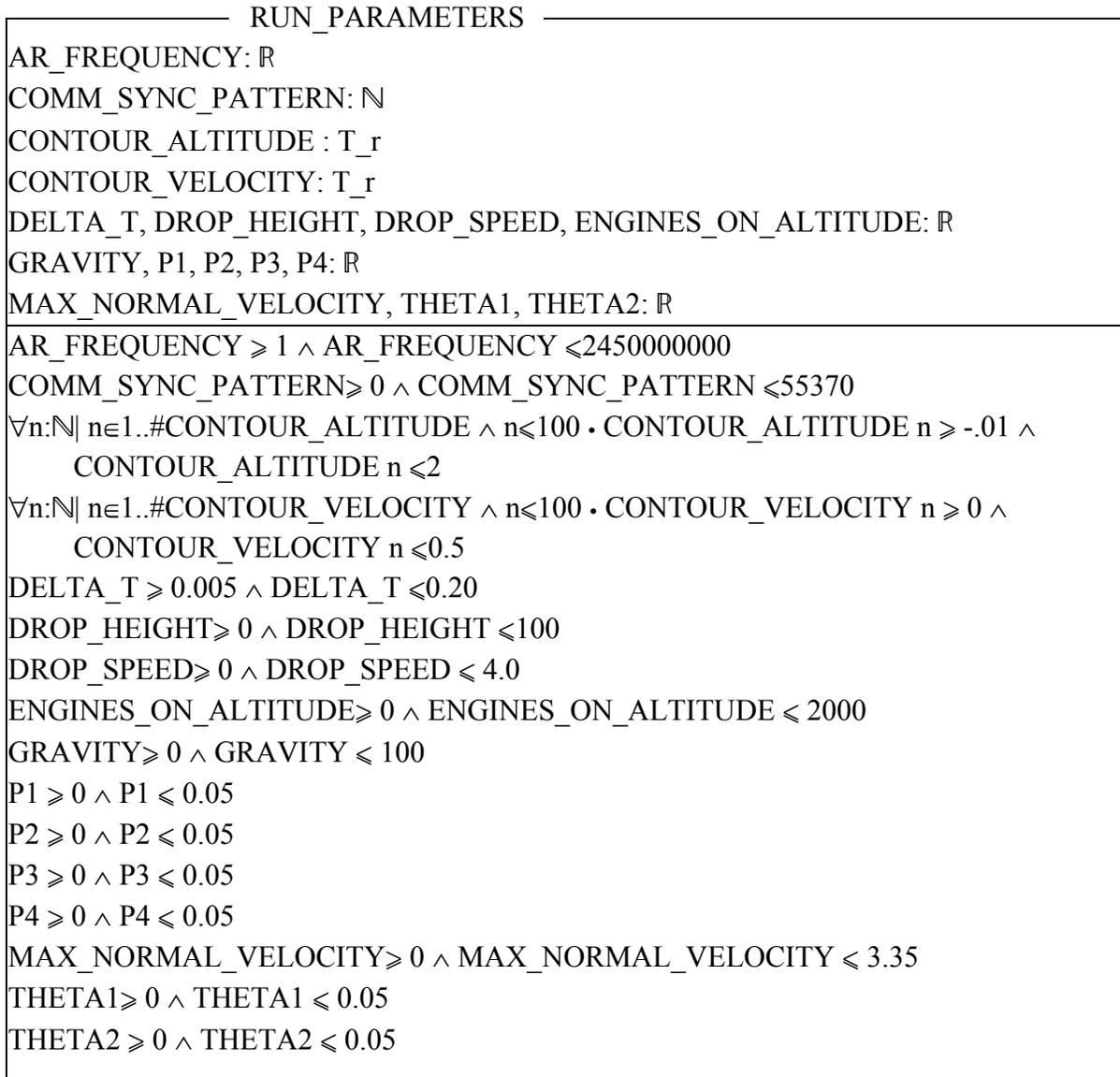


Figure 25. RUN_PARAMETERS schema

The SENSOR_OUTPUT schema (Figure 26) specifies the variables which are defined in the SENSOR_OUTPUT data store of the NL-based GCS SRS and used by the GCS excerpt.

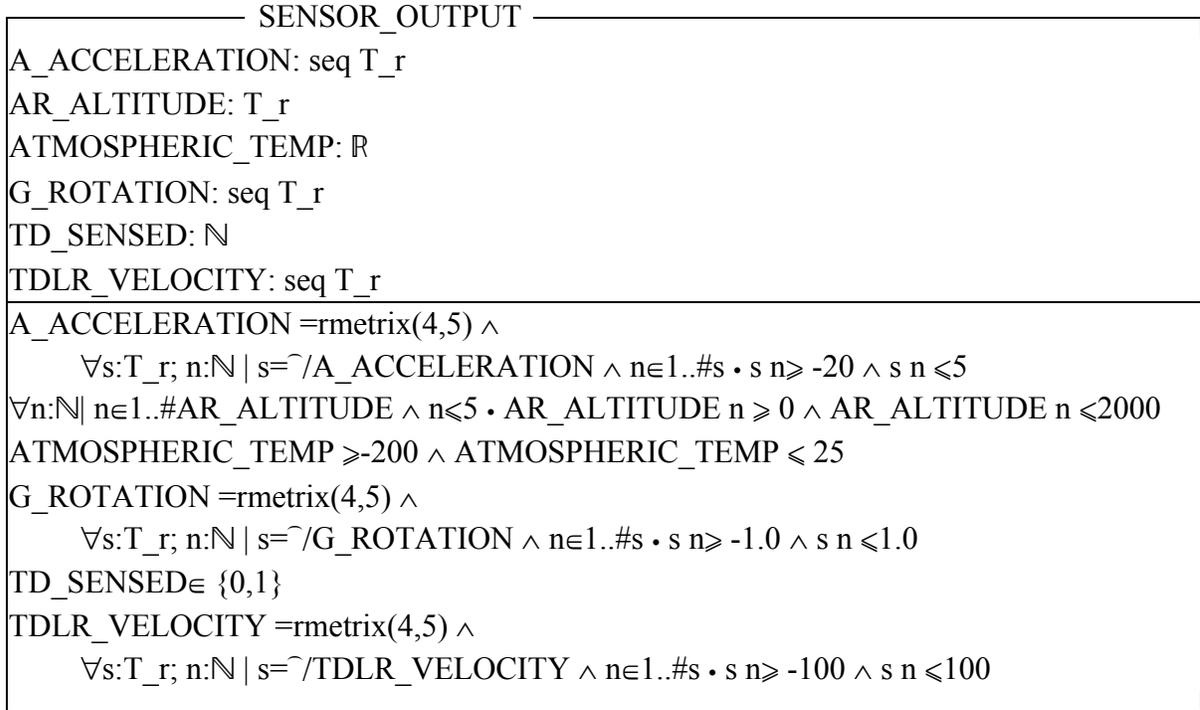


Figure 26. SENSOR_OUTPUT schema

5.1.2 ARSP Module

The Altimeter Radar Sensor Processing (ARSP) submodule is introduced in the Chapter 4.3. This chapter describes the same submodule with same functionality; however, the ARSP submodule is specified as a part of a structured system of schemas. In other words, the ARSP submodule imports data items from the previously defined schemas rather than defines all the variables locally.

The ARSP_RESOURCE schema (Figure 27) imports the RUN_PARAMETER schema and the EXTERNAL schema. “⊆” notation (Sig. ① in the Figure 27) represents that the imported schema is not being changed in the ARSP_RESOURCE schema. This means that ARSP submodule uses the variables from the imported data store but does not modify the value of those variables. “Δ” notation (Sig. ② in the Figure 27) represents that the ARSP_RESOURCE schema imports the SENSOR_OUTPUT schema for modification. This means that the ARSP submodule

takes input from the imported data store and exports outputs into the data store. The variables defined in the ARSP_RESOURCE schema is local variables.

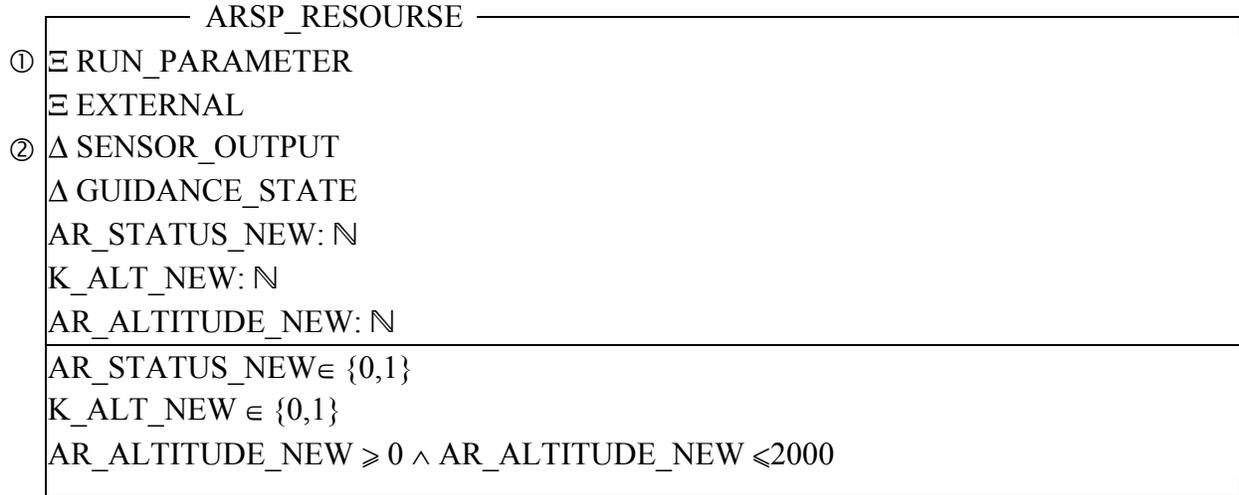


Figure 27. ARSP_RESOURCE schema

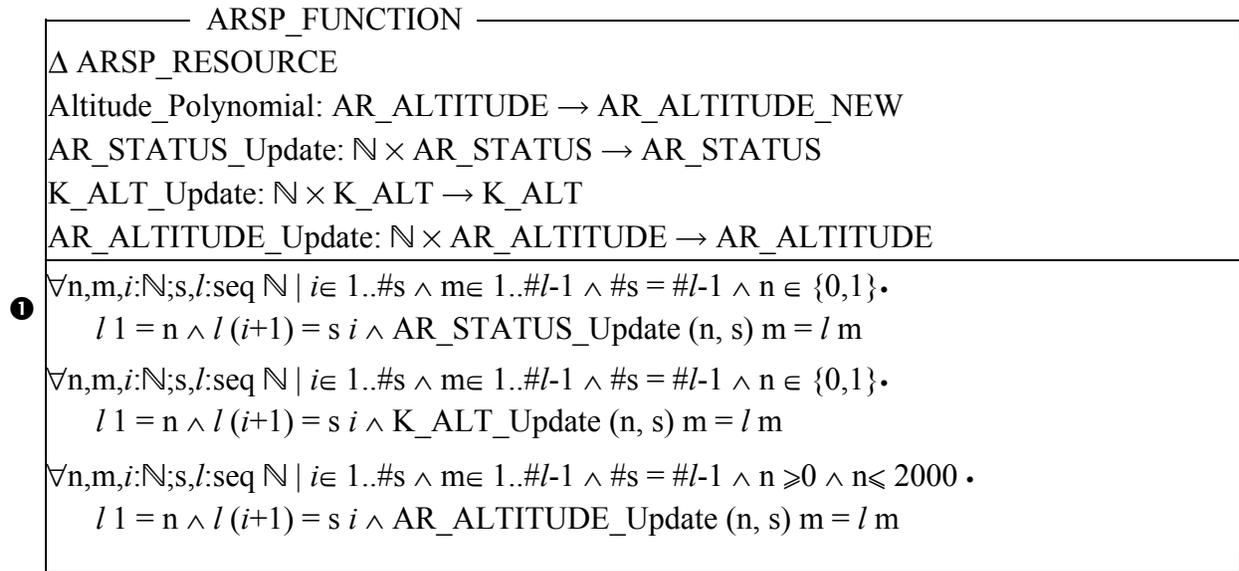


Figure 28. ARSP_FUNCTION schema

The ARSP_FUNCTION schema (Figure 28) defines functions that are used to modify variables inside the ARSP submodule. The operation of the Altitude_Polynomial is not defined because it is not defined in the NL-based SRS. The predicate ① describes the operation of the

function AR_STATUS_Update. AR_STATUS_Update function shifts the elements of the AR_STATUS by one index, and then place a new value into the index 1.

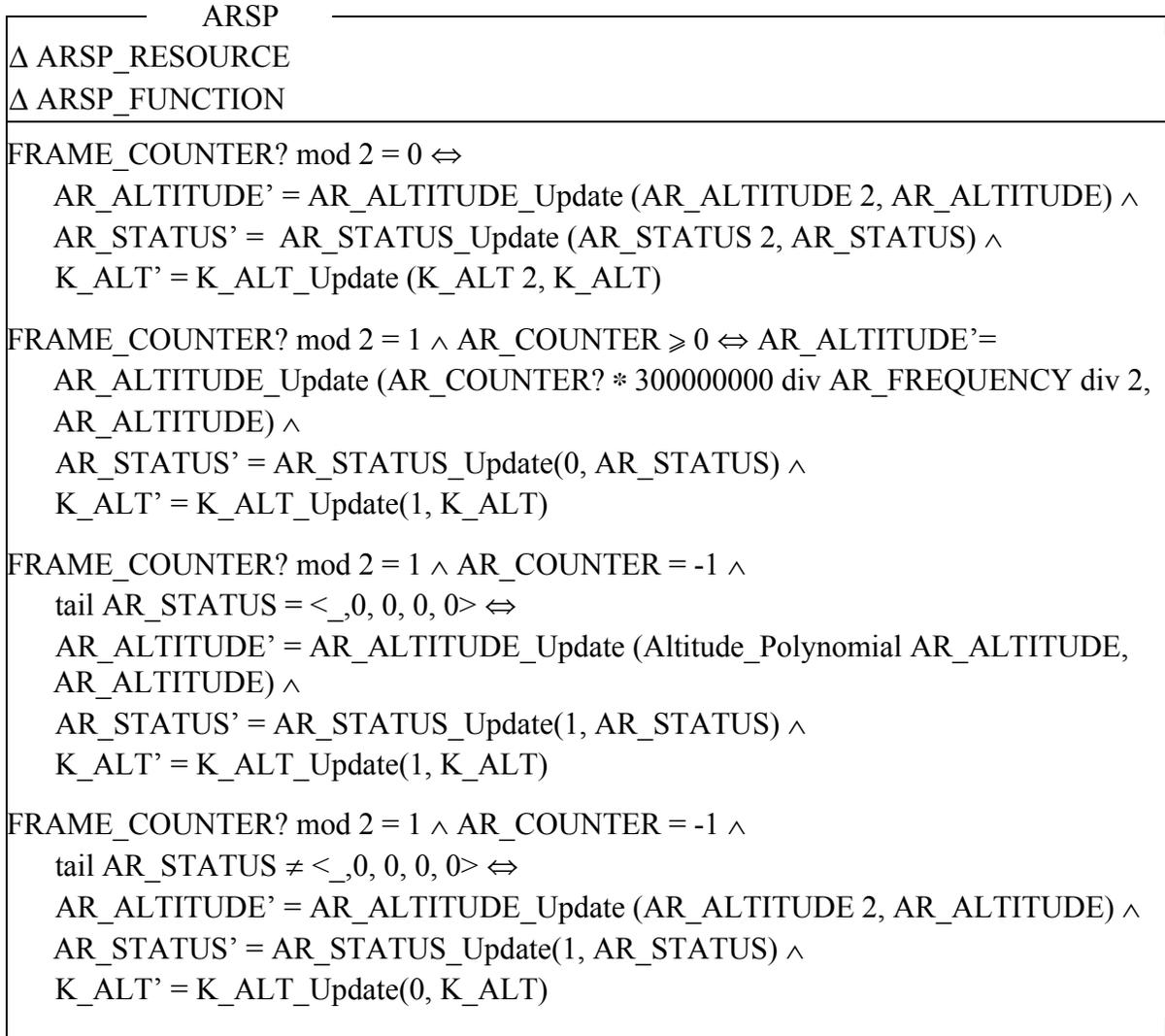


Figure 29. ARSP schema

The ARSP schema describes the process of the ARSP submodule. The contents of the ARSP schema in Figure 29 are equivalent to the contents of the ARSP schema in Figure 10. The predicate ❶ is the same as the predicated ❶ in Figure 10. The predicate ❷ represents the predicate ❷ and ❸ in Figure 10. The predicate ❸ specifies the predicate ❸ and ❹ in Figure 10. The predicate ❹ describes the predicate ❹ and ❺ in Figure 10.

The difference between the schemas in Figure 29 and Figure 10 is that the array variables are defined differently. In Figure 10, the array variables are defined as cross products while they are declared as sequences in the Figure 29.

5.1.3 CP Module

The Z specification of the Communications Processing (CP) submodule is described in this section. The CP submodule prepares a packet for transmission to the orbiting platform. The packet is consisted of a synchronization pattern, a sequence number, checksum, sample mask, and data. The variable order of the data section is given by the NL-based SRS.

The CP_RESOURCE schema (Figure 32) has two sub-schemas due to the large length of the schema definition. Those two sub-schemas, CP_RESOURCE_1 and CP_RESOURCE_2, have definitions of the local variables that are used in CP submodule schemas. They show that CP takes input from all four data store and places its outputs into the EXTERNAL and GUIDANCE_STATE data stores only.

In every subframe, GCS modifies variables and is required to transmit only the changed values of the variables since the last transmission. Therefore, it is necessary that CP keeps record of the previous value to compare with current value of those variables which are required to be transmitted. There are several variables defined to keep its history (i.e., array variables with time dimension). However, all the single value variables need an extra data store to preserve the previous values which are not defined in the NL-based SRS. This means the NL-based SRS is incomplete because it misses those variable definitions to be used. For this case study, local variables are defined to preserve the data history. Those variables are named starting with “P_” and followed by a variable name that is defined in the data store (in CP_RESOURCE_1 and CP_RESOURCE_2 schemas).

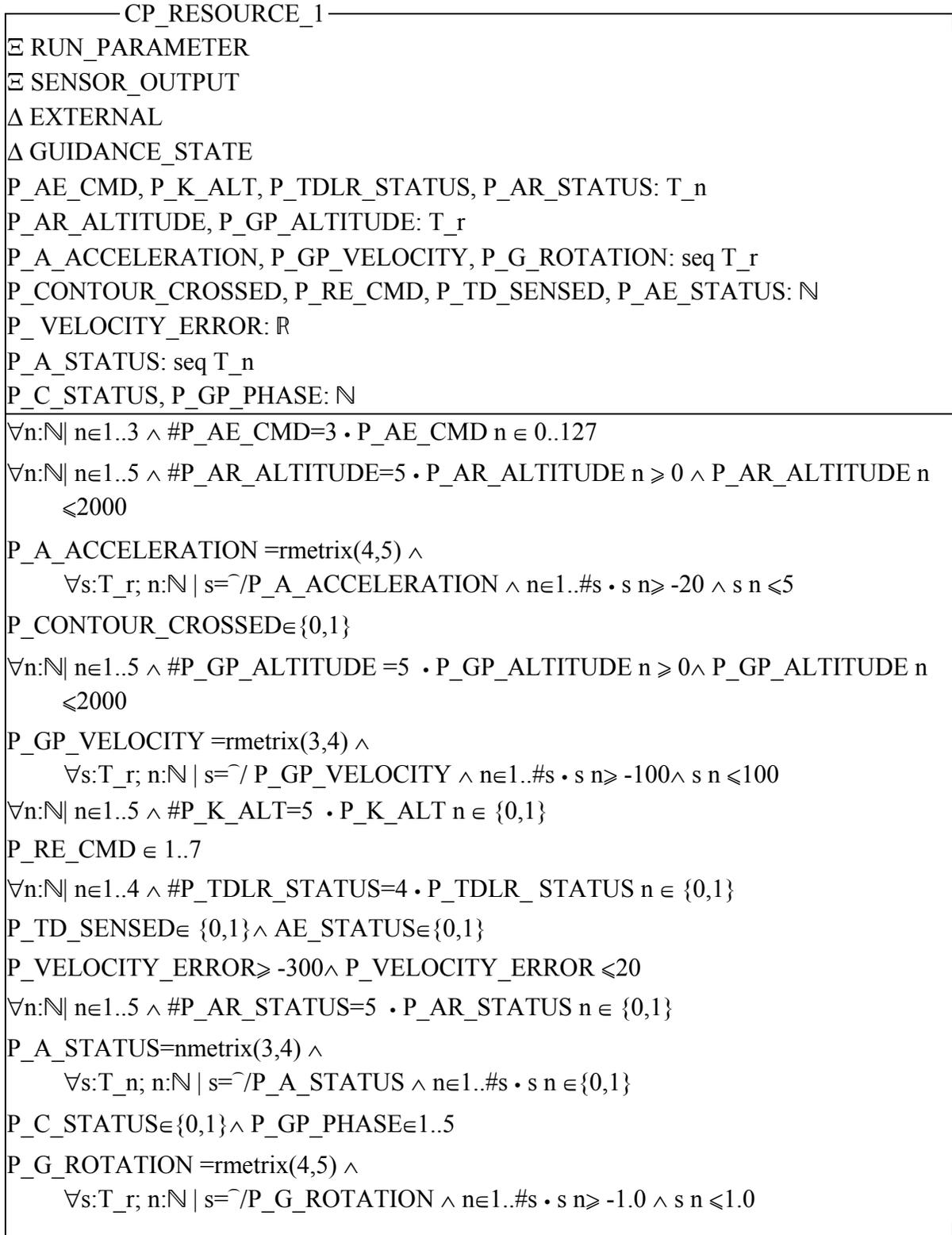


Figure 30. CP_RESOURCE_1 schema

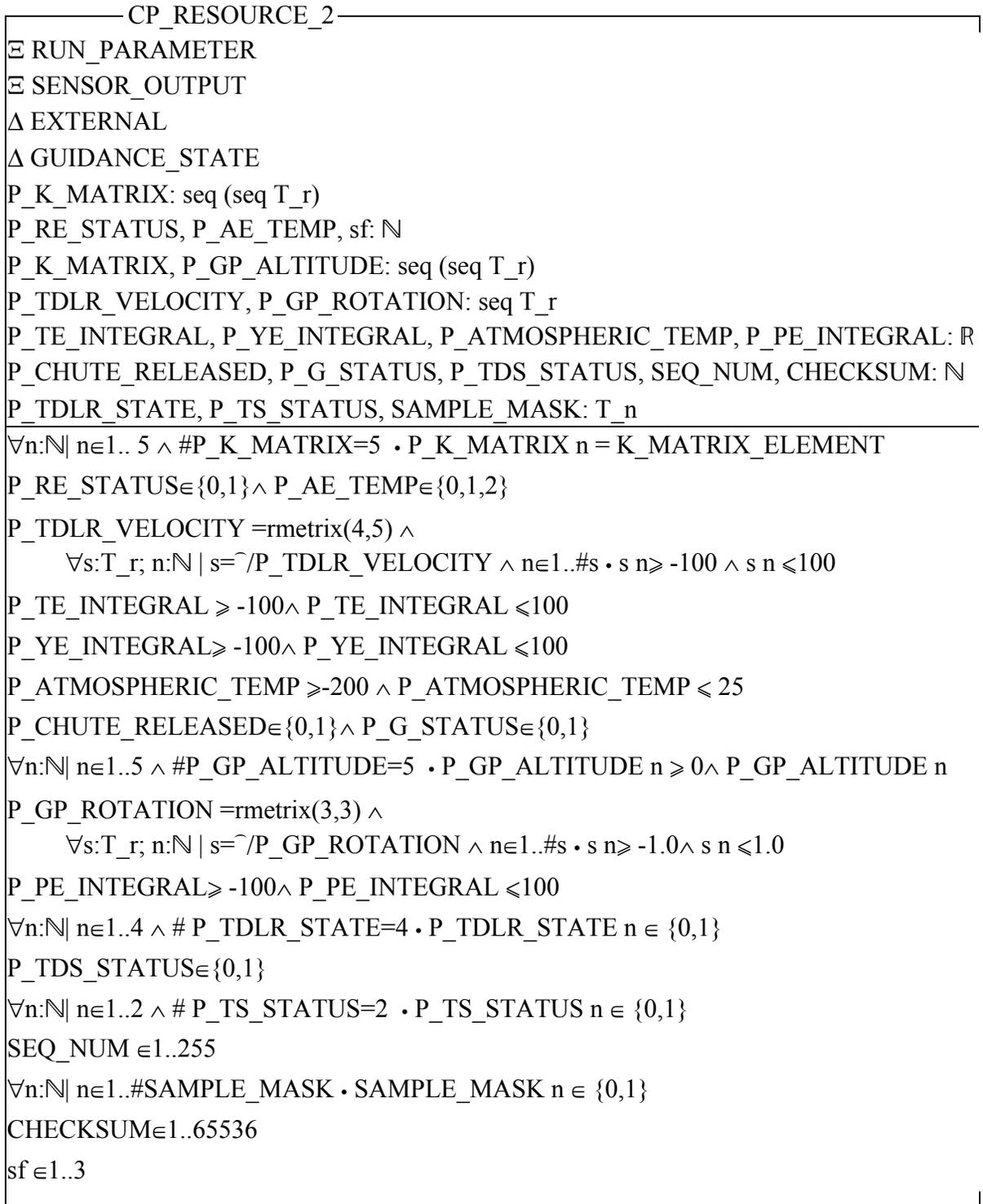


Figure 31. CP_RESOURCE_2 schema

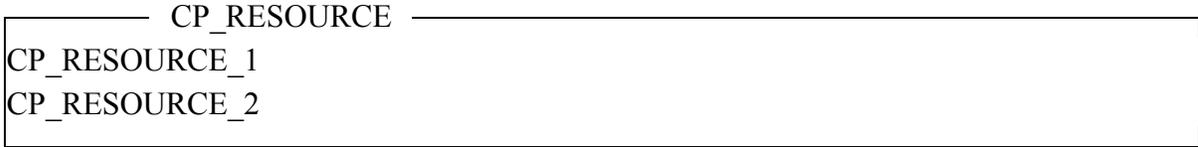


Figure 32. CP_RESOURCE schema

The CP_PREP_MASK1 (Figure 33), CP_PREP_MASK2, CP_PREP_MASK3, and CP_PREP_MASK4 shows the definitions of the functions for building a sample mask in CP_MASK schema (Figure 34). The sample mask (SAMPLE_MASK) is a Boolean vector where 1's represent the variables which have been modified since the last transmission.

In Figure 33, the predicate ❶ describes how an element of the sample mask is set. AE_CMD variable is a finite sequence of 3 natural numbers. When “P_AE_CMD I” matches with “AE_CMD I”, the Prep_Mask1 function places 1 into the index 1 of the return sequence.

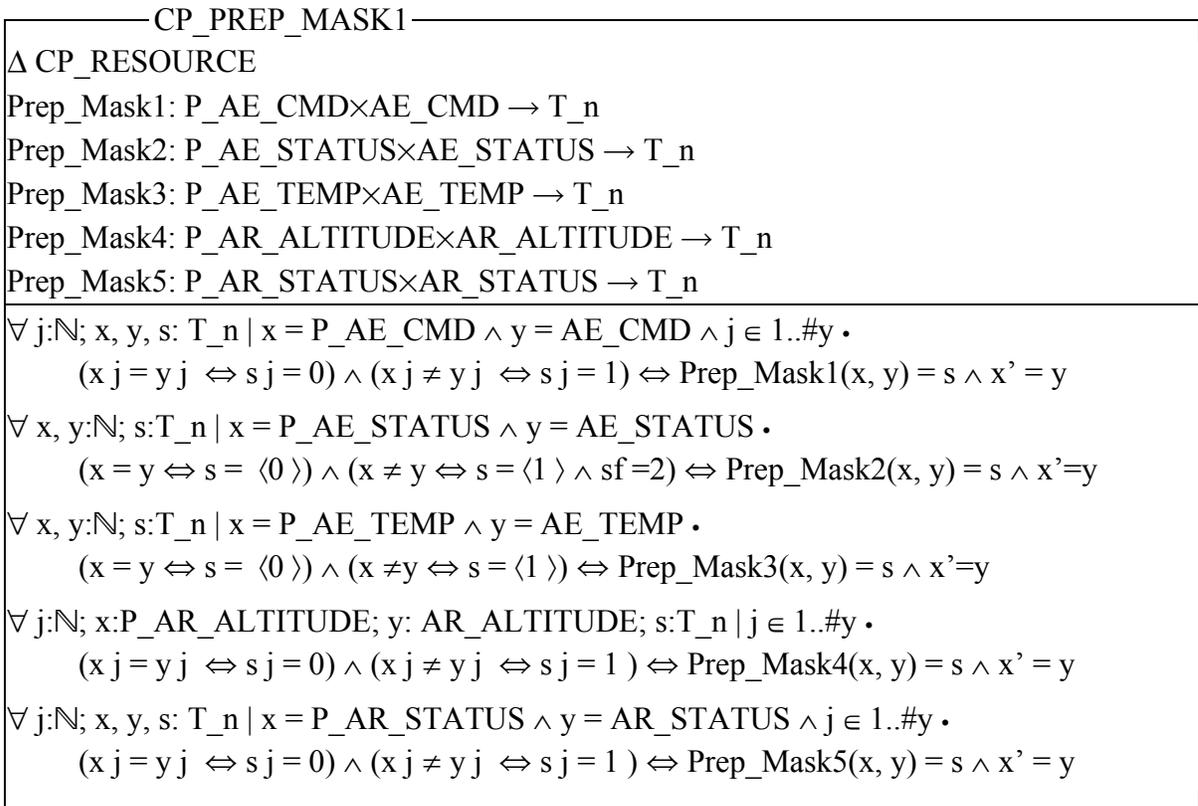


Figure 33. CP_PREP_MASK1 schema

CP_PREP_MASK2

Δ CP_RESOURCE

Prep_Mask6: P_ATMOSPHERIC_TEMP \times ATMOSPHERIC_TEMP \rightarrow T_n

Prep_Mask7: P_A_ACCELERATION \times A_ACCELERATION \rightarrow T_n

Prep_Mask8: P_A_STATUS \times A_STATUS \rightarrow T_n

Prep_Mask9: P_CHUTE_RELEASED \times CHUTE_RELEASED \rightarrow T_n

Prep_Mask10: P_CONTOUR_CROSSED \times CONTOUR_CROSSED \rightarrow T_n

Prep_Mask11: P_C_STATUS \times C_STATUS \rightarrow T_n

Prep_Mask12: P_GP_ALTITUDE \times GP_ALTITUDE \rightarrow T_n

Prep_Mask13: P_GP_ATTITUDE \times GP_ATTITUDE \rightarrow T_n

Prep_Mask14: P_GP_PHASE \times GP_PHASE \rightarrow T_n

Prep_Mask15: P_GP_ROTATION \times GP_ROTATION \rightarrow T_n

$\forall x, y: \mathbb{R}; s: T_n \mid x = P_ATMOSPHERIC_TEMP \wedge y = ATMOSPHERIC_TEMP \cdot$

$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow Prep_Mask6(x, y) = s \wedge x' = y$

$\forall j: \mathbb{N}; x, y: T_r; s: T_n \mid x = \neg/P_A_ACCELERATION \wedge y = \neg/A_ACCELERATION \wedge$

$j \in 1.. \#y \cdot (x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow Prep_Mask7(x, y) = s \wedge x' = y$

$\forall j: \mathbb{N}; x, y, s: T_n \mid x = \neg/P_A_STATUS \wedge y = \neg/A_STATUS \wedge j \in 1.. \#y \cdot$

$(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1 \wedge sf = 1) \Leftrightarrow Prep_Mask8(x, y) = s \wedge x' = y$

$\forall x, y: \mathbb{N}; s: T_n \mid x = P_CHUTE_RELEASED \wedge y = CHUTE_RELEASED \cdot$

$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow Prep_Mask9(x, y) = s \wedge x' = y$

$\forall x, y: \mathbb{N}; s: T_n \mid x = P_CONTOUR_CROSSED \wedge y = CONTOUR_CROSSED \cdot$

$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow Prep_Mask10(x, y) = s \wedge x' = y$

$\forall x, y: \mathbb{N}; s: T_n \mid x = P_C_STATUS \wedge y = C_STATUS \cdot$

$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow Prep_Mask11(x, y) = s \wedge x' = y$

$\forall j: \mathbb{N}; x: P_GP_ALTITUDE; y: GP_ALTITUDE; s: T_n \mid j \in 1.. \#y \cdot$

$(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow Prep_Mask12(x, y) = s \wedge x' = y$

$\forall j: \mathbb{N}; x, y: T_r; s: T_n \mid x = \neg/(\neg/P_GP_ATTITUDE) \wedge y = \neg/(\neg/GP_ATTITUDE) \wedge$

$j \in 1.. \#y \cdot$

$(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow Prep_Mask13(x, y) = s \wedge x' = y$

$\forall x, y: \mathbb{N}; s: T_n \mid x = P_GP_PHASE \wedge y = GP_PHASE \cdot$

$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle \wedge sf = 3) \Leftrightarrow Prep_Mask14(x, y) = s \wedge x' = y$

$\forall j: \mathbb{N}; x, y: T_r; s: T_n \mid x = \neg/P_GP_ROTATION \wedge y = \neg/GP_ROTATION \wedge j \in 1.. \#y \cdot$

$(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow Prep_Mask15(x, y) = s \wedge x' = y$

CP_PREP_MASK3

Δ CP_RESOURCE

Prep_Mask16: P_GP_VELOCITY \times GP_VELOCITY \rightarrow T_n

Prep_Mask17: P_G_ROTATION \times G_ROTATION \rightarrow T_n

Prep_Mask18: P_G_STATUS \times G_STATUS \rightarrow T_n

Prep_Mask19: P_K_ALT \times K_ALT \rightarrow T_n

Prep_Mask20: P_K_MATRIX \times K_MATRIX \rightarrow T_n

Prep_Mask21: P_PE_INTEGRAL \times PE_INTEGRAL \rightarrow T_n

Prep_Mask22: P_RE_CMD \times RE_CMD \rightarrow T_n

Prep_Mask23: P_RE_STATUS \times RE_STATUS \rightarrow T_n

Prep_Mask24: P_TDLR_STATE \times TDLR_STATE \rightarrow T_n

Prep_Mask25: P_TDLR_STATUS \times TDLR_STATUS \rightarrow T_n

Prep_Mask26: P_TDLR_VELOCITY \times TDLR_VELOCITY \rightarrow T_n

$\forall j:\mathbb{N}; x, y:T_r; s:T_n \mid x = \neg / P_GP_VELOCITY \wedge y = \neg / GP_VELOCITY \wedge j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask16}(x, y) = s \wedge x' = y$

$\forall j:\mathbb{N}; x, y:T_r; s:T_n \mid x = \neg / P_G_ROTATION \wedge y = \neg / G_ROTATION \wedge j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask17}(x, y) = s \wedge x' = y$

$\forall x, y:\mathbb{N}; s:T_n \mid x = P_G_STATUS \wedge y = G_STATUS \cdot$
 $(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask18}(x, y) = s \wedge x' = y$

$\forall j:\mathbb{N}; x:P_K_ALT; y:K_ALT; s:T_n \mid j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask19}(x, y) = s \wedge x' = y$

$\forall j:\mathbb{N}; x, y:T_r; s:T_n \mid x = \neg / (/ P_K_MATRIX) \wedge y = \neg / (/ K_MATRIX) \wedge j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask20}(x, y) = s \wedge x' = y$

$\forall x, y:\mathbb{R}; s:T_n \mid x = P_PE_INTEGRAL \wedge y = PE_INTEGRAL \cdot$
 $(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask21}(x, y) = s \wedge x' = y$

$\forall x, y:\mathbb{N}; s:T_n \mid x = P_RE_CMD \wedge y = RE_CMD \cdot$
 $(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask22}(x, y) = s \wedge x' = y$

$\forall x, y:\mathbb{N}; s:T_n \mid x = P_RE_STATUS \wedge y = RE_STATUS \cdot$
 $(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask23}(x, y) = s \wedge x' = y$

$\forall j:\mathbb{N}; x, y, s:T_n \mid x = P_TDLR_STATE \wedge y = TDLR_STATE \wedge j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask24}(x, y) = s \wedge x' = y$

$\forall j:\mathbb{N}; x:P_TDLR_STATUS; y:TDLR_STATUS; s:T_n \mid j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask25}(x, y) = s \wedge x' = y$

$\forall j:\mathbb{N}; x, y:T_r; s:T_n \mid x = \neg / P_TDLR_VELOCITY \wedge y = \neg / TDLR_VELOCITY \wedge$
 $j \in 1..#y \cdot$
 $(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask26}(x, y) = s \wedge x' = y$

CP_PREP_MASK4

Prep_Mask27: $P_TDS_STATUS \times TDS_STATUS \rightarrow T_n$

Prep_Mask28: $P_TD_SENSED \times TD_SENSED \rightarrow T_n$

Prep_Mask29: $P_TE_INTEGRAL \times TE_INTEGRAL \rightarrow T_n$

Prep_Mask30: $P_TS_STATUS \times TS_STATUS \rightarrow T_n$

Prep_Mask31: $P_VELOCITY_ERROR \times VELOCITY_ERROR \rightarrow T_n$

Prep_Mask32: $P_YE_INTEGRAL \times YE_INTEGRAL \rightarrow T_n$

$\forall x, y: \mathbb{N}; s: T_n \mid x = P_TDS_STATUS \wedge y = TDS_STATUS \cdot$

$$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask27}(x, y) = s \wedge x' = y$$

$\forall x, y: \mathbb{N}; s: T_n \mid x = P_TD_SENSED \wedge y = TD_SENSED \cdot$

$$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask28}(x, y) = s \wedge x' = y$$

$\forall x, y: \mathbb{R}; s: T_n \mid x = P_TE_INTEGRAL \wedge y = TE_INTEGRAL \cdot$

$$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask29}(x, y) = s \wedge x' = y$$

$\forall j: \mathbb{N}; x, y, s: T_n \mid x = P_TS_STATUS \wedge y = TS_STATUS \wedge j \in 1.. \#y \cdot$

$$(x_j = y_j \Leftrightarrow s_j = 0) \wedge (x_j \neq y_j \Leftrightarrow s_j = 1) \Leftrightarrow \text{Prep_Mask30}(x, y) = s \wedge x' = y$$

$\forall x, y: \mathbb{R}; s: T_n \mid x = P_VELOCITY_ERROR \wedge y = VELOCITY_ERROR \cdot$

$$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask31}(x, y) = s \wedge x' = y$$

$\forall x, y: \mathbb{R}; s: T_n \mid x = P_YE_INTEGRAL \wedge y = YE_INTEGRAL \cdot$

$$(x = y \Leftrightarrow s = \langle 0 \rangle) \wedge (x \neq y \Leftrightarrow s = \langle 1 \rangle) \Leftrightarrow \text{Prep_Mask32}(x, y) = s \wedge x' = y$$

CP_MASK
CP_PREP_MASK1
CP_PREP_MASK2
CP_PREP_MASK3
CP_PREP_MASK4
SAMPLE_MASK= Prep_Mask1(P_AE_CMD, AE_CMD) ^
Prep_Mask2(P_AE_STATUS, AE_STATUS) ^
Prep_Mask3(P_AE_TEMP, AE_TEMP) ^
Prep_Mask4(P_AR_ALTITUDE, AR_ALTITUDE) ^
Prep_Mask5(P_AR_STATUS, AR_STATUS) ^
Prep_Mask6(P_ATMOSPHERIC_TEMP, ATMOSPHERIC_TEMP) ^
Prep_Mask7(P_A_ACCELERATION, A_ACCELERATION) ^
Prep_Mask8(P_A_STATUS, A_STATUS) ^
Prep_Mask9(P_CHUTE_RELEASED, CHUTE_RELEASED) ^
Prep_Mask10(P_CONTOUR_CROSSED, CONTOUR_CROSSED) ^
Prep_Mask11(P_C_STATUS, C_STATUS) ^
Prep_Mask12(P_GP_ALTITUDE, GP_ALTITUDE) ^
Prep_Mask13(P_GP_ATTITUDE, GP_ATTITUDE) ^
Prep_Mask14(P_GP_PHASE, GP_PHASE) ^
Prep_Mask15(P_GP_ROTATION, GP_ROTATION) ^
Prep_Mask16(P_GP_VELOCITY, GP_VELOCITY) ^
Prep_Mask17(P_G_ROTATION, G_ROTATION) ^
Prep_Mask18(P_G_STATUS, G_STATUS) ^
Prep_Mask19(P_K_ALT, K_ALT) ^ Prep_Mask20(P_K_MATRIX, K_MATRIX) ^
Prep_Mask21(P_PE_INTEGRAL, PE_INTEGRAL) ^
Prep_Mask22(P_RE_CMD, RE_CMD) ^
Prep_Mask23(P_RE_STATUS, RE_STATUS) ^
Prep_Mask24(P_TDLR_STATE, TDLR_STATE) ^
Prep_Mask25(P_TDLR_STATUS, TDLR_STATUS) ^
Prep_Mask26(P_TDRL_VELOCITY, TDRL_VELOCITY) ^
Prep_Mask27(P_TDS_STATUS, TDS_STATUS) ^
Prep_Mask28(P_TD_SENSED, TD_SENSED) ^
Prep_Mask29(P_TE_INTEGRAL, TE_INTEGRAL) ^
Prep_Mask30(P_TS_STATUS, TS_STATUS) ^
Prep_Mask31(P_VELOCITY_ERROR, VELOCITY_ERROR) ^
Prep_Mask32(P_YE_INTEGRAL, YE_INTEGRAL)

Figure 34. CP_MASK schema

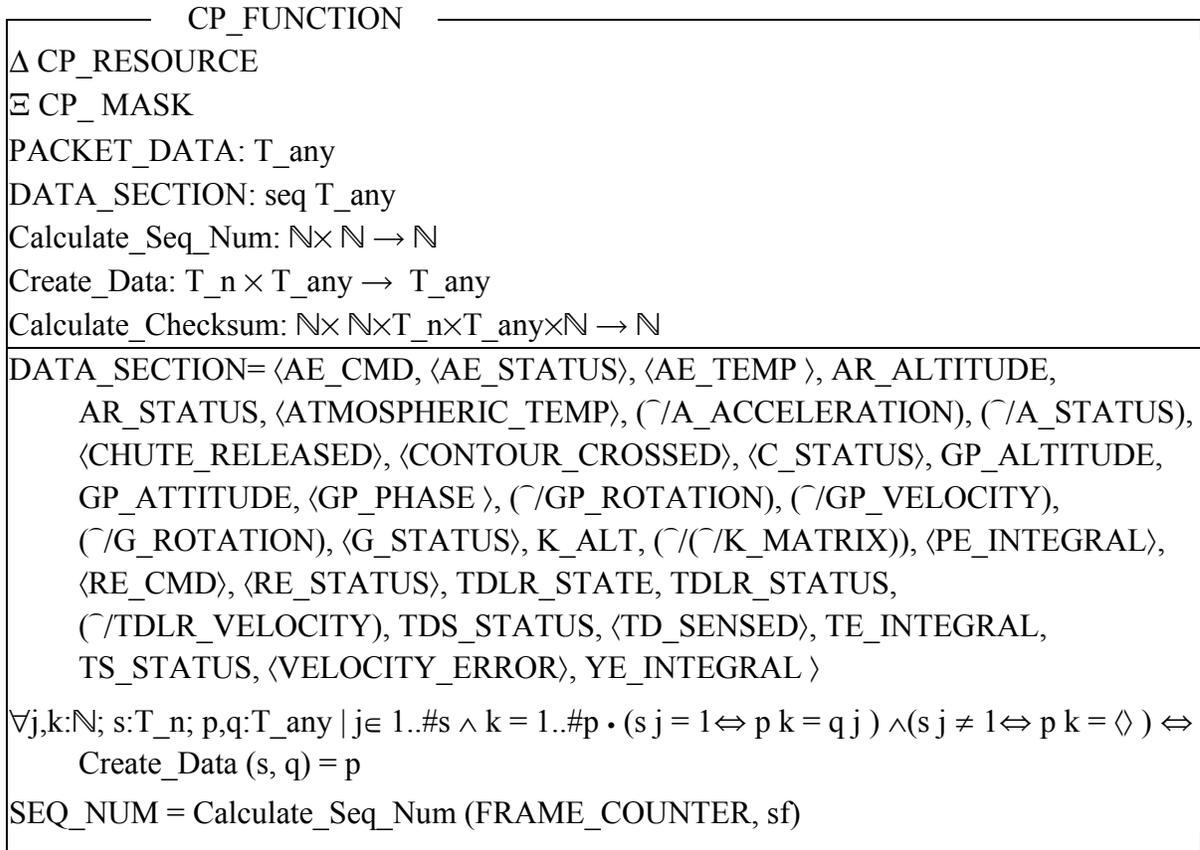


Figure 35. CP_FUCNTION schema

The CP_FUNCTION schema (Figure 35) defines the functions to be used for create data section and the checksum of a packet. Calauclate_Checksum is defined just with types because the operation was not defined in the specification. However, it was referenced to use CRC-16 which is very well known function. Create_Data function is defined to create the data section of the packet for transmission. It maps SAMPLE_MASK information with the variable data into a new T_any type sequence.

A sequence number of a packet is required to be different for each consecutive packet (represented in predicate **1** of Figure 36) and it is stated in the NL-based SRS as follows; “... the sequence number will be 0 during the first subframe of frame number 1. Sequence numbers repeat after the 255th packet and can be calculated based on the FRAME_COUNTER and the subframe where the present call to CP was made [25].” In the NL-based SRS, the subframe is not

defined as a variable. Therefore, a variable *sf* is defined locally in the CP submodule representing the subframe number (used in predicate ❶). *sf* marks the subframe by checking whether the variables that are modified uniquely in certain subframe. For example, the variable AR_STATUS (refer the predicate ❷ in Figure 33) is only modified at subframe2. Then *sf* marks 2 in its value. The exact equation for the sequence number calculation is not given. For the purpose of the development of the statecharts model the following equation is used; $((FRAME_COUNTER-1)*3 + sf) \bmod 256$.

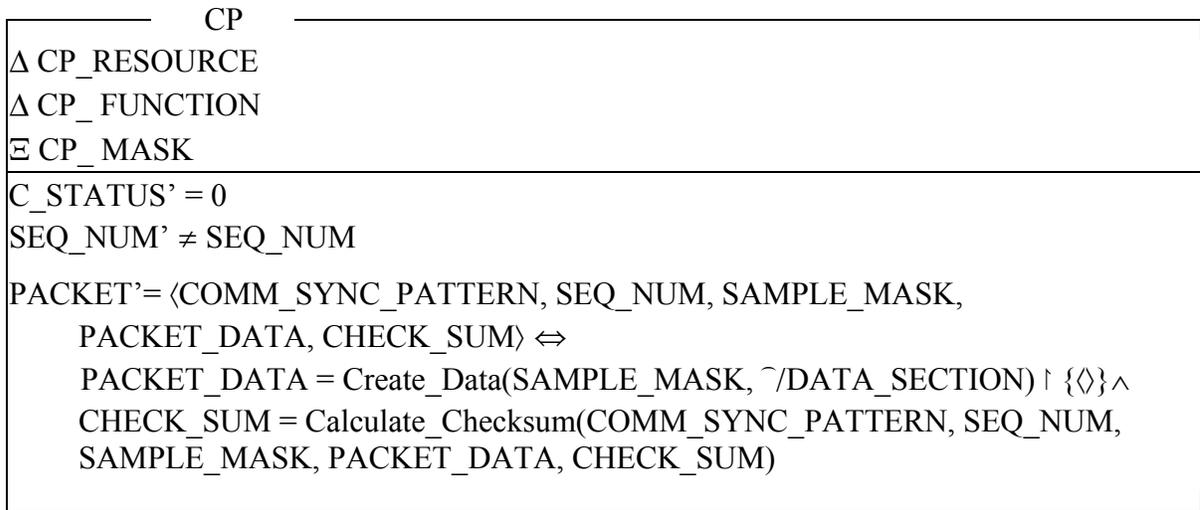


Figure 36. CP schema

The CP schema in Figure 36 describes the operation of the CP submodule. A packet is created in the CP with the communication synchronization pattern (COMM_SYNC_PATTERN), the sequence number (SEQ_NUM), the sample mask (SAMPLE_MASK), the data section (PACKET_DATA), and the checksum (CHECK_SUM) as described in the predicate ❷.

According to the NL-based SRS, some of the packet variables are real numbers. The size of real numbers is defined as 8 byte. Considering the size of a packet with real numbers and a byte-boundary limit (2 byte integer) specified in the NL-based SRS, the data section of a packet requires minimum of 642 bytes at the first subframe of frame number 1. However, the packet is

defined to have only 512 bytes (256 * 2 byte integer) in the data dictionary of the NL-based SRS. This means the definition and the function of the packet variable are inconsistent.

5.1.4 GP Module

The Guidance Processing (GP) submodule is defined in this section. The GP_RESOURCE schema shows that the GP submodule takes input from all four data stores and places its output into the GUIDANCE_STATE data store.

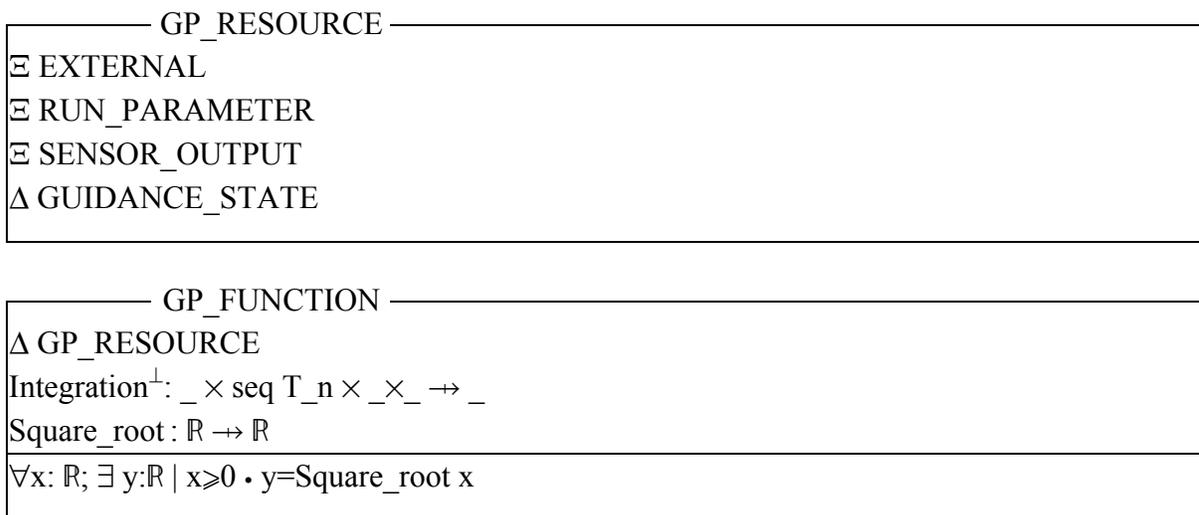


Figure 37. GP_FUCNTION schema

Using Z, one meets a difficulty specifying mathematical equations such as Integration function in Figure 37. It is declared as an “undefined” function marked by “ \perp ” notation. In this way, detailed implementation of integration function is remained for the system design phase. The square root computation is not being able to be defined in Z also. It is declared in the schema while the function implementation is remained for the system design.

The operation schema, GP_1, is a sub-schema of GP schema (shown in Figure 39). The GP submodule has multiple operations that are required to perform in order. Therefore, the separation of GP schema is based on the timely fashion. Operations in GP_1 take first. When the

operations in GP take turns, the post condition of GP_1 is considered as the initial condition of GP.

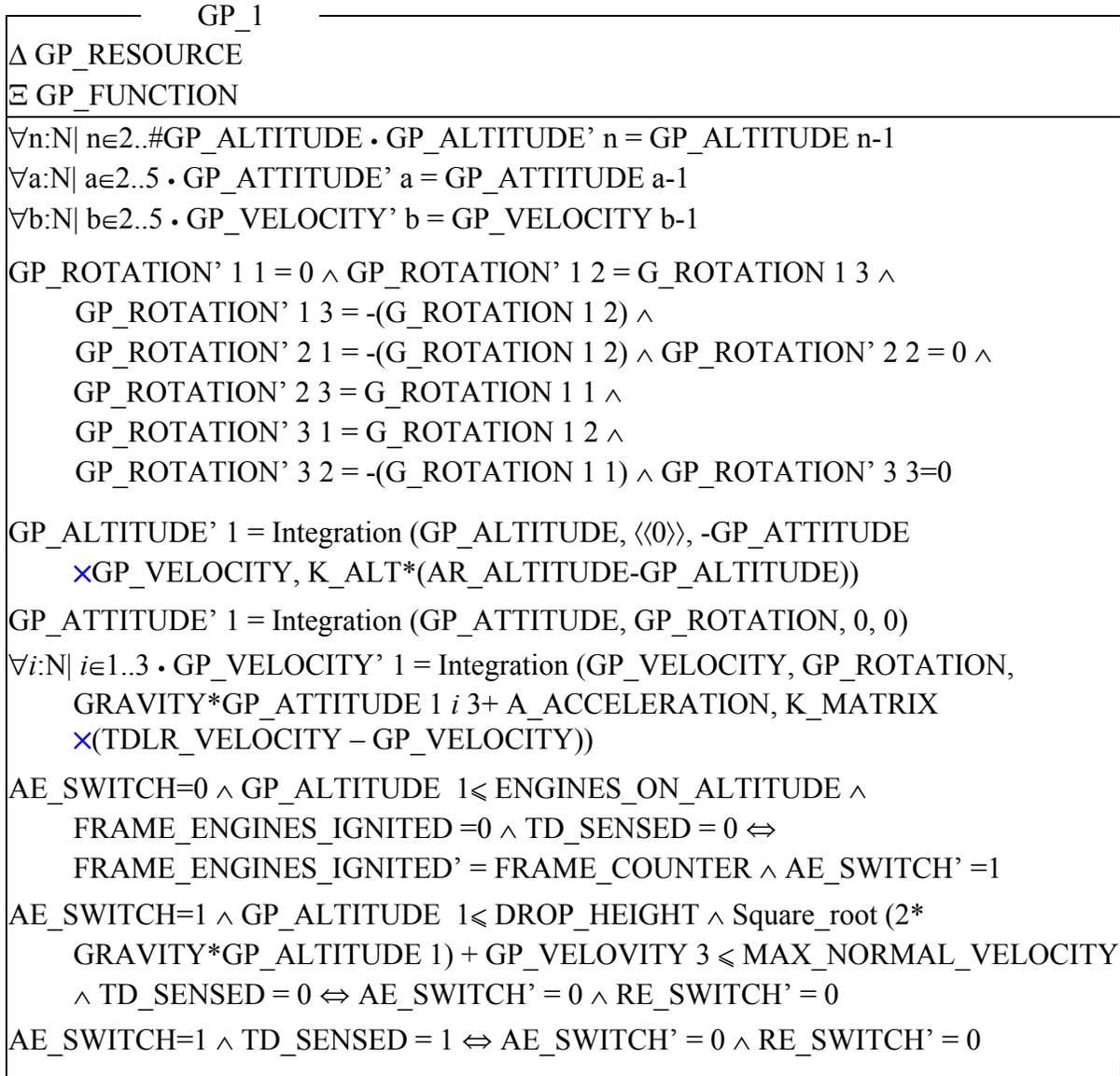


Figure 38. GP_1 schema

The predicate **4** in Figure 38 shows how to set the values for the matrix elements. The variable updates (represented by the predicates **5** and **7**) requires cross products of matrixes. It is marked by “ \times ”. However, it could be ambiguous for some readers to distinguish the symbol because Z uses the same symbol to represent a cross product of sets.

GP

ΔGP_1	
①	$\forall i:\mathbb{N} \mid i \in 1..100 \cdot \text{CONTOUR_ALTITUDE } i = \text{GP_ALTITUDE } 1 \Leftrightarrow$ $\text{VELOCITY_ERROR} = \text{GP_VELOCITY } 1 - \text{CONTOUR_VELOCITY } i$
①	$\forall i,j:\mathbb{N} \mid i \in 1..100 \wedge j \in 1..100 \wedge j > i \cdot \text{CONTOUR_ALTITUDE } i \leq \text{GP_ALTITUDE } 1 \wedge$ $\text{CONTOUR_ALTITUDE } j \geq \text{GP_ALTITUDE } 1 \Leftrightarrow \text{VELOCITY_ERROR} =$ $\text{GP_VELOCITY } 1 - (\text{CONTOUR_VELOCITY } i + (\text{GP_ALTITUDE } 1 -$ $\text{CONTOUR_ALTITUDE } i) * (\text{CONTOUR_VELOCITY } j - \text{CONTOUR_VELOCITY } i)$ $)/(\text{CONTOUR_ALTITUDE } j - \text{CONTOUR_ALTITUDE } i))$
②	$\text{CONTOUR_ALTITUDE } 1 \leq \text{GP_ALTITUDE } 1 \Leftrightarrow \text{VELOCITY_ERROR} =$ $\text{GP_VELOCITY } 1 - (\text{CONTOUR_VELOCITY } 1 - (\text{CONTOUR_ALTITUDE } 1 -$ $\text{GP_ALTITUDE } 1) * (\text{CONTOUR_VELOCITY } 1) / (\text{CONTOUR_ALTITUDE } 1))$
③	$\text{CONTOUR_ALTITUDE } 100 \geq \text{GP_ALTITUDE } 1 \Leftrightarrow \text{VELOCITY_ERROR} =$ $\text{GP_VELOCITY } 1 - (\text{CONTOUR_VELOCITY } 100 + (\text{GP_ALTITUDE } 1 -$ $\text{CONTOUR_ALTITUDE } 100) * (\text{CONTOUR_VELOCITY } 100) /$ $(\text{CONTOUR_ALTITUDE } 100))$
④	$\text{GP_ALTITUDE } 1 \leq \text{ENGINES_ON_ALTITUDE} \wedge \text{CONTOUR_CROSSED} = 0 \wedge$ $\text{VELOCITY_ERROR} > 0 \Leftrightarrow \text{CONTOUR_CROSSED}' = 1$
⑤	$\text{GP_PHASE}' = 1 \Leftrightarrow \text{GP_ALTITUDE } 1 > \text{ENGINES_ON_ALTITUDE}$
⑥	$\text{GP_PHASE} = 1 \wedge \text{GP_ALTITUDE } 1 \leq \text{ENGINES_ON_ALTITUDE} \wedge \text{AE_SWITCH} = 0 \wedge$ $\text{FRAME_ENGINES_IGNITED} = 0 \wedge \text{RE_SWITCH} = 0 \Leftrightarrow \text{GP_PHASE}' = 2$
⑦	$\text{GP_PHASE} = 2 \wedge \text{AE_TEMP} = 2 \wedge \text{CHUTE_RELEASED} = 1 \wedge \text{TD_SENSED} = 0 \Leftrightarrow$ $\text{GP_PHASE}' = 3$
⑧	$\text{GP_PHASE} = 2 \wedge \text{AE_TEMP} = 2 \wedge \text{CHUTE_RELEASED} = 1 \wedge \text{TD_SENSED} = 1 \Leftrightarrow$ $\text{GP_PHASE}' = 5$
⑨	$\text{GP_PHASE} = 3 \wedge \text{TD_SENSED} = 0 \wedge \text{GP_ALTITUDE } 1 \leq \text{DROP_HEIGHT} \wedge$ $\text{TDS_STATUS} = 0 \wedge \text{Square_root}(2 * \text{GRAVITY} * \text{GP_ALTITUDE } 1) +$ $\text{GP_VELOCITY } 1 \leq \text{MAX_NORMAL_VELOCITY} \Leftrightarrow \text{GP_PHASE}' = 4$
⑩	$\text{GP_PHASE} = 3 \wedge \text{TD_SENSED} = 0 \wedge \text{GP_ALTITUDE } 1 \leq \text{DROP_HEIGHT} \wedge$ $\text{TDS_STATUS} = 1 \Leftrightarrow \text{GP_PHASE}' = 5$
①	$\text{GP_PHASE} = 3 \wedge \text{TD_SENSED} = 1 \Leftrightarrow \text{GP_PHASE}' = 5 \wedge \text{RUN_DONE}' = 1$
②	$\text{GP_PHASE} = 4 \wedge \text{TD_SENSED} = 1 \Leftrightarrow \text{GP_PHASE}' = 5 \wedge \text{RUN_DONE}' = 1$
③	$\text{GP_PHASE} = 4 \wedge \text{TDS_STATUS} = 1 \Leftrightarrow \text{GP_PHASE}' = 5$
④	$\text{CL} = 1 \wedge \text{GP_VELOCITY } 1 < \text{DROP_SPEED} \wedge \exists x:\mathbb{N} \mid x \in 1..100 \cdot$ $\text{CONTOUR_VELOCITY } x..100 = \text{DROP_SPEED} \Leftrightarrow \text{CL}' = 2 \wedge \text{TE_INTEGRAL}' = 0.0$

Figure 39. GP schema

The GP submodule is required to do the following operations in order;

Operations	Predicates in GP_1
1. Rotate GP_ALTITUDE, GP_ATTITUDE, and GP_VELOCITY.	①, ②, ③
2. Setup the GP_ROTATION matrix	④
3. Calculate new values of GP_ALTITUDE, GP_ATTITUDE, and GP_VELOCITY with given equations in NL-based SRS.	⑤, ⑥, ⑦
4. Determine Axial and Roll engine should be on or off	⑧, ⑨, ⑩

Operations	Predicates in GP
5. Determine velocity error	①, ①, ②, ③
6. Determine if contour has been crossed	④
7. Determine guidance phase	⑤, ⑥, ⑦, ⑧, ⑨, ⑩, ①, ②, ③
8. Determine which set of control law parameter to use	④

The predicates listed next to the operations carry the operations in the respected schema.

5.1.5 RECLP Module

The Roll Engine Control Law Processing (RECLP) submodule is represented in this chapter. The RECLP submodule gets inputs from all the data stores and puts its operation results into RUN_PARAMETER, GUIDANCE_STATE, and EXTERNAL data store.

RECLP_FUNCTION
Δ RUN_PARAMETER
Ξ SENSOR_OUTPUT
Δ GUIDANCE_STATE
Δ EXTERNAL
* $THETA' = THETA + (DELTA_T * G_ROTATION \ 1 \ 1)$

In RECLP_FUNCTION schema, the predicate describes the integration method that is specified to be used to calculate THETA variable. The whole specification for the RECLP submodule was very ambiguous. The DELTA_T and G_ROTATION variable was listed as input

for this submodule with no description of usage. A variable “**p**” was presented on the graph that describes derivation values of THETA without specific description of the variable. Euler’s method integration is noted to be used for THETA while appropriate inputs are already provided but without specified instructions (G_ROTATION 1 1 is a slope, the **p** value, of the actual function and DELTA_T is a time slot size). It would be better to provide specific equations rather than to give information and make the reader figure out what the specification really means.

RECLP	
Δ RECLP_FUNCTION	
①	$\text{RE_SWITCH}=0 \Leftrightarrow \text{RE_CMD}'=1 \wedge \text{RE_STATUS}'=0$ $(\text{G_ROTATION } 1 \ 1 > \text{P3} \wedge \text{G_ROTATION } 1 \ 1 < \text{P4} \wedge \text{THETA} < 0) \vee$ $(\text{G_ROTATION } 1 \ 1 < -\text{P3} \wedge \text{G_ROTATION } 1 \ 1 > -\text{P4} \wedge \text{THETA} > 0) \vee$ $(\text{G_ROTATION } 1 \ 1 > 0 \wedge \text{G_ROTATION } 1 \ 1 < \text{P3} \wedge \text{THETA} < 0 \wedge$ $\text{THETA} > -\text{THETA2}) \vee (\text{G_ROTATION } 1 \ 1 < 0 \wedge \text{G_ROTATION } 1 \ 1 > -\text{P3} \wedge$
②	$\text{THETA} > 0 \wedge \text{THETA} < \text{THETA2}) \vee (\text{G_ROTATION } 1 \ 1 > 0 \wedge$ $\text{G_ROTATION } 1 \ 1 < \text{P1} \wedge \text{THETA} > 0 \wedge \text{THETA} < \text{THETA1}) \vee$ $(\text{G_ROTATION } 1 \ 1 < 0 \wedge \text{G_ROTATION } 1 \ 1 > -\text{P1} \wedge \text{THETA} < 0 \wedge$ $\text{THETA} > -\text{THETA1}) \Leftrightarrow \text{RE_CMD}'=1 \wedge \text{RE_STATUS}'=0$
③	$\text{G_ROTATION } 1 \ 1 > -\text{P1} \wedge \text{G_ROTATION } 1 \ 1 < 0 \wedge \text{THETA} > -\text{THETA2} \wedge$ $\text{THETA} < -\text{THETA1} \Leftrightarrow \text{RE_CMD}'=2 \wedge \text{RE_STATUS}'=0$
④	$\text{G_ROTATION } 1 \ 1 < \text{P1} \wedge \text{G_ROTATION } 1 \ 1 > 0 \wedge \text{THETA} < \text{THETA2} \wedge \text{THETA} >$ $\text{THETA1} \Leftrightarrow \text{RE_CMD}'=3 \wedge \text{RE_STATUS}'=0$
⑤	$\text{G_ROTATION } 1 \ 1 > -\text{P2} \wedge \text{G_ROTATION } 1 \ 1 < -\text{P1} \wedge \text{THETA} > -\text{THETA2} \wedge \text{THETA} < 0$ $\Leftrightarrow \text{RE_CMD}'=4 \wedge \text{RE_STATUS}'=0$
⑥	$\text{G_ROTATION } 1 \ 1 < \text{P2} \wedge \text{G_ROTATION } 1 \ 1 > \text{P1} \wedge \text{THETA} < \text{THETA2} \wedge \text{THETA} > 0$ $\Leftrightarrow \text{RE_CMD}'=5 \wedge \text{RE_STATUS}'=0$
⑦	$\text{G_ROTATION } 1 \ 1 < -\text{P4} \vee (\text{G_ROTATION } 1 \ 1 < -\text{P2} \wedge \text{THETA} < 0) \vee$ $(\text{G_ROTATION } 1 \ 1 < \text{P3} \wedge \text{THETA} < -\text{THETA2})$ $\Leftrightarrow \text{RE_CMD}'=6 \wedge \text{RE_STATUS}'=0$
⑧	$\text{G_ROTATION } 1 \ 1 > \text{P4} \vee (\text{G_ROTATION } 1 \ 1 > \text{P2} \wedge \text{THETA} > 0) \vee$ $(\text{G_ROTATION } 1 \ 1 > -\text{P3} \wedge \text{THETA} > \text{THETA2})$ $\Leftrightarrow \text{RE_CMD}'=7 \wedge \text{RE_STATUS}'=0$

The RECLP schema describes the operations of the RECLP submodule. The RECLP required to set RE_STATUS variable and to determine if engines are on (the predicate ❶ in RECLP schema). The pulse intensity and directions, and roll engine commands are specified in the predicate ❷-❸. **Boundary values are ambiguous (p=p1, p=p2, etc).**

5.1.6 GCS Schema

The GCS schemas are the highest level schemas of the GCS excerpt. GCS_RESOURCE schema defines control variables and state variables of the GCS system. The INIT_GCS, RUN_GCS, and END_GCS! variables are mentioned in the high level specification part while actual definition of the variable is not specified in the data dictionary. START_SIGNAL? variable is self descriptive; however, it is necessary to provide complete information for each of variables used in the SRS. SUBFRAME1, SUBFRAME2, and SUBFRAME3 are defined as local variable for the GCS schema because they are not defined as state variable in the NL-based GCS data dictionary. Those subframes are referred frequently and even used to calculate some data in some submodules in the NL-based SRS, yet they were not specified as variables. This reveals the fact that the SRS is incompletely specified.

The gcs_control function is defined to map two Boolean variables with 1 when they have the same value.

GCS_RESOURCE
INIT_GCS, RUN_GCS, END_GCS!: \mathbb{N}
START_SIGNAL?: \mathbb{N}
SUBFRAME1, SUBFRAME2, SUBFRAME3 : \mathbb{N}
gcs_control: $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
$INIT_GCS \in \{0,1\} \wedge RUN_GCS \in \{0,1\} \wedge END_GCS! \in \{0,1\}$
$START_SIGNAL? \in \{0,1\} \wedge INIT_DONE \in \{0,1\} \wedge RUN_DONE \in \{0,1\}$
$SUBFRAME1 \in \{0,1\} \wedge SUBFRAME2 \in \{0,1\} \wedge SUBFRAME3 \in \{0,1\}$
$\forall x,y:\mathbb{N} \mid x \in \{0,1\} \wedge y \in \{0,1\} \wedge x = y \cdot gcs_control \ x \ y = 1$

GCS_INIT
Δ GCS_RESOURCE
$INIT_GCS = 1 \wedge RUN_GCS = 0 \wedge END_GCS! = 0$
$INIT_DONE? = 1 \wedge RUN_DONE = 0$
$SUBFRAME1 = 0 \wedge SUBFRAME2 = 0 \wedge SUBFRAME3 = 0$

GCS_INIT schema represents the initialization of the GCS excerpt.

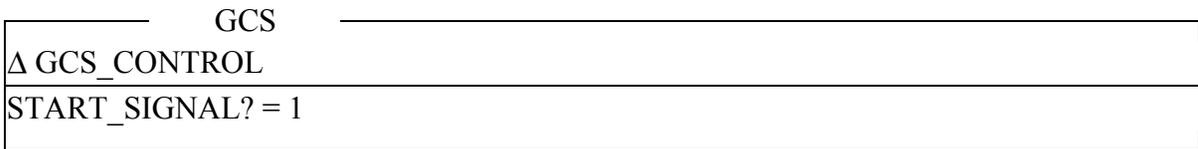
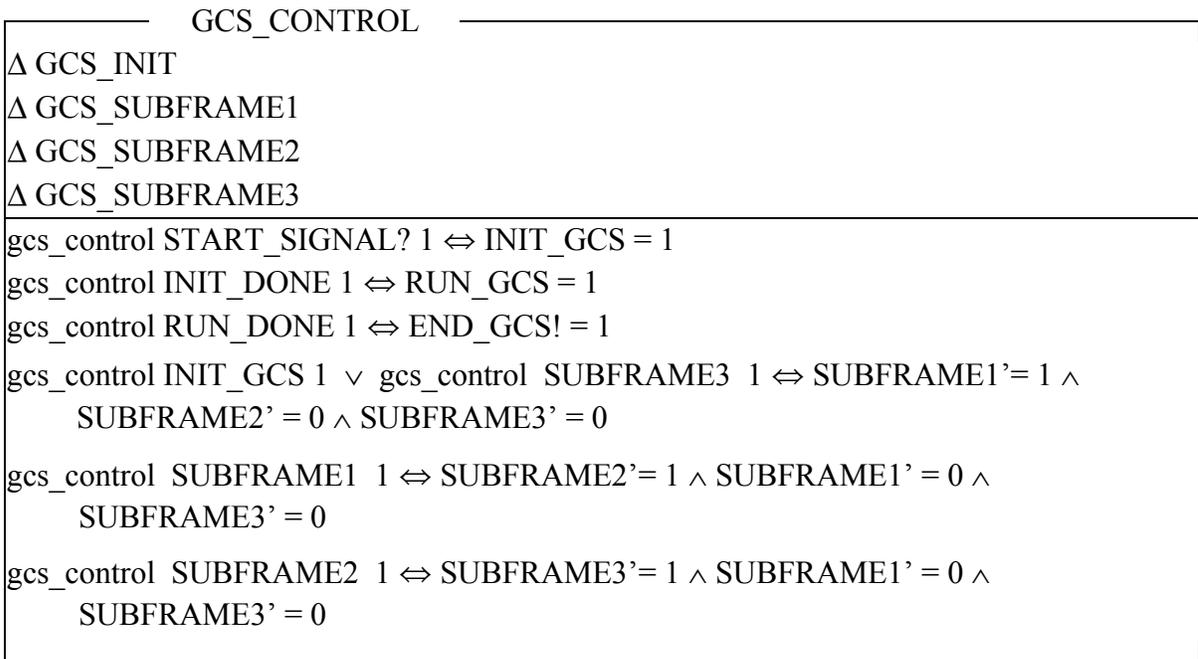
GCS_SUBFRAME1
Δ ARSP
Δ CP
SUBFRAME1=1

GCS_SUBFRAME2
Δ GP
Δ CP
SUBFRAME2=1

GCS_SUBFRAME3
Δ RECLP
Δ CP
SUBFRAME3=1

GCS_SUBFRAME1, GCS_SUBFRAME2, and GCS_SUBFRAME3 schemas are the state schemas showing the system running status based on the subframes. They represent the same

information of the Table 2. The GCS_CONTROL schema is the operation schema of the GCS. The GCS is initiated when the START_SIGNAL? input variable turns 1. After initialization, the GCS updates its global state variable INIT_DONE to 1 and running the GCS system starting from subframe1. The GCS system takes infinite loop between subframes until GP updates the global state variable RUN_DONE to 1. When the RUN_DONE set to 1, the GCS system state set to END_GCS and the output END_GCS! returns.



The GCS schema imports GCS_CONTROL schema to modify and starts the initialization process with START_SIGNAL?=1.

5.2. Executable Models

The Z specification shown in Chapter 5.1 is developed in the Statemate environments. First, a

module chart is developed to represent the structure of the GCS excerpt for reference. Then Activity/State charts are developed for specification testing and fault injection.

5.2.1 Module Chart

The module charts derived from the Figure 3 is shown in Figure 40. The module chart presented in the Figure 41 is the correct version of the module chart. The difference between two figures is because the NL-based SRS provides incomplete data transition directions with the Figure 3.

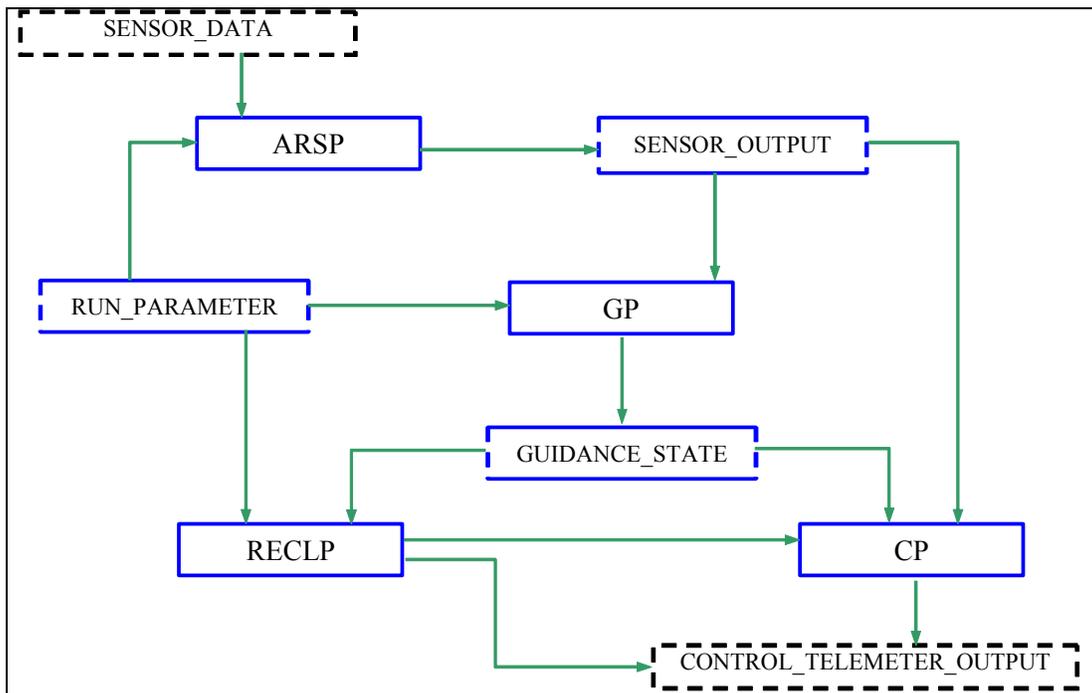


Figure 40. A module chart of the GCS excerpt

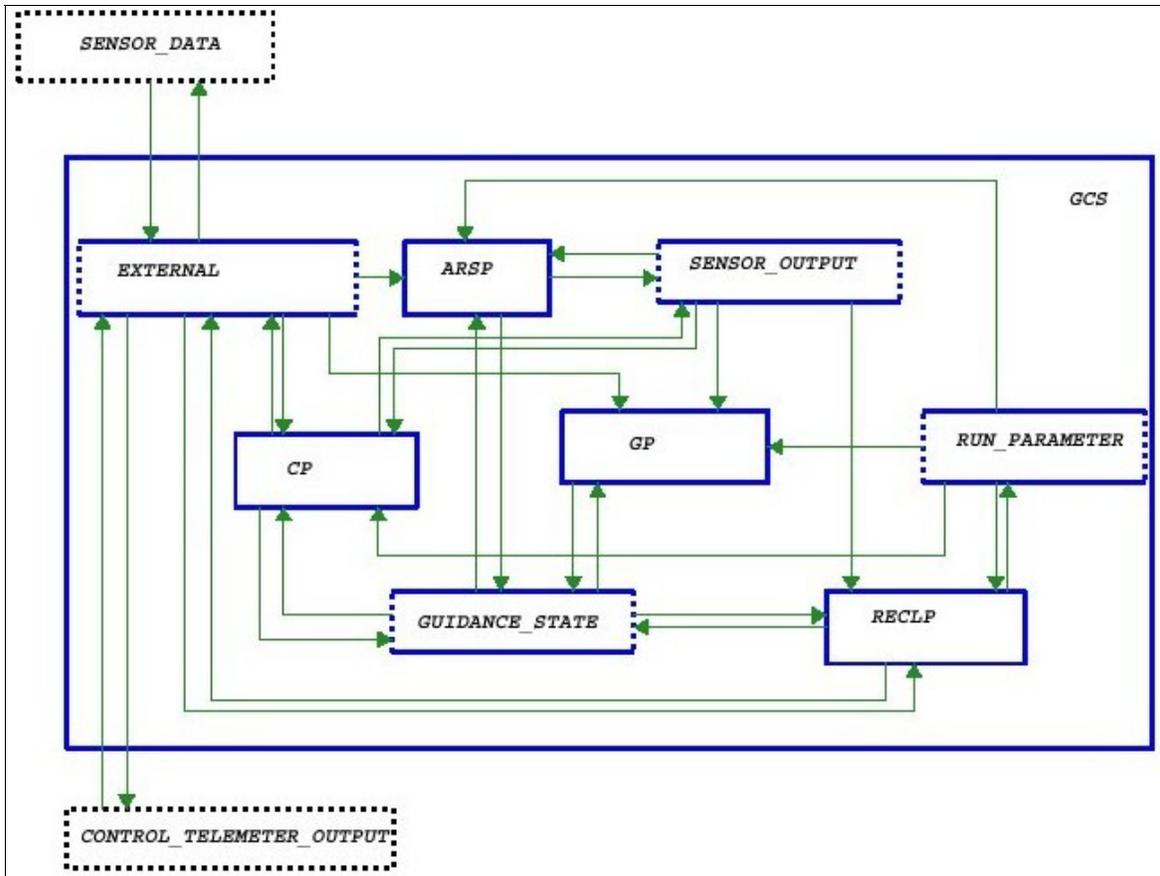


Figure 41. Actual module chart of the GCS excerpt

5.2.2 Activity Charts

In a GCS project created in the Statemate, the GCS activity chart is developed. Figure 42 shows the GCS activity chart with four data stores which contains the data definitions. The GCS activity is representing the GCS schemas in Chapter 5.1.6. The data stores contain the same variable definitions of Z schemas. The @GCS_CONTROL state represents a link with the GCS_CONTROL statechart. The @ARSP, @CP, @ GP, and @ RECLP activities are link to their own activity charts. Every activity requires to have only one control state.

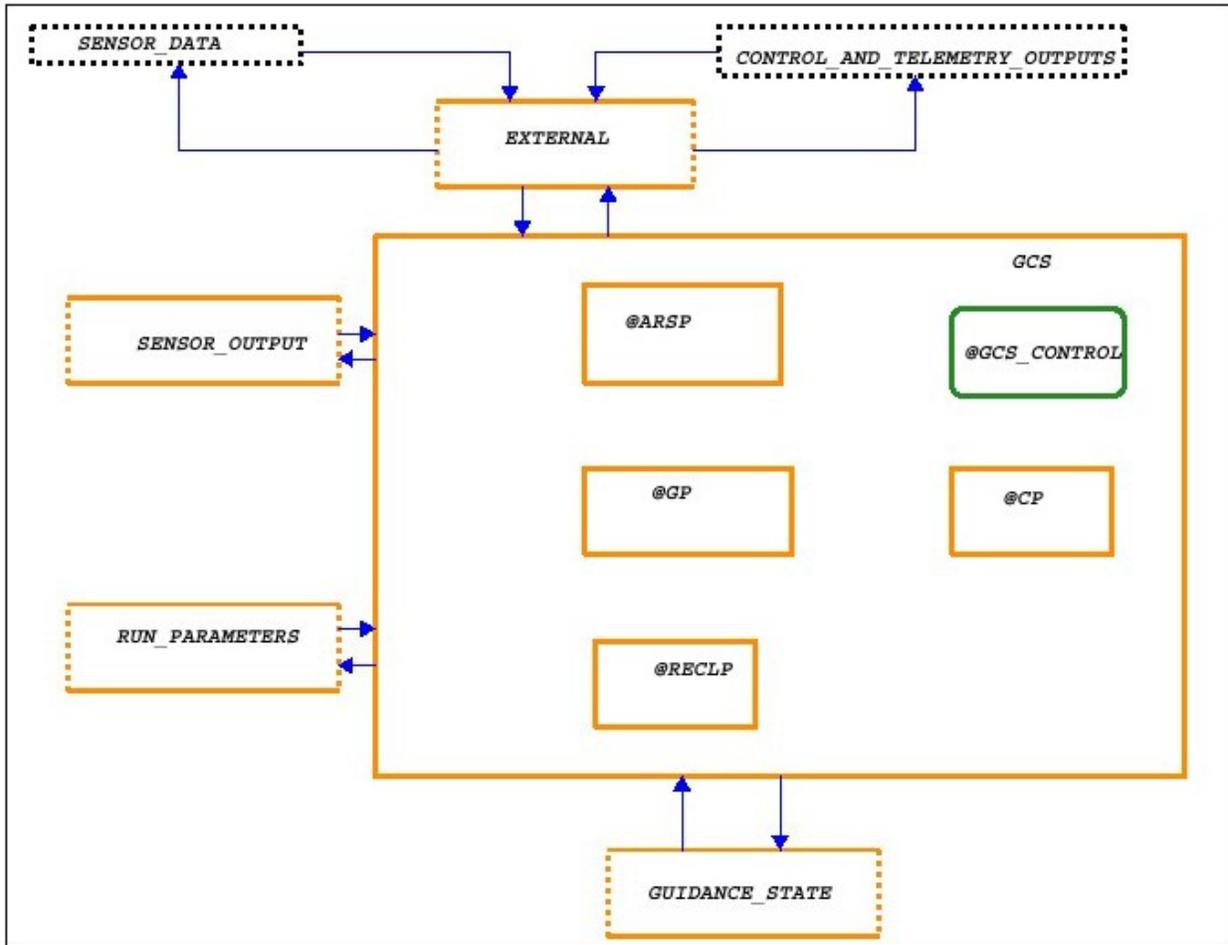


Figure 42. GCS activity chart

5.2.3 Statecharts

The GCS_CONTROL statecharts represents the GCS_CONTROL schemas. The default transition represent the moment START_SIGNAL? input for the GCS schema set to 1. The INITIALIZATION state is equivalent to the GCS_INIT schema. @SUBFRAME1, @SUBFRAME2, and @SUBFRAME3 states represent the local state variable defined in the GCS_RESOURCE schema. Every subframe has its own state charts linked to the superstate.

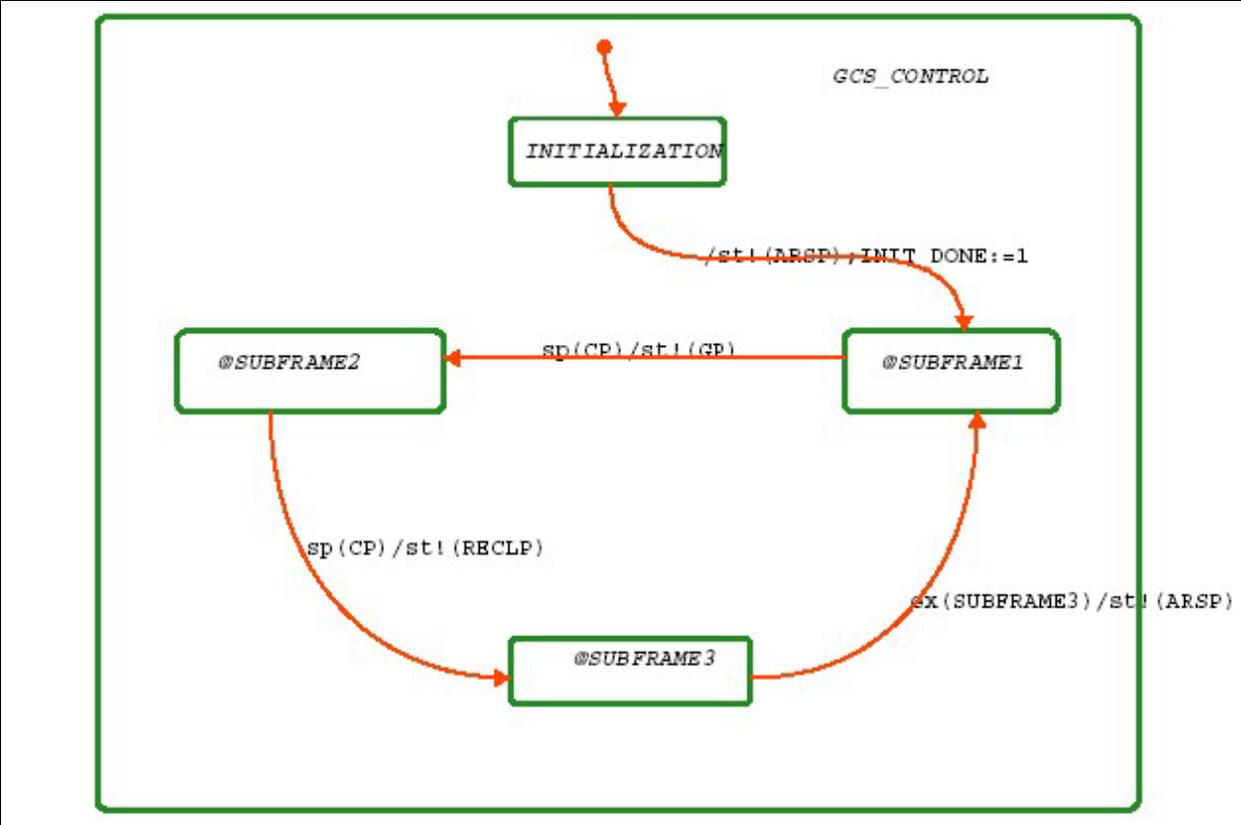


Figure 43. GCS_CONTROL statechart

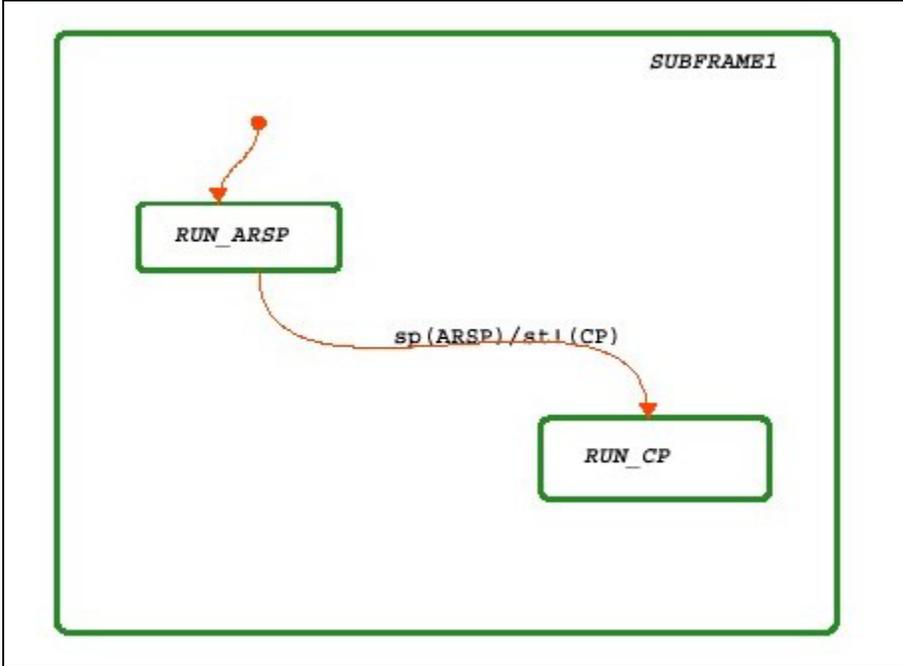


Figure 44. SUBFRAME1 statechart

The SUBFRAME1 statechart is representing the operation of the GCS_SUBFRAME1 schema. The SUBFRAME2 statechart shows the GCS excerpt operation at the subframe 2.

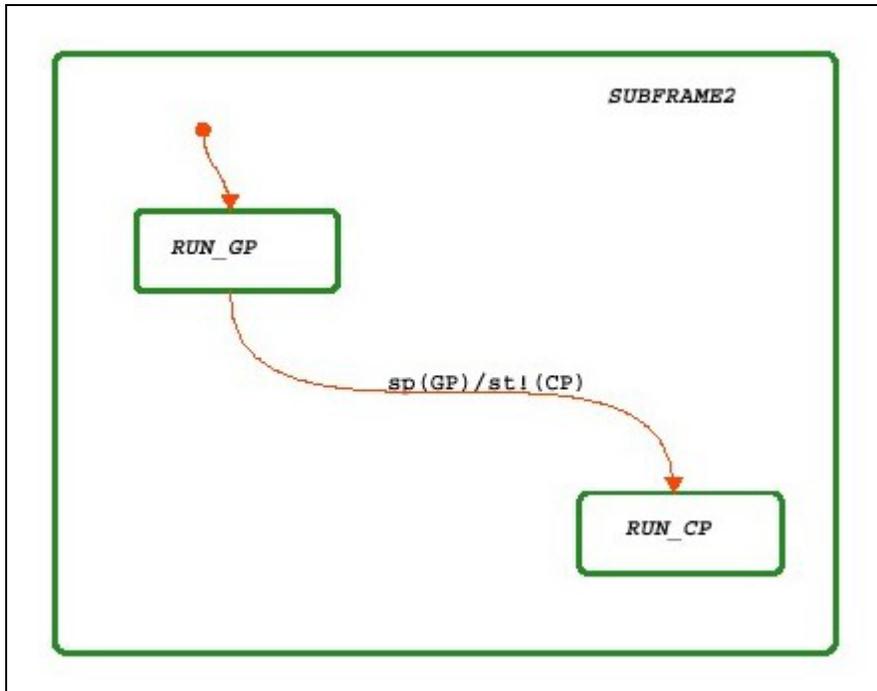


Figure 45. SUBFRAME2 statechart

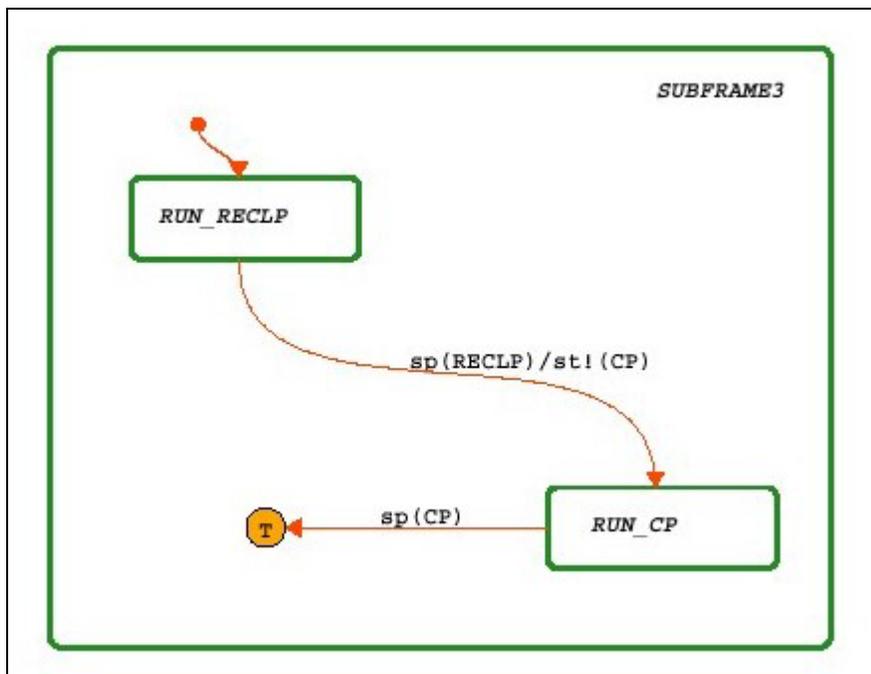


Figure 46. SUBFRAME3 statechart

The SUBFRAME3 statechart represents the GCS_SUBFRAME3 schema. However, it has a controlled termination because the GCS excerpt has many of shared data items with other undefined submodule. Without correct interaction, we cannot predict exact data item test results. Therefore, the simulation in this case study did not run over time.

5.3. Specification Test Results

In this section, the specification test results using simulations are presented. The test results from Finite State Machine Approach (FSMA) and Data Item Approach (DIA) are described in the following subsections.

5.3.1 Test Results

The GCS project has GCS activity charts and four sub-activity chart. Each activity chart has one control state that is linked to a control state chart. Most of the control statecharts are divided into several statecharts using superstates to reduce the complexity. In this section, the test results only high level GCS activity and state transition are presented. Detailed statecharts and activity charts are located in the appendices.

Table 8 shows the execution orders of the GCS excerpt high-level schemas. Those schemas are equivalent to the Z specification placed in the GCS schema chapter. The finite state machine approach test results show the statecharts model does not have absorbing states/activities. In other words, all the activities and states are reachable and there is no inconsistency in the model. These charts do not involved with any data item; therefore, the data item approach test is not presented here. The data item approach with the linked activities and states are placed in the appendix.

Table 8. GCS excerpt high-level activity/state charts simulation result

Name of Chart	Activity/State Name	Activity/State Transition order					
GCS	@GCS_CONTROL	En ₁	Ex ₃₃				
	@ARSP	En ₄	Ex ₇				
	@GP	En ₁₄	Ex ₁₇				
	@RECLP	En ₂₄	Ex ₂₇				
	@CP	En ₉	Ex ₁₂	En ₁₉	Ex ₂₂	En ₂₉	Ex ₃₁
GCS_CONTROL	INITIALIZATION	En ₂	Ex ₃				
	@SUBFRAME1	En ₅	Ex ₁₃				
	@SUBFRAME2	En ₁₅	Ex ₂₃				
	@SUBFRAME3	En ₂₅	Ex ₃₃				
SUBFRAME1	RUN_ARSP	En ₆	Ex ₈				
	RUN_CP	En ₁₀	Ex ₁₁				
SUBFRAME2	RUN_GP	En ₁₆	Ex ₁₈				
	RUN_CP	En ₂₀	Ex ₂₁				
SUBFRAME3	RUN_RECLP	En ₂₆	Ex ₂₈				
	RUN_CP	En ₃₀	Ex ₃₂				

En_{*i*}: entering the activity/state on *i*th order, Ex_{*i*}: exiting the activity/state on *i*th order.

5.3.2 Fault Injection Results/Discussion

First, one has to identify the module whether it can cause a system failure or not. For example, the CP is not taking any position for system failure of the GCS system function because it does not change any system variable values. Even when it fails, system can still run the operation and produces correct outputs. Therefore, the fault injection step is skipped for the CP submodule. This means that the submodules, which do not have ability to cause catastrophic system failures, are tested using simulations based on its operational profile only.

The GCS state/activity charts presented in the result section has only one path. Any failure of performing transition between activities and statecharts means system failure. It is because the scheduling of the sub-module processes. This requires changing the functional schedule to

tolerate failures that is caused by unnecessary sub-functionality (i.e., CP). By specifying CP as a concurrent process with the next submodule, the system failure caused by CP failure can be tolerated.

CHAPTER SEVEN

CONCLUSIONS

The result of this analysis revealed that it is possible to construct a complete and consistent specification using this method (Z-to-Statecharts). In the process, some ambiguous specifications are uncovered, which are associated with the reader's interpretation of the NL-based specification.

The outputs from the modules were examined and shown to be consistent with the expectations by running simulations based on the Statecharts/Activity-charts. All of the state activation/transition paths were in the correct order as expected for all test cases. Moreover, no nondeterministic state transitions were detected for all simulation runs (based on the conditions provided). In this context, the simulation has provided a means for determining the consistency of the requirements.

The output values from the simulation were checked and compared against the requirements, then found to be valid. There are several issues indicating that the SRS is incomplete some are found with the Z specification and others are found during various simulations including fault injection. In addition, some fault-prone states were identified where faults were injected into the GCS excerpt system model (i.e., Statecharts) while the models were executed using simulations. Though the GCS NL-based SRS did not specify fault tolerance, one can conclude that the system would not be able to tolerate certain system faults. Through the whole process of this case study, the SRS for the excerpt are found to be inconsistent, incomplete and not completely fault-tolerant. Therefore, the findings indicate that one can better understand the implications of the system requirements using this approach (Z-Statecharts) to facilitate their specification and

analysis. While developing Z specification, the task of following up variables for their data store and analyzing the functional specifications of submodules are time consuming. When the specification is written with the concerns of traceability, the effort and the amount of time spent for the specification analysis can be reduced for this approach. Consequently, this approach can help to avoid the problems that result when incorrectly specified artifacts (i.e., in this case requirements) force corrective work.

BIBLIOGRAPHY

1. Kotonya, G., and Sommerville, I., Requirements Engineering: Process and Techniques. 1998: Wiley.
2. Shaw, A.C., Real-Time Systems and Software. 2001: Wiley. 33-38.
3. Sommerville, I., Software Engineering. 6th ed. 2000, Reading, MA: Addison-Wesley. 742.
4. Mars Climate Orbiter Mishap Investigation Board Phase I Report. 1999.
5. Schwalbe, K., Information Technology Project Management. 2000: Course Technology.
6. Collard, R., Software Testing and Quality Assurance, working paper. 1997.
7. Fitch, D., Software Safety Engineering (S2E) Program Status. 2001.
8. Pradhan, D.K., Fault-Tolerant Computer System Design. 1996: Prentice Hall. 428-477.
9. Potter, B., Sinclair, J., and Till, D., An Introduction to Formal Specification and Z. Int'l. series in CS. 1996: Prentice Hall.
10. Leveson, N., Safeware - system safety and computers. 1995: Addison Wesley.
11. Czerny, B. Integrative Analysis of State-Based Requirements for Completeness and Consistency. PhD dissertation in Computer Science, Michigan State University. 1998.
12. Gaudel, M.-C., and Bernot, G., ed. The Role of Formal Specifications. IFIP State-of-the-Art Report: Algebraic Foundations of Systems Specification, ed. E. Astesiano, Kreowski, H.-J., and Krieg-Bruckner, B. 1999, Springer.
13. Vliet, H.V., Software Engineering: Principles and Practice. 2000: Wiley.
14. Sannella, D., and Tarlecki, A., ed. Algebraic Preliminaries. IFIP State-of-the-Art Reports: Algebraic Foundations of Systems Specification, ed. E. Astesiano, Kreowski, H.-J., and Krieg-Bruckner, B. 1999, Springer.

15. Fabbrini, F., Fusani, M., Gnesi, S., and Lami, G. An Automatic Quality Evaluation for Natural Language Requirements. Seventh International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ) 2001. Accessed from: www.ifi.uib.no/conf/refsq2001/papers/p3.pdf. Accessed on: Mar. 25, 2002.
16. Heitmeyer, C., Kirby, J. Jr., Labaw, B., Archer, M., and Bharadwaj, R., Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specification. *IEEE Transactions on Software engineering*, 1998. 24(11): p. 927-948.
17. Heimdahl, M.P.E., Leveson, Nancy G., Completeness and consistency in Hierarchical State-Based Requirements. *IEEE Trans on SE*, 1996. 22(N0.6, June 1996).
18. He, X., PZ nets - a formal method integrating Petri nets with Z. *Information and Software Technology*, 2001. 43: p. 1-18.
19. Hierons, R.M., Sadeghipour, S., Singh, H., Testing a system specified using Statecharts and Z. *Information and Software Technology*, 2001. 43(Feb).
20. Bussow, R., Weber, M., ed. A Steam-boiler Control Specification with Statecharts and Z. LNCS 1165. 1996.
21. Grieskamp, W., Heisel, M., and Dorr, H., ed. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. LNCS 1382. 1998.
22. Damm, W., Hungar, H., Kelb, P., and Schlor, R., ed. Statecharts - Using Graphical Specification Languages and Symbolic Model checking in the Verification of a Production Cell. LNCS 891. 1995. 131-149.
23. Bussow, R., Geisler, R., and Klar, M., ed. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. LNCS 1382. 1998.
24. Castello, R. From Informal Specification to Formalization: an Automated Visualization

- Approach. PhD dissertation in Computer Science, University of Texas at Dallas. 2000.
25. NASA, Software Requirements - Guidance and Control Software Development Specification Version 2.2 with the formal mods 1-8. 1993, NASA, Langley Research Center.
 26. Woodcock, J., and Davies, J., Using Z: Specification, Refinement, and Proof. Series of Computer Science. 1996: Prentice Hall International.
 27. Jacky, J., The Way of Z : practical programming with formal methods. 1997: Cambridge University Press.
 28. Harel, D., Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 1987. 8: p. 231-274.
 29. Harel, D., and Politi, M., Modeling Reactive Systems with Statecharts. 1998: McGraw-Hill.
 30. Bogdanov, K., and Holcombe, M., Statechart testing method for aircraft control systems. Software Testing, Verification & Reliability, 2001. 11(1): p. 39-54. J. Wiley.
 31. Voas, J., McGraw, G., Kassab, L., and Voas, L., A Crystal Ball for Software Liability. IEEE Computer, 1997. 30(6): p. 29-36.
 32. Sheldon, F., and Kim, H. Y. Validation of Guidance Control Software Requirements Specification for Reliability and Fault-Tolerance. Proceedings of Annual Reliability and Maintainability Symposium. 2002. Seattle, WA. USA: IEEE.
 33. Sheldon, F., Kim, H. Y., and Zhou, Z. A Case Study: Validation of the Guidance Control Software Requirements for Completeness, Consistency, and Fault Tolerance. Proceedings of IEEE 2001 Pacific Rim International Symposium on Dependable Computing. 2001. Seoul, Korea: IEEE Computer Society.

APPENDIX A

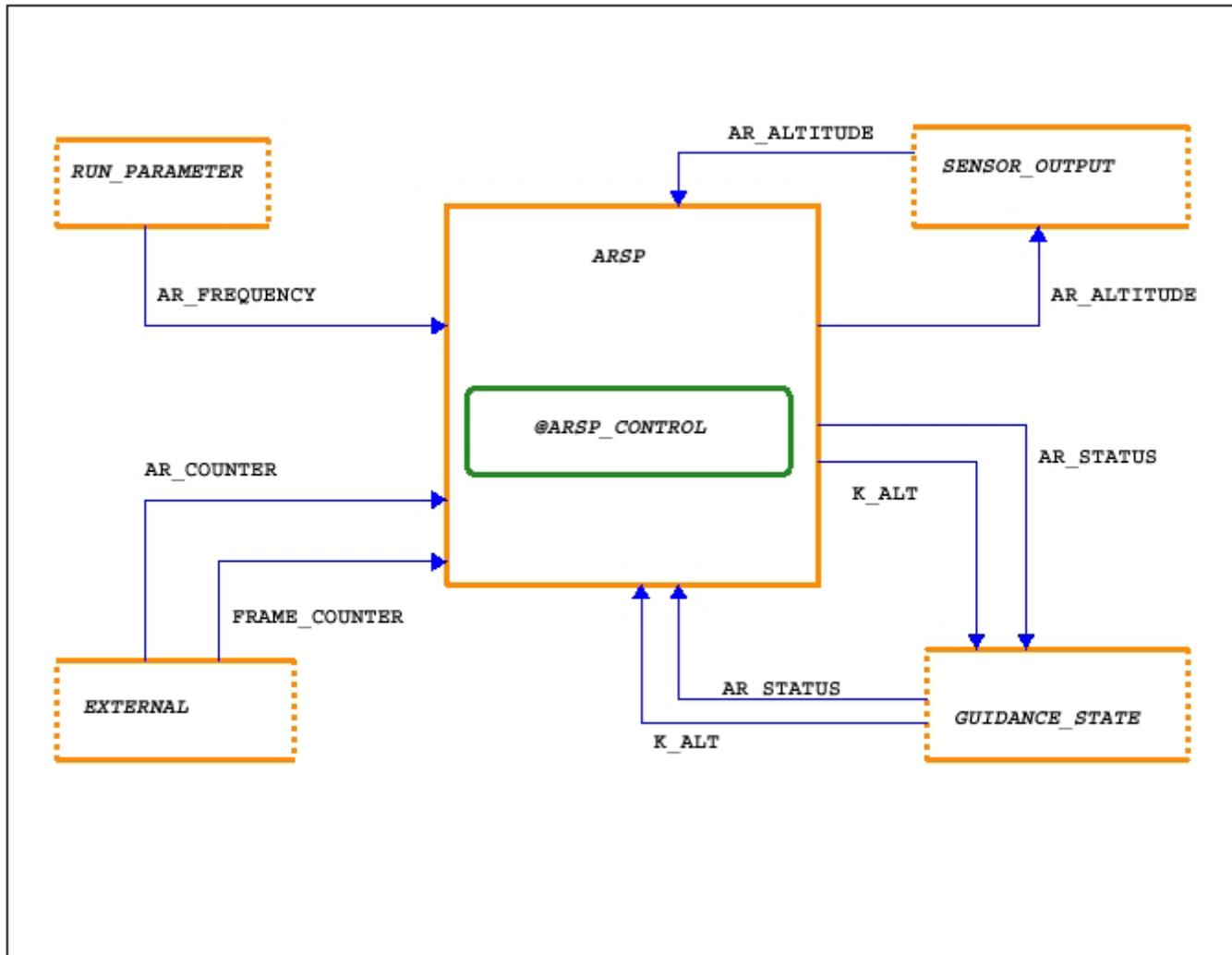


Figure 47. ARSP activity chart

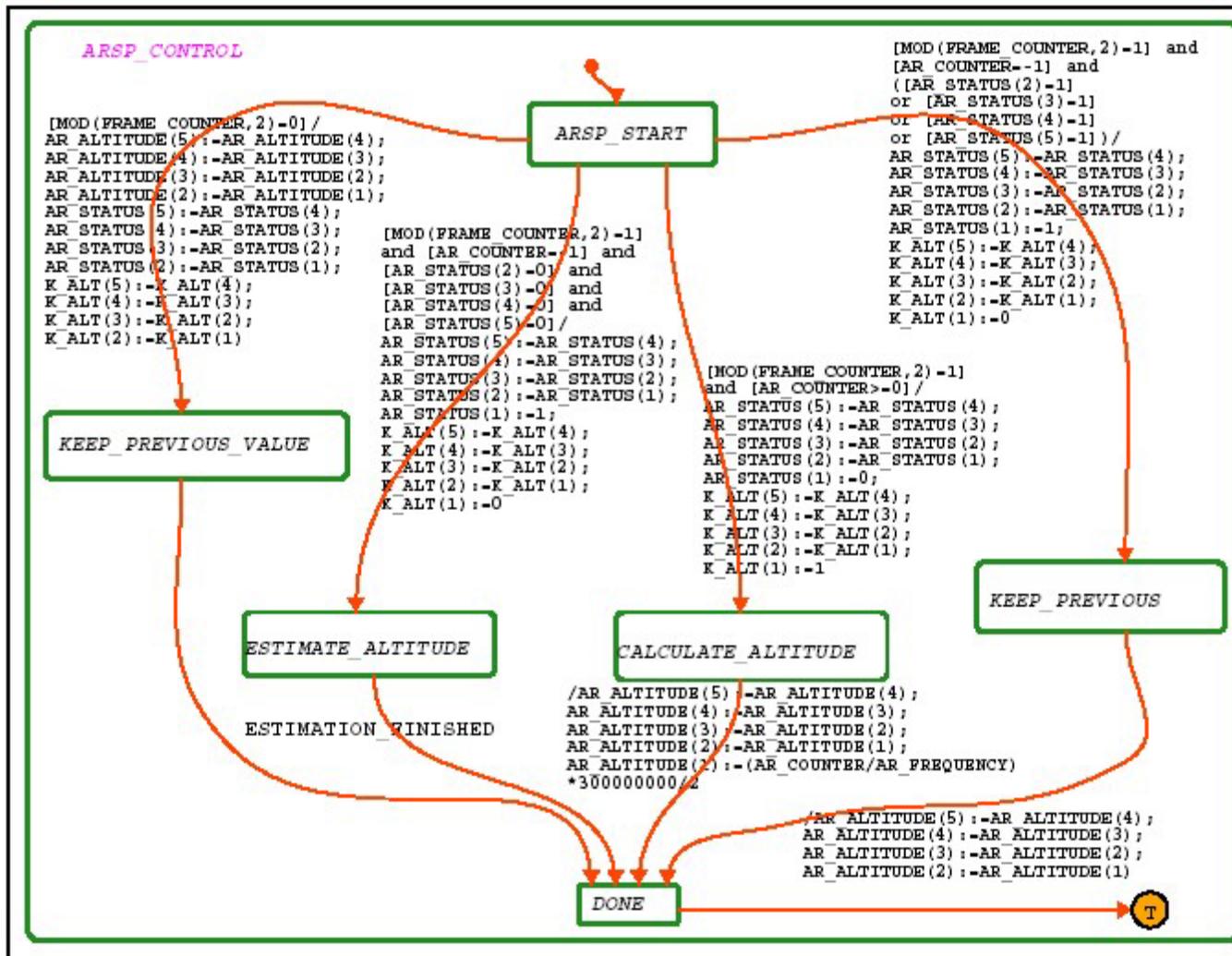


Figure 48. ARSP_CONTROL state chart

The Figure 47 and Figure 11 are the equivalent activity charts. Figure 48 represents the Z specification of the ARSP submodule shown in Figure 29. Statecharts model in the chapter 4.3 has the ARSP activity, the CALCULATE sub-activity and two control states. The ARSP submodule statecharts model in this section is consists of one activity and one control state based on the Z specification presented in the chapter 5.1.2.

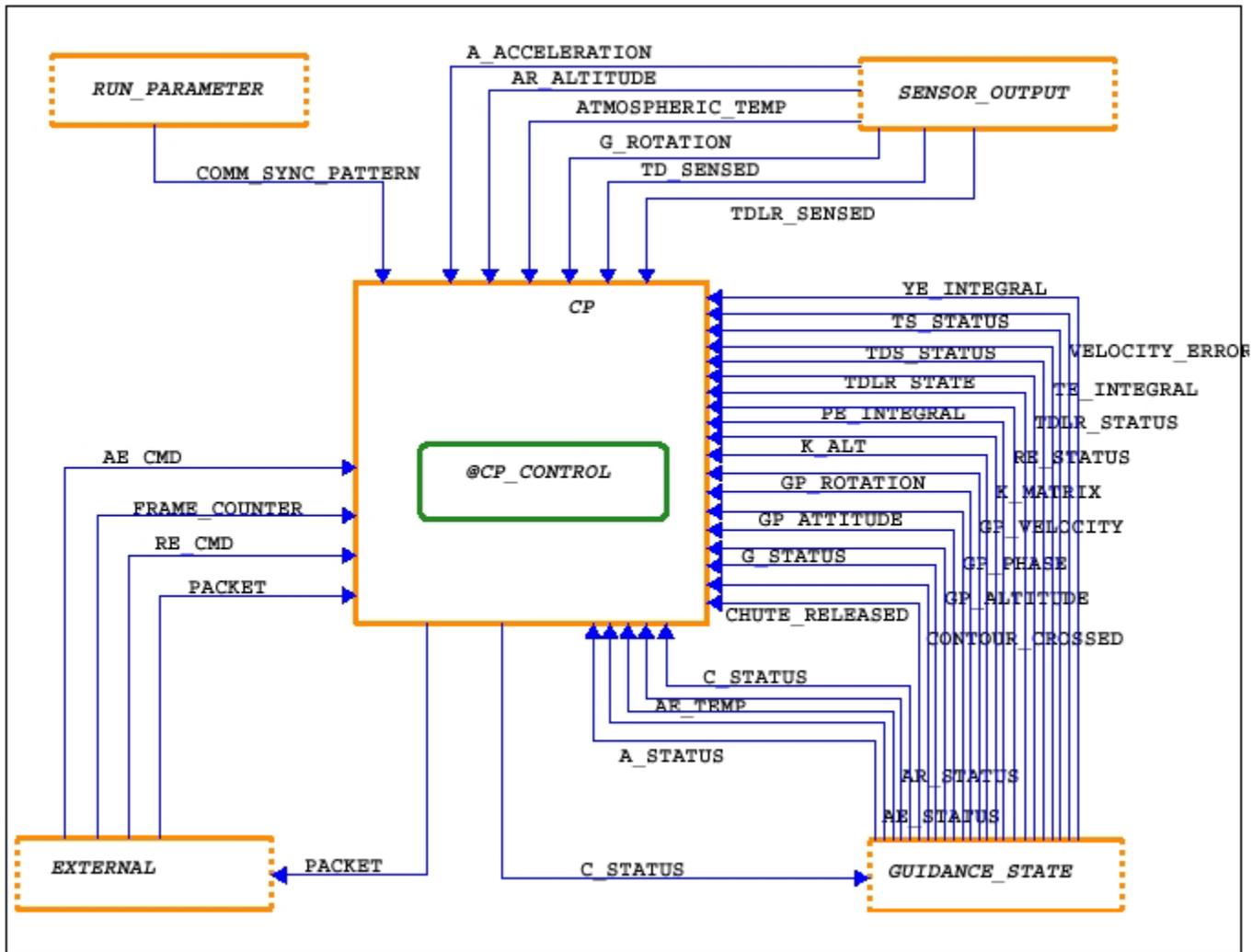
This ARSP model has 4 distinctive path. The simulation results of the state transition path are as presented in the following table.

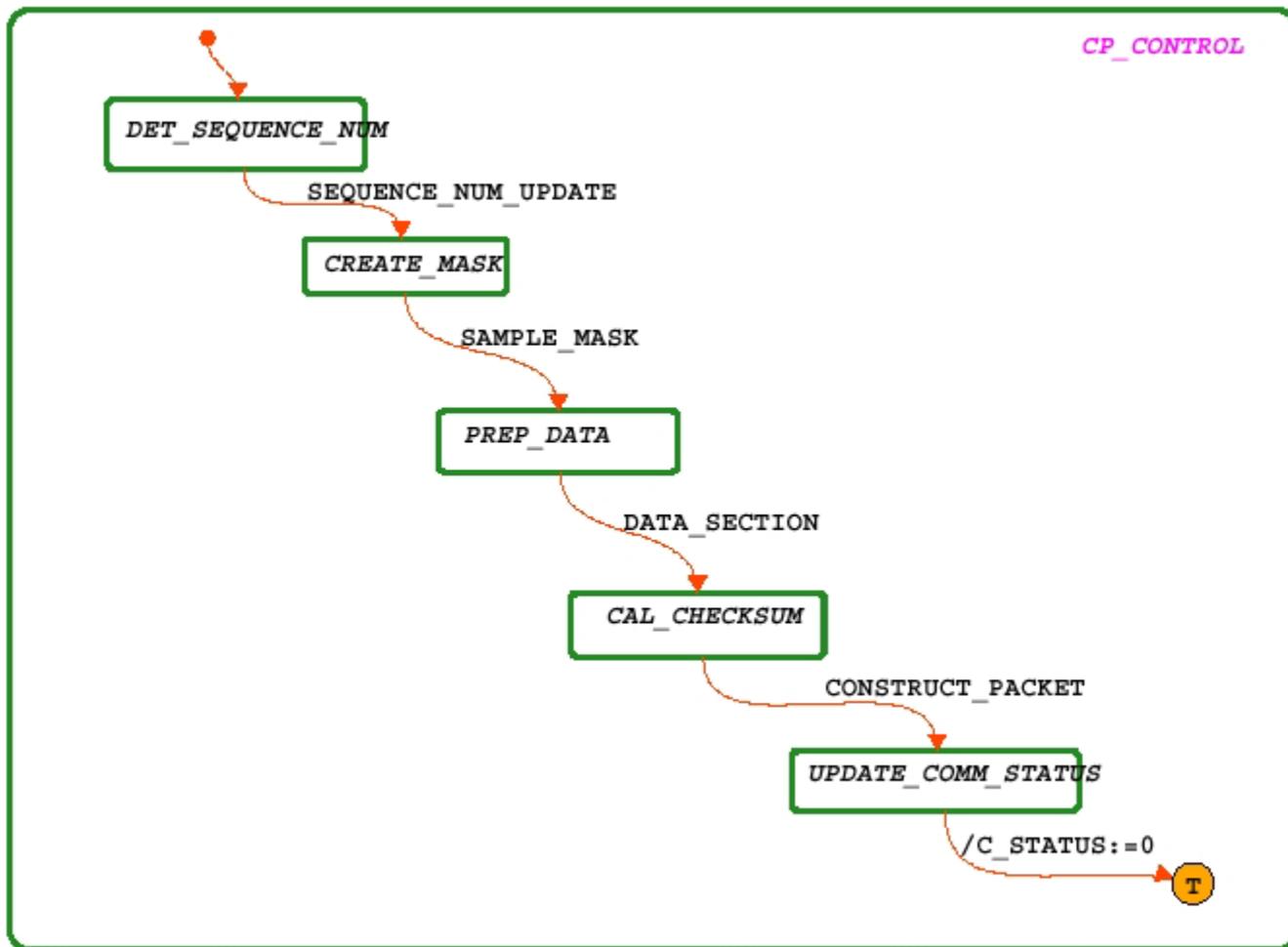
Name of Chart	Activity / State Name	Activity/State Transition Paths			
		1	2	3	4
ARSP	ARSP	E ₁	E ₁	E ₁	E ₁
	@ARSP_CONTROL	E ₂	E ₂	E ₂	E ₂
ARSP_CONTROL	ARSP_START	E ₃	E ₃	E ₃	E ₃
	KEEP_PREVIOUS_VALUE	E ₄	-	-	-
	ESTIMATE_ALTITUDE	-	E ₄	-	-
	CALCULATE_ALTITUDE	-	-	E ₄	-
	KEEP_PREVIOUS	-	-	-	E ₄
	DONE	E ₅	E ₅	E ₅	E ₅

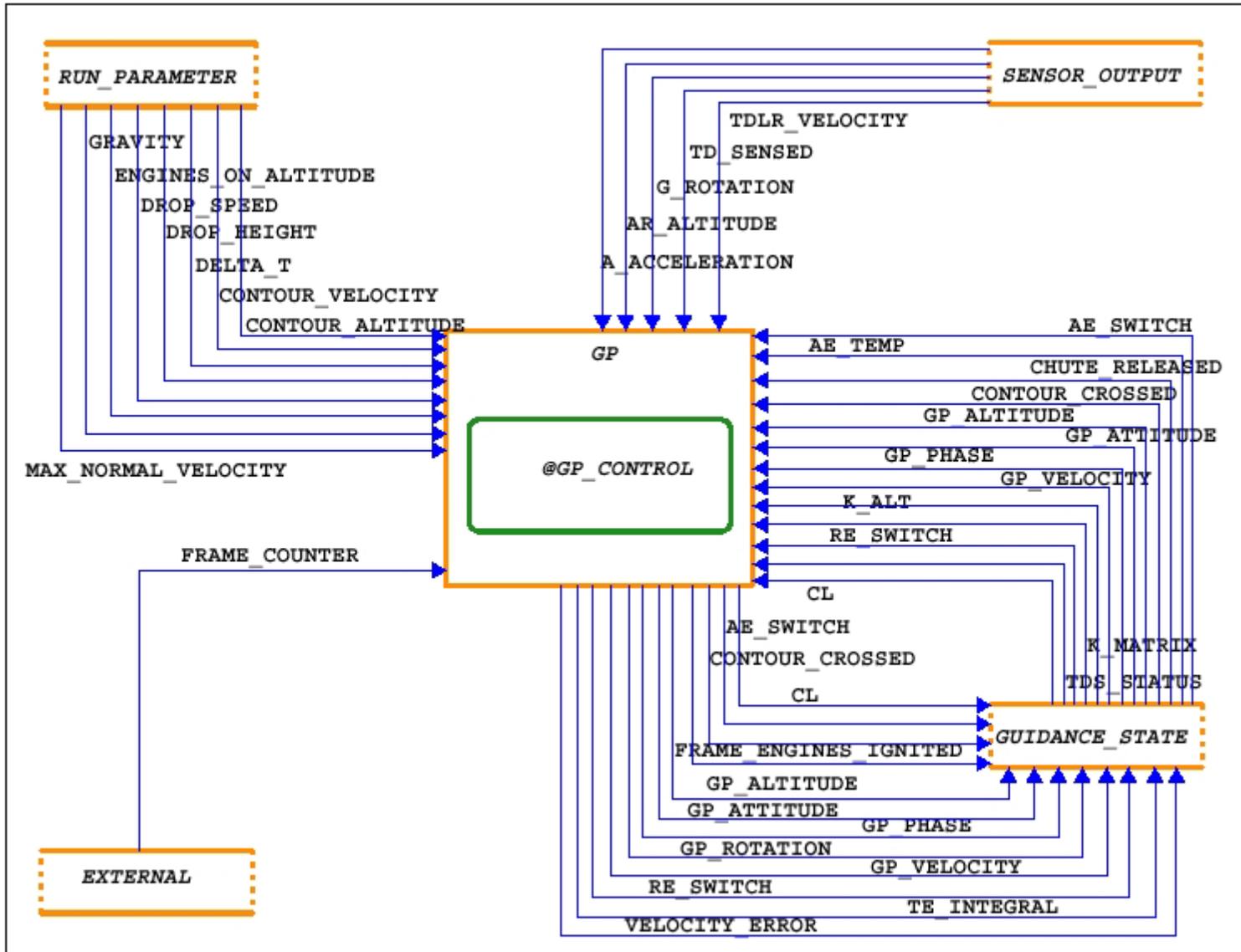
E_i entered in *i*th order, - not activated.

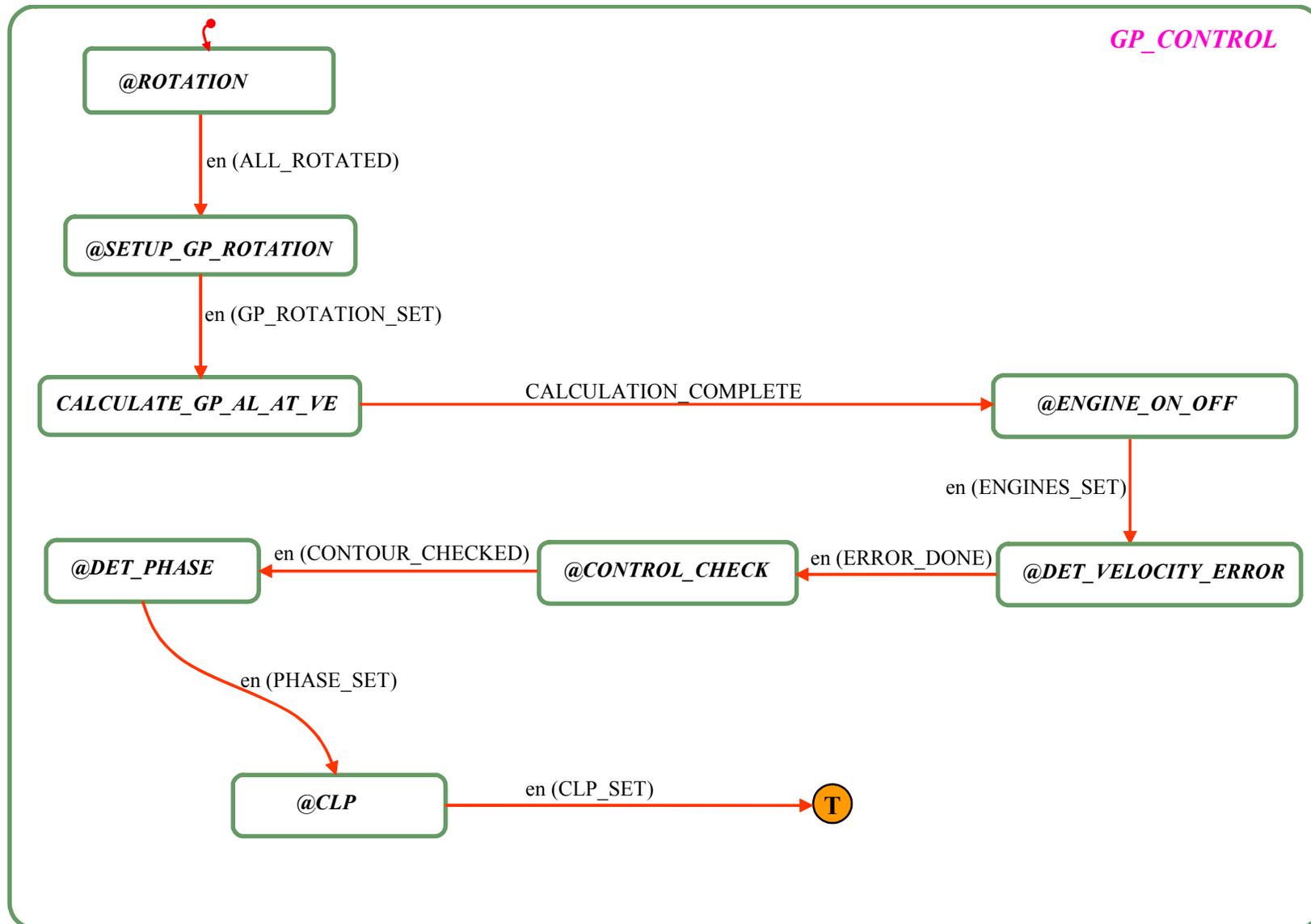
The test results using DIA are the same as shown in the chapter 4.3.3.2. The fault injection results are described in the following table.

Fault injected State	Altered state variable														
	FRAME_COUNTER					AR_COUNTER					AR_STATUS				
	Case					Case					Case				
	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
ARSP_START	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
KEEP_PREVIOUS_VALUE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ESTIMATE_ALTITUDE	✓	✓	✓	✓	✓	✓	✓	N/A	✓	✓	✓	✓	N/A	✓	✓
CALCULATE_ALTITUDE	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓	✓
KEEP_PREVIOUS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DONE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

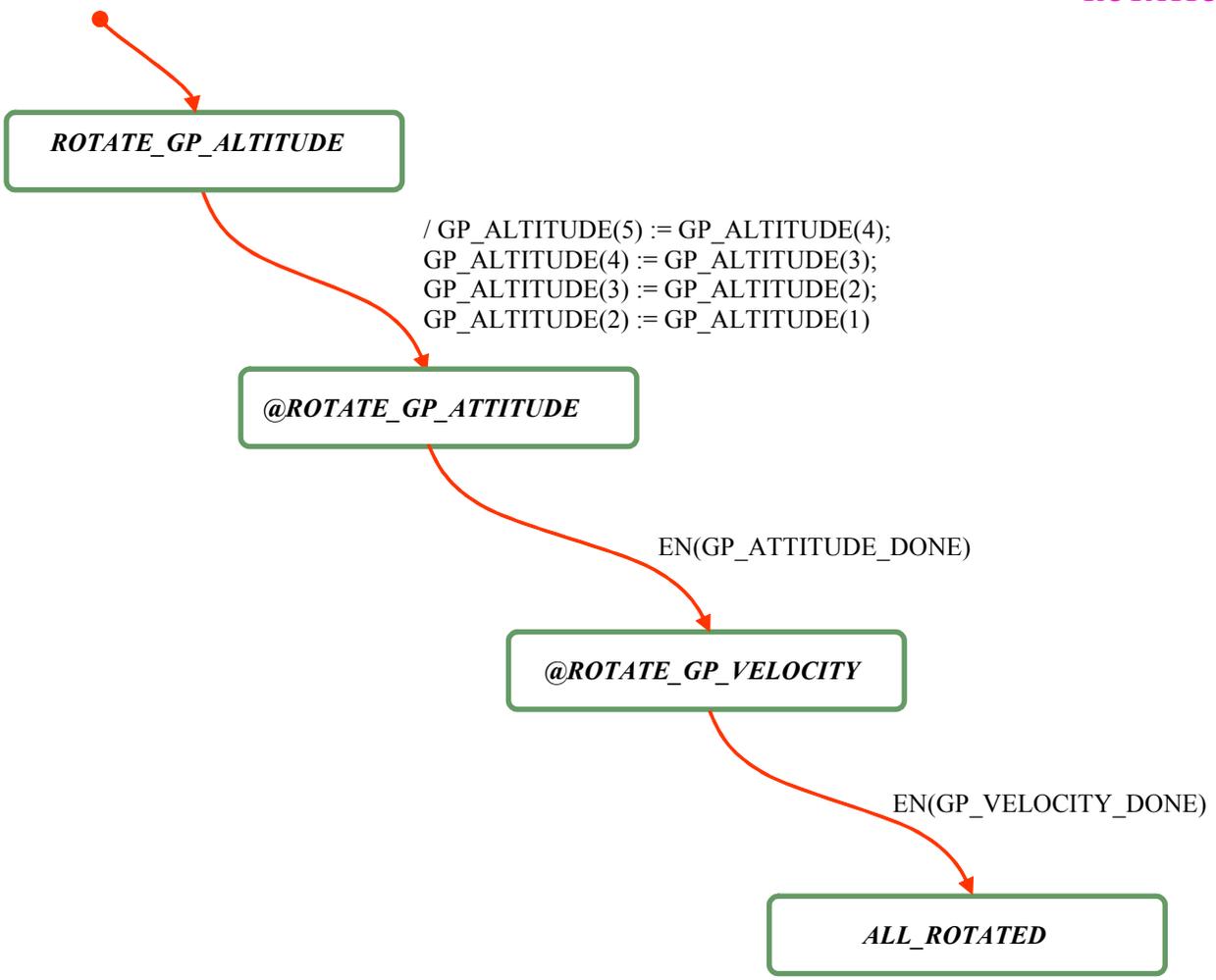








ROTATION



```
/ GP_ATTITUDE(45) := GP_ATTITUDE(36); GP_ATTITUDE(44) := GP_ATTITUDE(35);  
GP_ATTITUDE(43) := GP_ATTITUDE(34); GP_ATTITUDE(42) := GP_ATTITUDE(33);  
GP_ATTITUDE(41) := GP_ATTITUDE(32); GP_ATTITUDE(40) := GP_ATTITUDE(31);  
GP_ATTITUDE(39) := GP_ATTITUDE(30); GP_ATTITUDE(38) := GP_ATTITUDE(29);  
GP_ATTITUDE(37) := GP_ATTITUDE(28)
```

ROTATE_GP_ATTITUDE

GP_ATTITUDE_5

```
/ GP_ATTITUDE(36) := GP_ATTITUDE(27); GP_ATTITUDE(35) := GP_ATTITUDE(26);  
GP_ATTITUDE(34) := GP_ATTITUDE(25); GP_ATTITUDE(33) := GP_ATTITUDE(24);  
GP_ATTITUDE(32) := GP_ATTITUDE(23); GP_ATTITUDE(31) := GP_ATTITUDE(22);  
GP_ATTITUDE(30) := GP_ATTITUDE(21); GP_ATTITUDE(29) := GP_ATTITUDE(20);  
GP_ATTITUDE(28) := GP_ATTITUDE(19)
```

GP_ATTITUDE_4

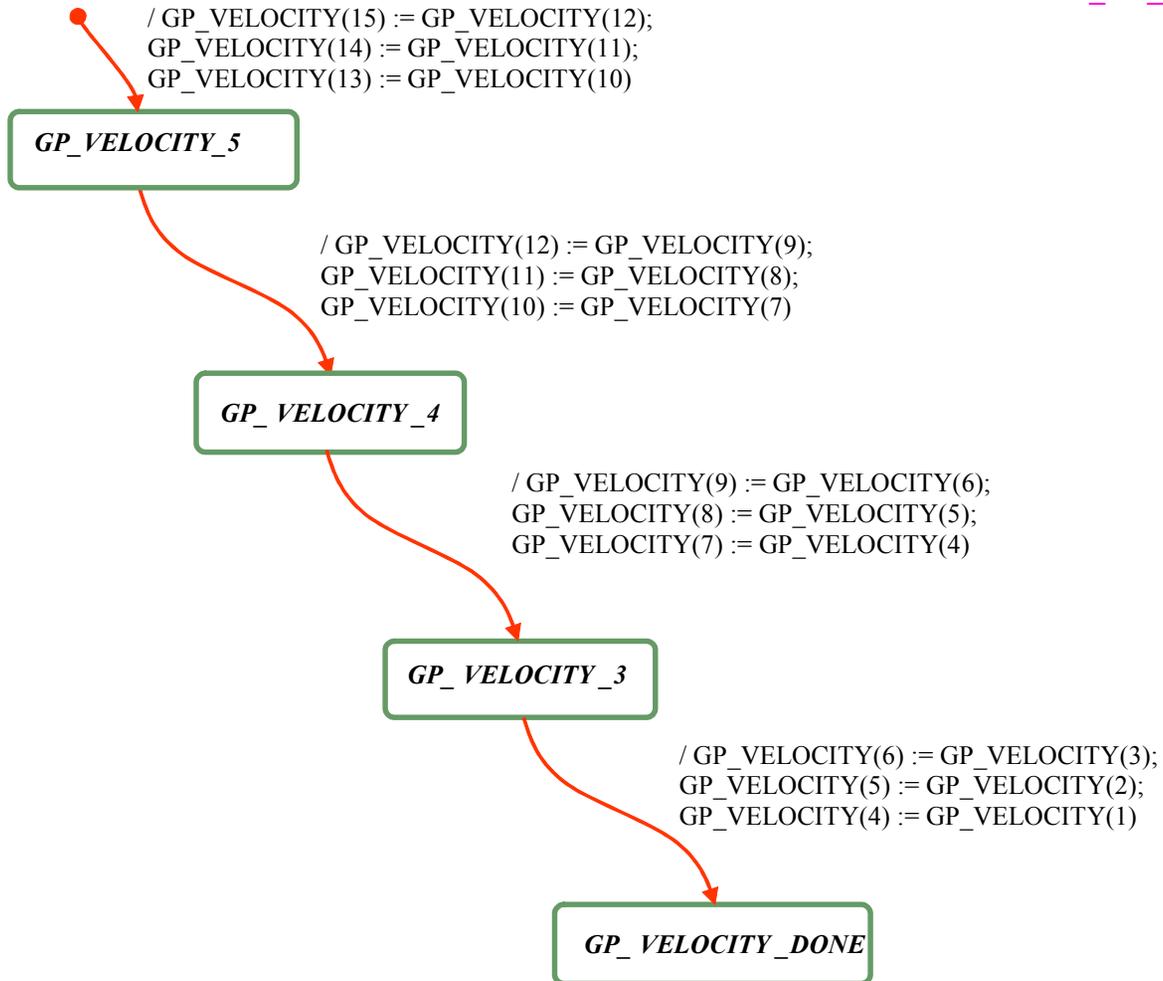
```
/ GP_ATTITUDE(27) := GP_ATTITUDE(18); GP_ATTITUDE(26) := GP_ATTITUDE(17);  
GP_ATTITUDE(25) := GP_ATTITUDE(16); GP_ATTITUDE(24) := GP_ATTITUDE(15);  
GP_ATTITUDE(23) := GP_ATTITUDE(14); GP_ATTITUDE(22) := GP_ATTITUDE(13);  
GP_ATTITUDE(21) := GP_ATTITUDE(12); GP_ATTITUDE(20) := GP_ATTITUDE(11);  
GP_ATTITUDE(19) := GP_ATTITUDE(10)
```

GP_ATTITUDE_3

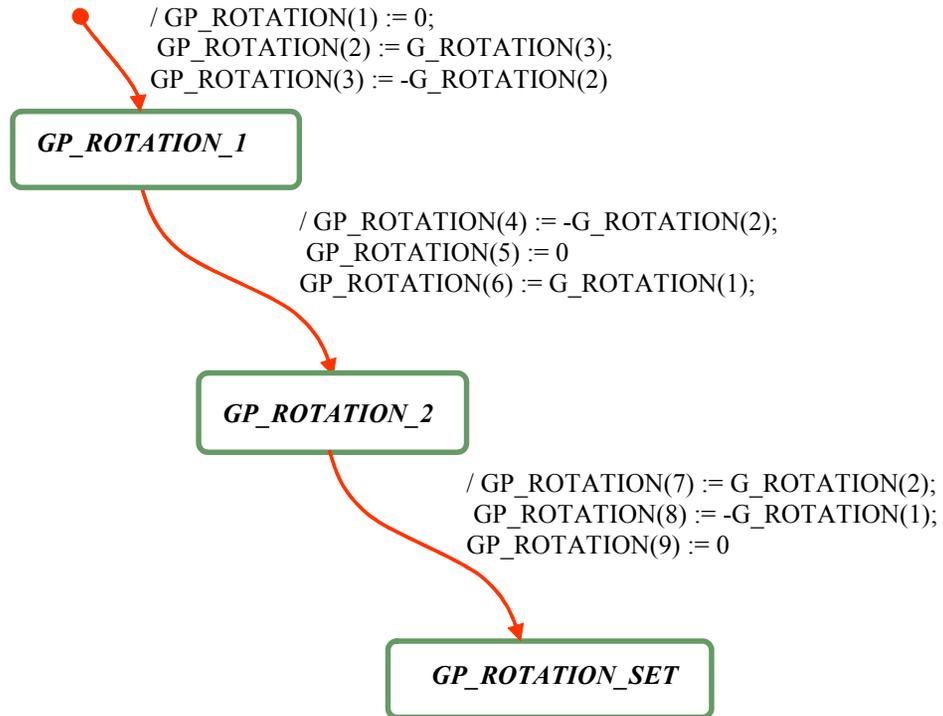
```
/ GP_ATTITUDE(18) := GP_ATTITUDE(9); GP_ATTITUDE(17) := GP_ATTITUDE(8);  
GP_ATTITUDE(16) := GP_ATTITUDE(7); GP_ATTITUDE(15) := GP_ATTITUDE(6);  
GP_ATTITUDE(14) := GP_ATTITUDE(5); GP_ATTITUDE(13) := GP_ATTITUDE(4);  
GP_ATTITUDE(12) := GP_ATTITUDE(3); GP_ATTITUDE(11) := GP_ATTITUDE(2);  
GP_ATTITUDE(10) := GP_ATTITUDE(1)
```

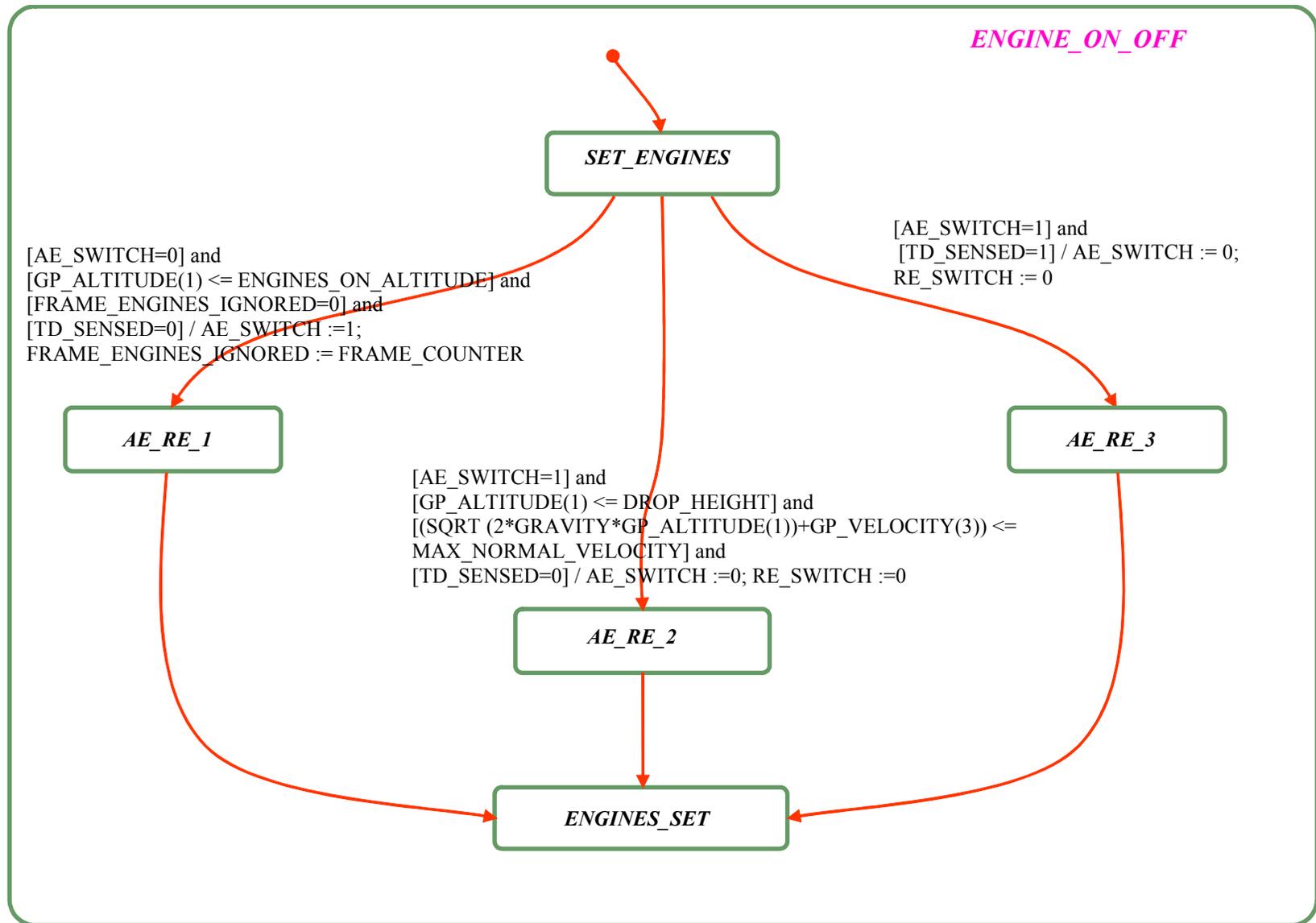
GP_ATTITUDE_DONE

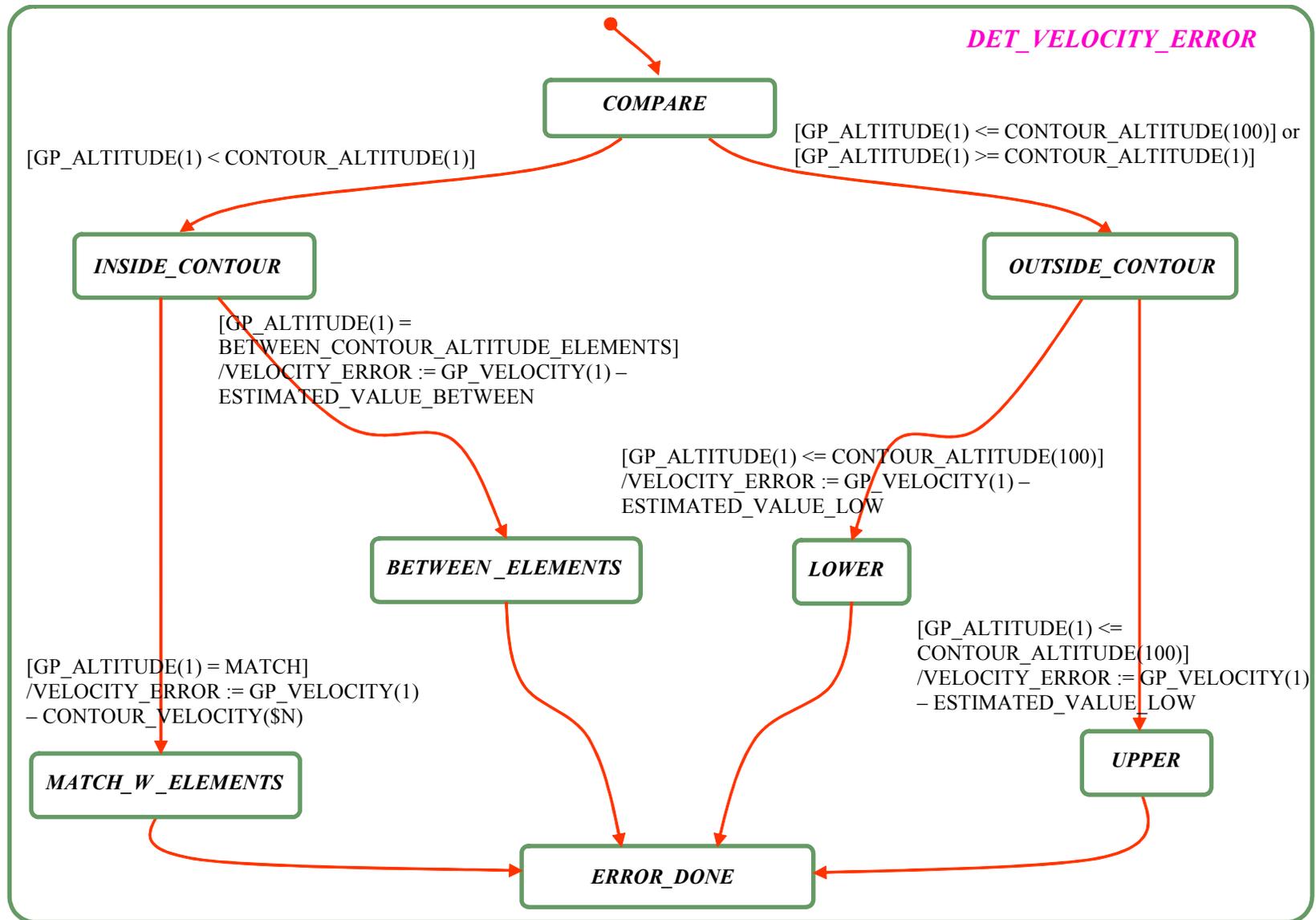
ROTATE_GP_VELOCITY



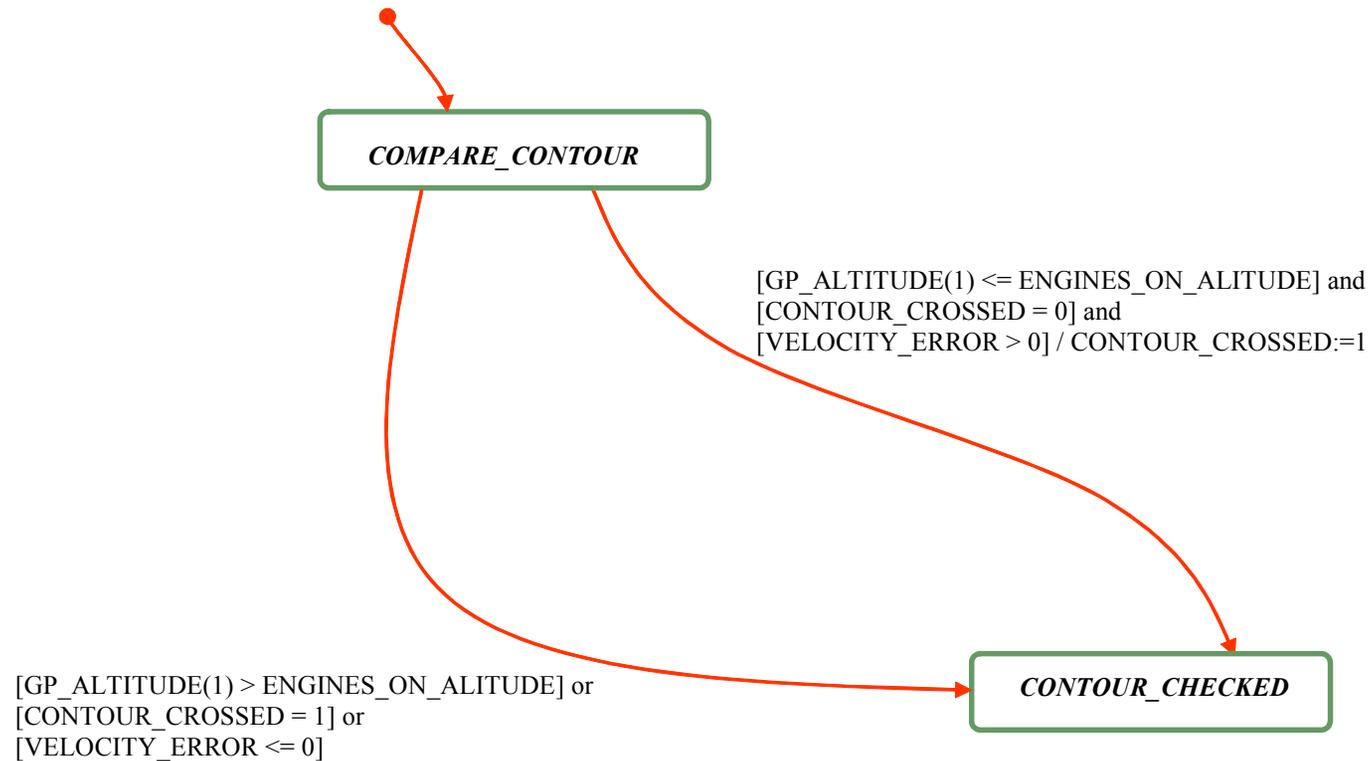
SETUP_GP_ROTATION

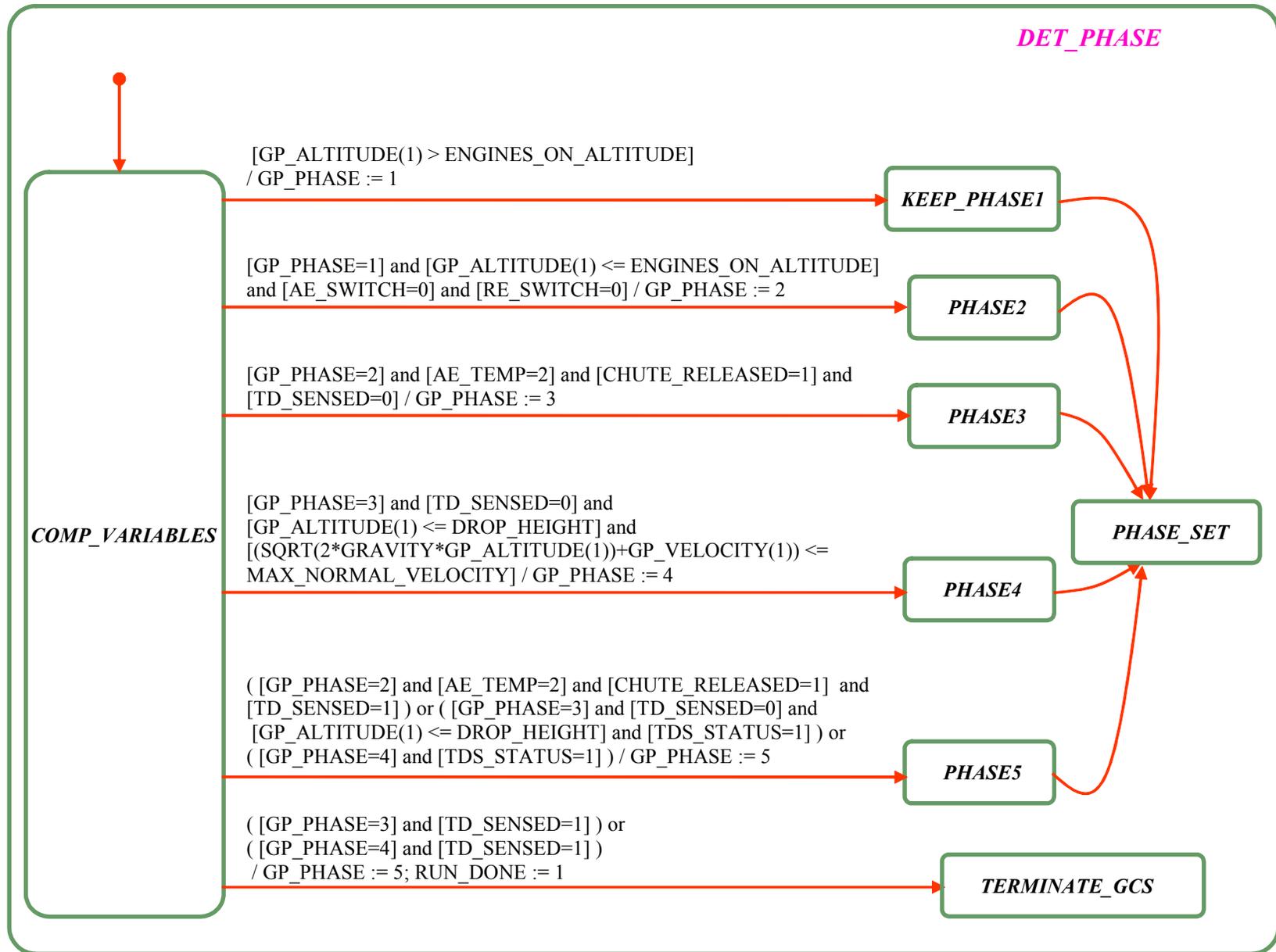




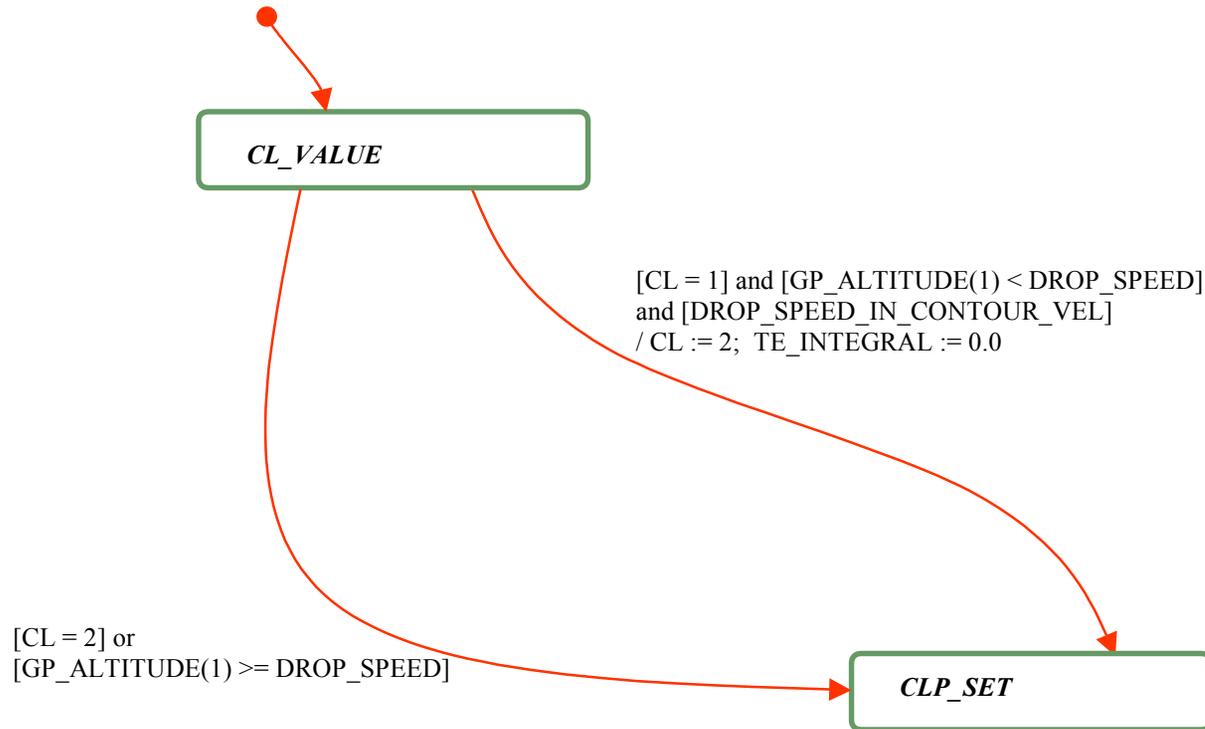


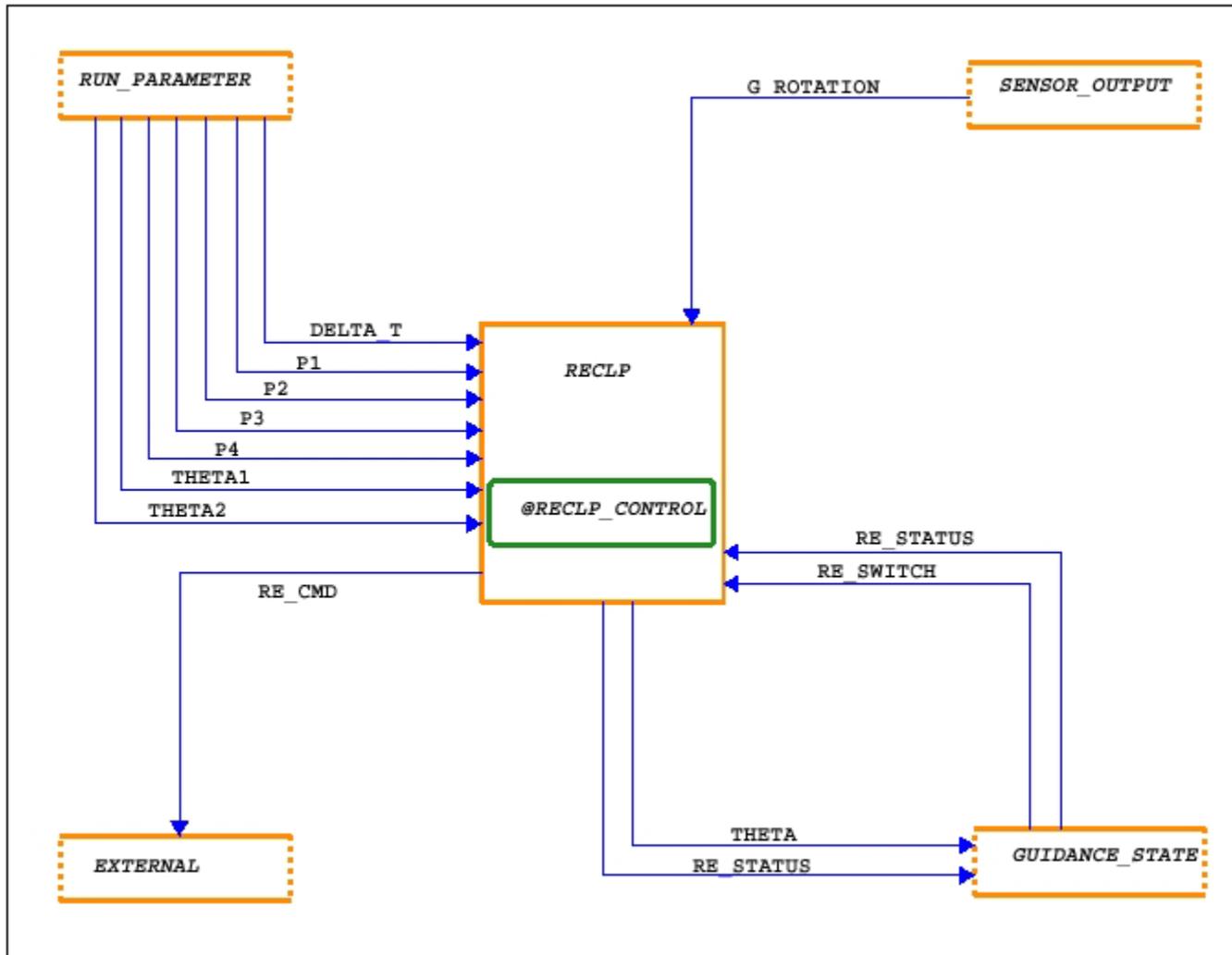
CONTOUR_CHECK



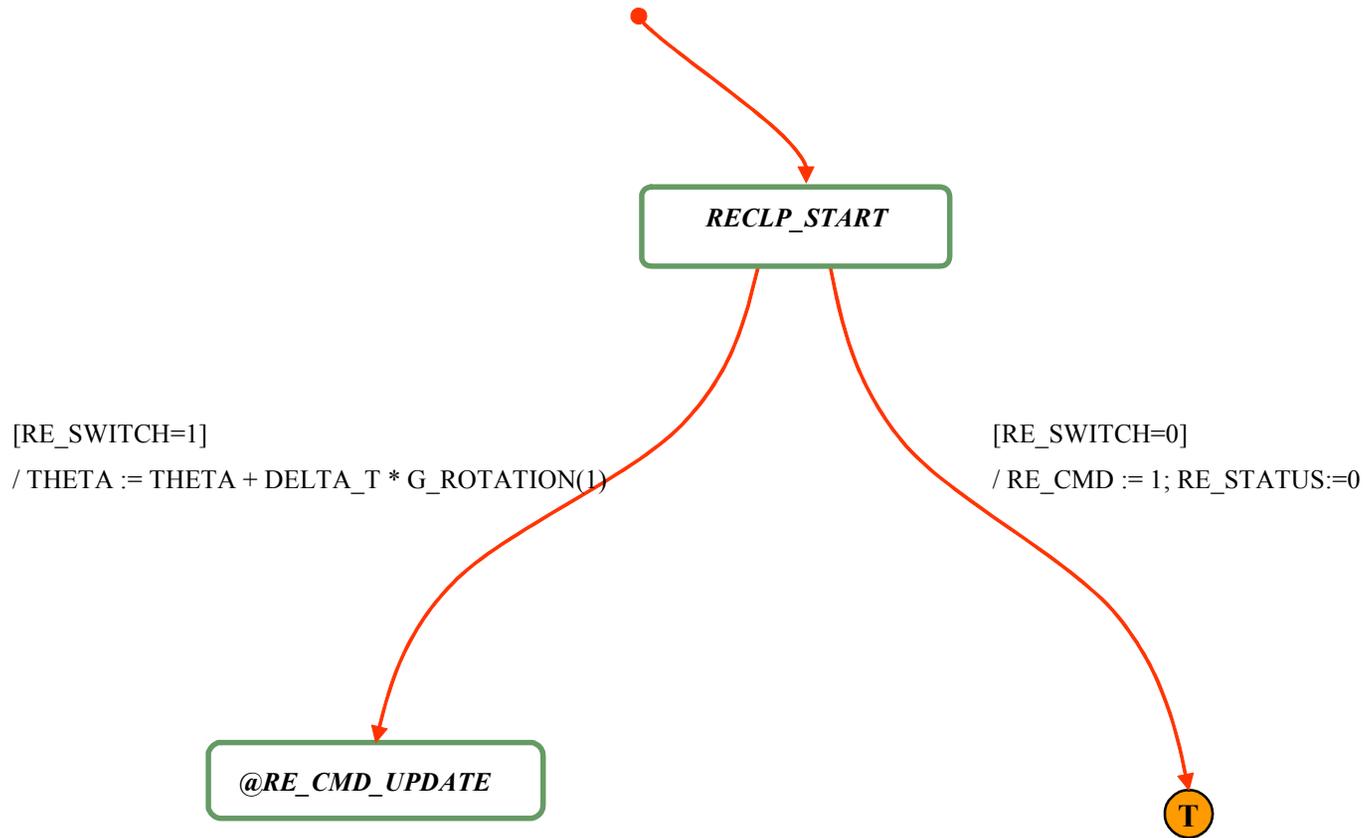


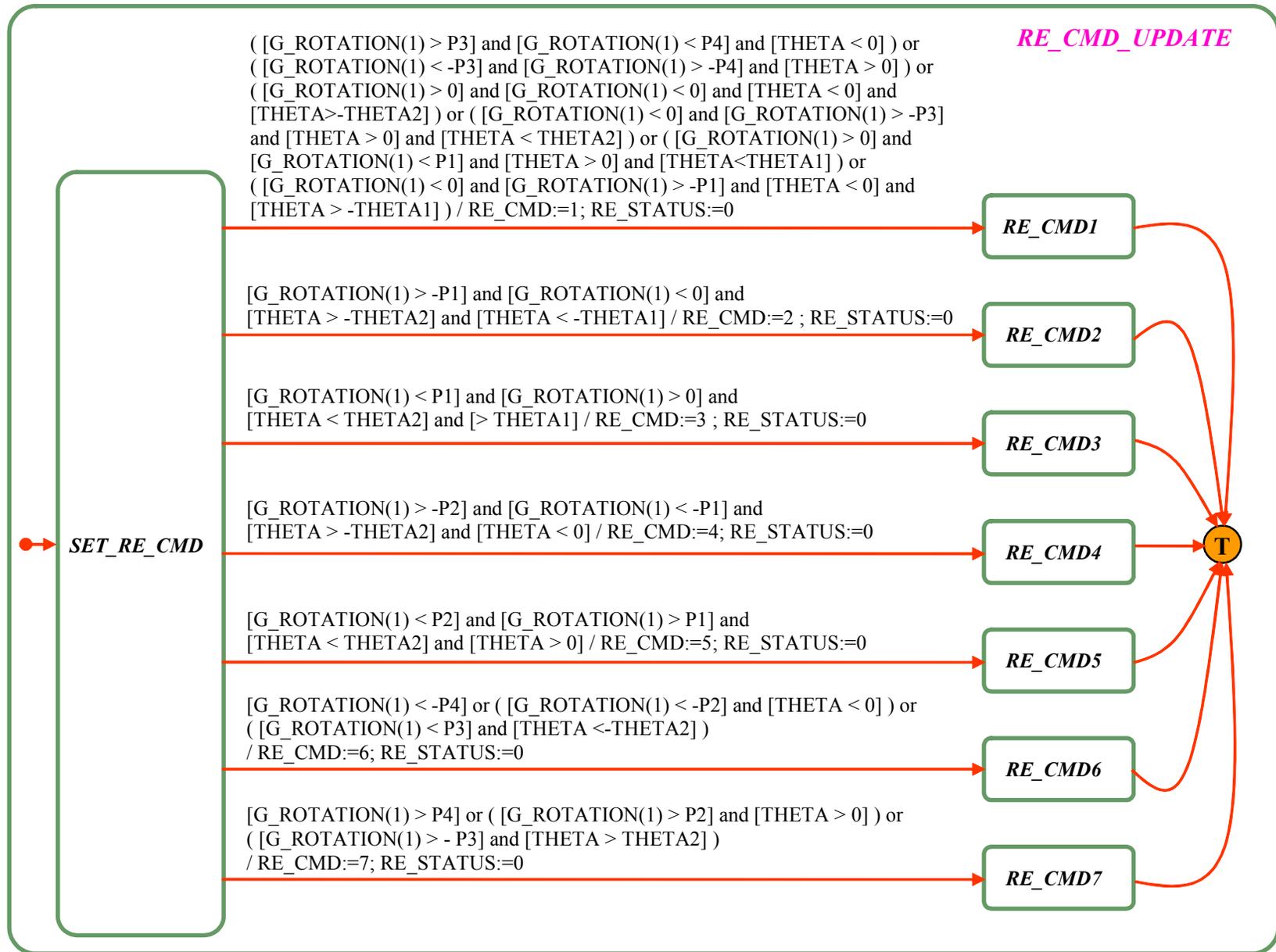
CLP





RECLP_CONTROL





The CP submodule contains too much of inconsistency to develop complete Statecharts model. Moreover, the bit wise transactions to build the packet mask are too complicated to transform into statecharts (Covered by CP_PREP_MASK1-3 and CP_MASK schemas in Z). Therefore, the CP statecharts model is build with events that represents the functional sequences that CP follows according to the SRS. CP has only one state transaction path which is tested using FSMA. The fault injection and DIA test are not performed for this submodule because CP model did not have enough data processing functionality and CP is not a submodule that can create catastrophic failure for the system.

The GP submodule has multiple functions to perform. All the sequences of functions are transformed into Statecharts model. However, it was impossible to test all the data input and output with realistic variable values because the initial values of all the variables are not clearly given. Therefore, FSMA test was performed to entire GP model while DIT test is performed on some parts of GP model that uses only the variables processed inside of the GCS excerpt.

The RECLP is comparable complete and has several data initiated function processes. Therefore, FSMA and DIT test and Fault injections are performed on this submodule. The test results are as presented in the following tables.

Name of Chart	Activity / State Name	Activity/State Transition Paths							
		1	2	3	4	5	6	7	8
RECLP	RECLP	E ₁	E ₁	E ₁	E ₁	E ₁	E ₁	E ₁	E ₁
	@RECLP_CONTROL	E ₂	E ₂	E ₂	E ₂	E ₂	E ₂	E ₂	E ₂
RECLP_CONTROL	RECLP_START	E ₃	E ₃	E ₃	E ₃	E ₃	E ₃	E ₃	E ₃
	@RE_CMD_UPDATE	-	E ₄						
RE_CMD_UPDATE	SET_RE_CMD	-	E ₅						
	RE_CMD1	-	E ₆	-	-	-	-	-	-
	RE_CMD2	-	-	E ₆	-	-	-	-	-
	RE_CMD3	-	-	-	E ₆	-	-	-	-
	RE_CMD4	-	-	-	-	E ₆	-	-	-
	RE_CMD5	-	-	-	-	-	E ₆	-	-
	RE_CMD6	-	-	-	-	-	-	E ₆	-
RE_CMD7	-	-	-	-	-	-	-	E ₆	

Variable values (constants)

<i>Variable name</i>	<i>Values</i>
DELTA_T	0.005
P1	0.005
P2	0.010
P3	0.015
P4	0.020
THETA1	0.010
THETA2	0.020

	Variable	Case 1	Case 2	Case 3	Case 4
Input	G_ROTATION 1	0.016	-0.016	0.01	-0.01
	THETA	-0.00500	0.005	-0.005	0.01
Output	THETA	-0.00492	0.00492	-0.00495	0.00995
	RE_CMD	1	1	1	1
	RE_STATUS	0	0	0	0

	Variable	Case 5	Case 6	Case 7	Case 8
Input	G_ROTATION 1	0.001	-0.001	-0.001	0.001
	THETA	0.005	-0.005	-0.015	0.015
Output	THETA	0.005005	-0.005005	-0.015005	0.015005
	RE_CMD	1	1	2	3
	RE_STATUS	0	0	0	0

	Variable	Case 9	Case 10	Case 11	Case 12
Input	G_ROTATION 1	-0.006	0.006	-0.025	-0.015
	THETA	-0.01	0.01	0	-0.001
Output	THETA	-0.01003	0.01003	-0.000125	-0.001075
	RE_CMD	4	5	6	6
	RE_STATUS	0	0	0	0

	Variable	Case 13	Case 14	Case 15	Case 16
Input	G_ROTATION 1	0.01	0.025	0.015	-0.01
	THETA	-0.021	0	0.01	0.025
Output	THETA	-0.02095	0.000125	0.010075	0.02495
	RE_CMD	6	7	7	7
	RE_STATUS	0	0	0	0

Where the boundary values of each variable belong are not clearly specified in the SRS. Due to that reason, the fault injection results are not conclusive.