Submtd Dec. 02 Software Quality Jr. *Special Issue on Software Measurement*, Kluwer, Dordrecht Netherlands

Testing Software Requirements with Z and Statecharts Applied to an Embedded Control System[1]

Hye Yeon Kim *
Samsung Electro-Mechanics Network_Bluetooth Research Group
314, Meatan-3Dong, Paldal-Gu,
Suwon, Kyounggi-Do, Korea 442-743
http://www.wsu.edu/~dustkhy
hyekim@ieee.org

Frederick T. Sheldon[2]
Oak Ridge National Laboratory CSE Div/ASE Lab
Oak Ridge, Tennessee 37831-6363
http://computing.ornl.gov/cse_home/aser.shtml
sheldon@acm.org

Abstract

Software development starts from specifying the requirements. 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 development cycle. To avoid problems in the latter development phases and reduce life-cycle costs, it is crucial to ensure that the specification be reliable. In this paper, we describe how to model and test (i.e., check, examine, verify and prove) the SRS using two formalisms (Z and Statecharts). Moreover, these formalisms were used to determine strategies for avoiding design defects and system failures. We introduce a case study performed to validate the integrity of a Guidance Control Software SRS in terms of completeness, consistency, and fault-tolerance.

Keywords: Z; Statecharts; Requirements specification validation; Completeness; Consistency; Fault-tolerance

---

[1] The appendix gives the total system architecture (see Figure A-1) and all the Statecharts. It is included as background material only. We do not plan to include it in the final version. The final proved Z schemas can be obtained from http://www.eecs.wsu.edu/seds/hkim_thesis_final_ilogix.pdf or the http://www.ilogix.com university page.

[*] Most of this research was completed when Ms. Kim was a graduate student in Washington State University (WSU). She was a founding member of the Software Engineering for Dependable Systems Laboratory (http://www.eecs.wsu.edu/seds) which was relocated with Dr. Sheldon.
We would like to thank Kshamta Jerath (PhD. Student, WSU) for her valuable critique and Markus Degen (DaimlerChrysler System Safety [RIC/AS Stuttgart]) for reinforcing the need for a reliable software requirement specification especially in mission/safety critical applications.

[2] Dr. Sheldon (865-576-1339, 865-574-6275 fax), is currently a member of the research staff at ORNL and director of the SEDS (Software Engineering for Dependable Systems) Lab. Much of this work was completed while he was a research staff member at DaimlerChrysler (RIC/AS) on leave of absence from WSU.

1 INTRODUCTION

The trend of using software in embedded real-time systems and the fact that software requirements are often complex necessitate the use of formal and rigorous approaches in the specification and validation of requirements. Requirements validation is concerned with checking the requirements document (i.e., SRS [Software Requirements Specification]) for consistency, completeness and accuracy [1]. In other words, validation ensures the specification represents a clear description of the system for design and implementation and is a final check that the requirements meet stakeholders' needs. This process should be complete prior to design and implementation. In this paper, the SRS of an embedded real-time software - the Viking Mars Lander Guidance Control Software [2]- is validated.

The notations 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 through some means of simulations. The notation should make it possible to state and prove properties of a system before it is built; then, if the system is constructed according to the specifications, there will be a high level of confidence that the system will 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 [3].

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. The approach chosen combines a model-based FM which uses the theory of sets, propositional and predicate logic with a state-based diagrammatic formalism to visualize and simulate the specification (including fault injection). The Z is employed to prove correctness of the SRS while the behavior of executable specifications is gauged through visualization and simulation using Statecharts.

1.1 Definitions

Reliability, as applied to the SRS, investigates the questions: (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 multiple interpretations [4]. Consistency, the presence of a lack of ambiguity in requirements, means the specification is free from conflicting requirements and *undesired* nondeterminism [5].

Typically, fault-tolerance is considered as an 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 [6]. For the SRS, it can be defined as (1) existence of specified requirements to detect errors for all fault conditions, and (2) presence of specified requirements that support the system robustness, software diversity, and temporal redundancy for continuing service of critical system functions in the case of failure.

2 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 studies are presented that aim to find a way to verify the consistency and completeness of SRSs. Numerous other case studies are reviewed that use Z and other formal methods, to gain benefit from visualization and/or dynamical assessment.

2.1 Formal Methods

Formal methods are a collection of techniques, rather than a single technology, most notably for specifying a software system. The sole objective is to provide a way to eliminate inconsistency, incompleteness, and ambiguities. Because formal methods have an underlying

mathematical basis, they provide valid analysis of a system better than ad hoc reviews. There are several classes of distinguishable formal specification techniques. They are property-oriented specifications, model-oriented specifications, and operational specifications [7].

In the property-oriented approaches, known as constructive techniques, one declares a name list of functions and properties. 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 [8]. The algebraic specification describes a system consisting of a set of data and a number of functions over this set [9]. 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 [8].

The objective of the model-oriented approach, known as declarative techniques, is to build a unique model from a choice of built-in data structures and construction primitives provided by the specification language [7]. 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 [8]. Model behaviors are compared against the specified functionality as a measure of correctness [7]. Vienna Development Method (VDM), B, and Z belong to 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 STATEMATE[3] environment [3] are within this category [7].

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., proposed an automatic evaluation method called "Quality Analyzer of Requirements Specification (QuARS)" to evaluate quality. They define testability, completeness, understandability, and

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

consistency as properties of a high quality SRS [10]. The QuARS tool parses requirement sentences written in natural language (NL) to detect potential sources of errors. This is a linguistic, informal evaluation approach rather than a formal method but it shows that informal systematic methods are useful for revealing errors. The authors claim this approach can be used for any domain based on the tool's ability to customize its dictionaries.

Heitmeyer et al., used the Software Cost Reduction (SCR) tabular notation to identify inconsistencies in SRSs. They describe, using this notation/method, how a safety violation is exposed. Typically, the enormous state space of practical software specifications renders direct analysis impractical [11]. They used the "Two Pushbutton" abstraction method to reduce a system state space from infinite to finite. Two redundant specifications represent the required system behavior using both Petri-net and TRIO specification logic. They abstracted and analyzed their SRS with Spin and a simulator that was developed to support the SCR method.

Heimdahl and Leveson used their Requirements State Machine Language (RSML) to verify requirements specifications for completeness and consistency [12]. RSML is a state-based language suitable for the specification of reactive systems. It includes several features developed by Harel for Statecharts. In RSML, the transitions are represented as relationships between states (i.e., hierarchical, next-state mappings). The functional framework defined in [12] is used to check the model against every possible input to find conflicting requirements (i.e., to verify whether the model is deterministic). They used a textual-representation-based simulator developed for RSML to execute the specification. One advantage is the ability to analyze subparts of the whole system without the need to generate a global reachability graph.

2.3 Related Z Case Studies

Numerous studies have been conducted that combine Z with other formal methods. A hybrid formal method called PZ-nets is proposed by Xudong He. PZ-nets combine Petri nets and Z [13]. 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 μSZ [14]. The language uses Statecharts to describe the dynamical system behavior and Z to describe the data and their transformations. In μSZ, Statecharts define sequencing while Z is used to define the data and operations. They abstracted data from the Z specifications to produce an Extended Finite State Machine (EFSM) represented with Statecharts. EFSM features can be utilized for test case generation. These features automate setting up the initial state and checking the final state for each test. The dynamic system behaviors specified in Statecharts are checked using these features.

Bussow and Weber present a mixed method consisting of Z and Statecharts [15]. Each method was applied to a separate part of the system. Z was used to define the data structures and transformations. Statecharts were used to represent the overall system and 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 [16-18].

Castello developed a framework for the automatic generation of Statecharts layouts from a database that contains information abstracted from an SRS [19]. The framework centerpiece is the "statecharts layout" tool. The tool's output is then transformed into Z schemas. Data is abstracted from the SRS to generate a database that provides the basis from which to automatically generate "statecharts layout." Statecharts are translated one-by-one into Z schemas to validate the correctness. The Z schemas are exact replicas of the Statecharts (i.e., the Z schema is the text version of the Statecharts). Both the method and the criteria for the SRS abstraction are not explained [19].

2.4 Contribution from this study

In this study, Statecharts depend on the Z specification. The natural language based (NL-based) SRS components are translated completely into Z and then the Z specification is

completely translated into State/Activity charts. The Z specification is type checked and proved using Z/EVES[4] with reduction/refinement prior to the translation. The State/Activity charts are tested to determine consistency and completeness using simulations and model checking. The transformed SRS is evaluated for fault-tolerance by injecting faults into the Statecharts model. Details of the tests and fault injections are described in Sections 3 and 4.

Z and Statecharts do have different precision for revealing inherent SRS flaws. In general, it is believed that Z is more suitable for defining data types while Statecharts are better for describing the dynamic behavior (i.e., state transitions) [16-18]. When one uses conjoined methods like other case studies, the consistency between the joined methods is difficult to verify. Instead, we abstracted the SRS into Z schemas (method one) and then from Z to Statecharts (method 2). In this way a higher confidence in their consistency can be achieved. For example, the consistency of Z is verifiable using type-checking and proofs. The consistency and completeness of the Statecharts model are verifiable using the model checker and simulations. Refinement between these two different formalisms gives in-depth understanding of requirements, and reveals different flaws that exist in the SRSs. The usefulness of this approach was assessed in our case study by applying it to critical parts of the SRS.

3 APPLIED METHODS

A two-step process using Z/Statecharts is employed. First, the NL-based SRS is transformed using Z. Z is used because it provides a concrete way to transform the requirements into state-based models using schematic structuring facilities. The transformation elucidates assumptions and provides mechanisms for refining specifications by clarifying data and functional definitions. This compositional process helped to clarify ambiguities. For example, an ambiguity associated with the Altitude Radar Counter was uncovered during schema construction.

The variable AR_COUNTER is specified in two different sections (Table 1) in the SRS. The Processing Unit describes the AR_COUNTER modification rules and the value ranges.

[4] Z/EVES is a tool developed by ORA, Canada. It provides theorem proving, domain checking, type checking, precondition calculation, and schema expansion for Z specifications.

One can conclude from the first two sentences that the AR_COUNTER increases after the radar pulse is transmitted. However, this indicates that the AR_COUNTER value is a positive number when the radar pulse is transmitted irrespective of whether an echo has arrived or not. Furthermore, this conflicts with the last sentence that states that the AR_COUNTER will contain sixteen one bits representing a negative one (-1) according to the definition in the data dictionary.

Table 1. NL-based specification for AR_COUNTER [2]

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

Second, the Schemas are manually transformed into State/Activity charts and symbolically executed to assess the model's behavior based on the GCS specified mission profile. Developing State/Activity charts from the Z schema is not a direct/mechanical transformation process and requires an in-depth knowledge of Z. One can specify a countably infinite number of system states using Z. To develop Statecharts from the Z specification, one must refine the infinite number down to a finite number. Simulations are then performed to verify the Statecharts have no nondeterministic state/activity transitions (i.e., inconsistencies). After checking for inconsistencies, in a second step, all data and transition-conditions are specified (i.e., added in). Simulations are performed again to verify the second step. In this second step, some function/data items improperly defined in Z were discovered. These items agreed in ranges and types in both Z and Statecharts; however, they generated incorrect output during the simulations. This (kind of) information is then carried back to refine the Z schemas.

In a third step, after the simulation, faults are injected into the state/activity charts. This is done by changing state variable values while running a simulation. The output from the

simulation using injected faults is compared with the expected output. The expected output values are obtained based on the formulae given in the SRS. Using fault-injection enables one to evaluate the system's ability to cope with unexpected system failure.

3.1 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 more concrete specifications. In combination with natural language, it can be used to produce a formal specification [20].

An axiom is a common way of defining global objects in Z. It consists of two parts: declaration and predicate as shown in Figure 1. The predicate constrains the objects introduced in the declaration. Schemas model system states and are the main structuring mechanism used to create patterns, objects, and operations. A schema consists of two parts (Figure 1): a declaration of variables; and a predicate constraining their values. The name of a schema is optional; however, for compositional purpose, it is convenient to give a name that can be referred to other schemas. These facilities are useful and essential in clarifying ambiguities and solidifying one's understanding of the requirements.

Figure 1. Forms of an axiom and a schema

3.2 Statecharts

Statecharts constitute a state-based formal diagrammatic language that provide a visual formalism for describing states and transitions in a modular fashion, enabling cluster orthogonality (i.e., concurrency) and refinement, and supporting the capability to move between different 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 [21].

Statecharts 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 [22]. The State/Activity charts are used to specify conceptual system models for symbolic simulation. Using these facilities, assumptions were verified, faults were injected, and hidden errors were identified that represent inconsistencies or incompleteness in the specification.

A GCS project was created within the Statemate environment. Graphical editors were used to create State/Activity charts. Once the graphical forms were characterized, state transition conditions and data items were defined within the "data dictionary" of the project. The Activity chart and Statecharts reflect all variables/conditions defined in the Z formulation. During simulation, we observed the sequence of state changes that occur to validate the system against its specified structure (based on Schema declarations) and constraints (based on Schema predicates). Initial (and current) values and conditions were changed while rerunning and/or resuming the simulation in the process of verifying consistency and completeness against the Statecharts specification.

3.3 Specification Tests

The Statecharts model is examined in two different ways. First, the State/Activity charts are tested as finite state machines (ensuring state transition conditions and activity triggers are deterministic). Next, their functionality is tested. The actual outputs (values generated by the State/Activity charts simulations) are compared with the expected output.

3.3.1 Finite State Machine Approach

This approach assures against absorbing States/Activities as well as nondeterministic State/Activity transitions. Bogdanv and Holcombe have discussed how to test Statecharts for

an aircraft control system [23] by examining the underlying finite state machine(s). We extend their method to evaluate if the Statecharts are behaviorally equivalent to the SRS. In other words, every activity and state transition is exercised as described in the SRS.

3.3.2 Data Item Approach

In the data item approach, the state/activity charts are treated like a software program (i.e., black-box testing). Test cases are generated to evaluate if the Statecharts model produces the correct data outputs. Input and expected output values are determined based on the information from the data dictionary and according to the SRS/Z schemas. This test assures that there are no inconsistent or unspecified data driven operations.

3.4 Fault Injection

Fault injection is used to observe how the software system behaves under experimentally-controlled anomalous circumstances. Voas et al., claim that system anomalies are caused by either faulty code or corrupted input, or some combination of both [24]. Their approach injects faulty codes into the software and then observes the software behavior. Conversely, we injected faults (i.e., by altering the correct values of system variables) into an executable *model* of the SRS while it is being run (i.e., at various stages of execution).

After injecting faults into the model, we observe the behavior to detect if there are any incorrect state transitions and/or outputs. The choice of test cases is based on a functional analysis of the submodules. Submodules are evaluated to determine if they could cause a critical failure. Fault injection is not performed on non-critical submodules. In this way, the SRS is evaluated for the fault-tolerance.

4 APPLICATION EXAMPLE

In this section, an example is presented to explain how to apply the method described above[25, 26]. This section shows one small part (i.e., ARSP submodule) of the larger NL-based GCS SRS that was transformed.

The selected Altitude Radar Sensor Processing (ARSP) submodule specification shows inputs, outputs, and subsystem processing descriptions. The SRS provides a data dictionary with variable definitions, type, and units, and a brief description of variables and functions.

This module specification was abstracted into Z, preserving the variable names, operations (i.e., functionality), dependency and scope. Figure 2 provides an example using the FRAME_COUNTER input variable that illustrates the complete translation from the SRS to Z and Statecharts. The top box in the Figure 2 represents the SRS. The box in the middle of the Figure 2 represents the Z Specification while the bottom box shows a part of the Statecharts model. In the 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 declaration part, and the range of the variable is represented in the predicate part (lower half of the schema). The Statecharts 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.

In translating from the SRS to Z, four different ambiguous requirements were identified. The first ambiguity committed leaves the rotational direction (i.e., left/right array shifting) because it only specifies "rotate." Second, an undefined third order polynomial was revealed that is used to estimate the AR_ALTITUDE value. The third 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 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 assumes that the AR_COUNTER is updated outside of the module. Both scenarios were constructed separately and compared to understand how Z could be useful in clarifying ambiguity and avoiding conflicts.

In this first scenario (Scenario One) to properly update AR_COUNTER within the ARSP, the two different purposes of the variable should be separated. Accordingly, the Z specification of the ARSP was defined to account for two separate variables (AR_COUNTER and Echo). This implies that the AR_COUNTER represents only the pulse counter while Echo represents whether the radar echo pulse is received on time. The Z specification is consistent with the SRS as long as the newly introduced variable (Echo) does not cause a side

effect. The Echo was treated as an additional ARSP input. Obtaining the input value requires changes of the SRS outside of the ARSP module. This in turn requires the specification to be revised to satisfy the data decoupling principle [27]. Therefore, the interpretation of Scenario One is inconsistent with the SRS.

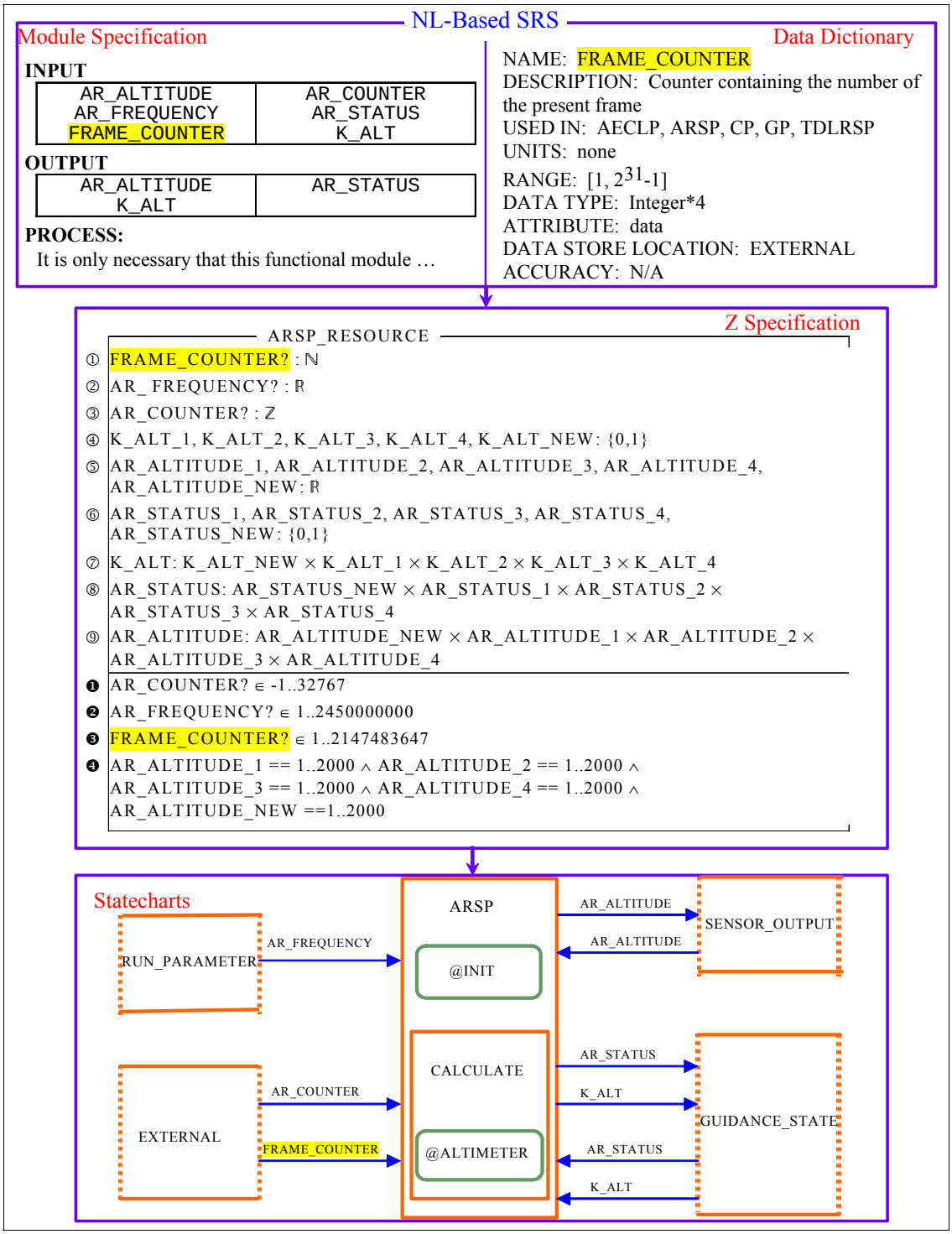

Figure 2. Translation example from NL-based to Statecharts

Conversely, in Scenario Two (details described in Section 4.1) no additional variables were defined. Only the variables defined in the SRS were modeled (as well as covering the required ARSP behaviors). Therefore, this reformulation of the SRS in Z was considered complete and consistent. The Statecharts were developed based on Scenario Two.

4.1 Z Specification

Scenario Two is described here. This scenario assumes that the AR_COUNTER value is updated outside of the ARSP module (i.e., 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 3) defines the ARSP module input and output variables. The FRAME_COUNTER? (Signature [Sig] ①) is an input variable giving the present frame number and its type is a natural number. AR_FREQUENCY? (Sig②) represents the rate at which the AR_COUNTER? is incremented and its type is a real number. The AR_COUNTER? (Sig③) 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 (Sig④) variables are defined as sets of binary elements. The K_ALT value is updated in the ARSP to be used in the Guidance Processing (GP) module to determine the correction term value of GP_ALTITUDE variable. The AR_ALTITUDE_1, AR_ALTITUDE_2, AR_ALTITUDE_3, AR_ALTITUDE_4, and AR_ALTITUDE_NEW (Sig⑤) 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 (Sig⑥) are defined as binary values that represent the health status for various elements of the altimeter radar. The AR_STATUS, AR_ALTITUDE, and K_ALT (Sigs⑦-⑨) arrays hold the previous 4 values and the current value of their elements respectively.

These 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 though the sequence. The predicates ❶, ❷, and ❸ represent the variables ranges. The predicate ❹ restricts the values for the sets in the Signature ⑤.

ARSP_RESOURCE

① FRAME_COUNTER? : $\mathbb{N}$
② AR_ FREQUENCY? : $\mathbb{R}$
③ AR_COUNTER? : $\mathbb{Z}$
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: $\mathbb{R}$
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

---

❶ AR_COUNTER? ∈ -1..32767
❷ AR_FREQUENCY? ∈ 1..2450000000
❸ FRAME_COUNTER? ∈ 1..2147483647
❹ 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

Figure 3. ARSP_RESOURCE schema

The ARSP schema (Figure 4) is the main functional schema of the ARSP module. The ARSP_RESOURCE schema is imported (and is modified) in the Sig①. The Altitude_Polynomial function (Sig②) 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 (Sig③), K_ALT_Update (Sig④), and AR_ALTITUDE_Update (Sig⑤) 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 in the predicates to determine if the FRAME_COUNTER? is odd or even.

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 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.

ARSP

① Δ ARSP_RESOURCE
② Altitude_Polynomial: AR_ALTITUDE $\rightarrow \mathbb{R}$
③ AR_STATUS_Update: AR_STATUS_NEW $\times$ AR_STATUS $\rightarrow$ AR_STATUS
④ K_ALT_Update: K_ALT_NEW $\times$ K_ALT $\rightarrow$ K_ALT
⑤ AR_ALTITUDE_Update: AR_ALTITUDE_NEW $\times$ AR_ALTITUDE $\rightarrow$ AR_ALTITUDE

❶ FRAME_COUNTER? mod 2 = 0 $\Leftrightarrow$
AR_ALTITUDE' = AR_ALTITUDE_Update (AR_ALTITUDE_NEW, AR_ALTITUDE) $\wedge$
AR_STATUS' = AR_STATUS_Update (AR_STATUS_NEW, AR_STATUS) $\wedge$ K_ALT' = K_ALT_Update (K_ALT_NEW, K_ALT)

❷ FRAME_COUNTER? mod 2 = 1 $\wedge$ AR_COUNTER $\geqslant$ 0 $\Leftrightarrow$ AR_ALTITUDE'= AR_ALTITUDE_Update ({AR_COUNTER? $*$ 300000000 div AR_FREQUENCY div 2}, AR_ALTITUDE)

❸ FRAME_COUNTER? mod 2 = 1 $\wedge$ AR_COUNTER = -1 $\wedge$ AR_STATUS = (_, 0, 0, 0, 0) $\Leftrightarrow$
AR_ALTITUDE' =
AR_ALTITUDE_Update ({Altitude_Polynomial AR_ALTITUDE}, AR_ALTITUDE)

❹ FRAME_COUNTER? mod 2 = 1 $\wedge$ AR_COUNTER = -1 $\wedge$ AR_STATUS $\neq$ (_, 0, 0, 0, 0) $\Leftrightarrow$
AR_ALTITUDE' = AR_ALTITUDE_Update (AR_ALTITUDE_1, AR_ALTITUDE)

❺ FRAME_COUNTER? mod 2 = 1 $\wedge$ AR_COUNTER $\geqslant$ 0 $\Leftrightarrow$ AR_STATUS' = AR_STATUS_Update(0, AR_STATUS) $\wedge$ K_ALT' = K_ALT_Update(1, K_ALT)

❻ FRAME_COUNTER? mod 2 = 1 $\wedge$ AR_COUNTER = -1 $\wedge$ AR_STATUS = (_, 0, 0, 0, 0) $\Leftrightarrow$
AR_STATUS' = AR_STATUS_Update(1, AR_STATUS) $\wedge$
K_ALT' = K_ALT_Update(1, K_ALT)

❼ FRAME_COUNTER? mod 2 = 1 $\wedge$ AR_COUNTER = -1 $\wedge$ AR_STATUS $\neq$ (_, 0, 0, 0, 0) $\Leftrightarrow$
AR_STATUS' = AR_STATUS_Update(1, AR_STATUS) $\wedge$
K_ALT' = K_ALT_Update(0, K_ALT)

Figure 4. ARSP schema

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.

4.2 Statecharts

The state/activity charts, derived from the Z specification are described in here. The ARSP Activity-chart (Figure 5) shows the data flow between the data stores (dotted line boxes) and the ARSP module. The data flow directions reflect what is specified in the data dictionary of the SRS. The "@INIT" control state in the ARSP activity chart represents the link to the INIT Statechart (Figure 6). Each activity is allowed to have only one control state. The control state can be a superstate or an AND/OR decomposed state.

INIT Statechart (Figure 6) shows the initialization of the ARSP module and a portion of the ARSP operational schema (Figure 4). The default transition activates the CURRENT_STATE when the ARSP activity (in the ARSP activity chart) is begun. The transition from the CURRENT_STATE state to KEEP_PREVIOUS_VALUE state describes predicate ❶ of Figure 4. 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, described by the statement "FRAME_COUNTER mod 2 = 1" in Figure 4.

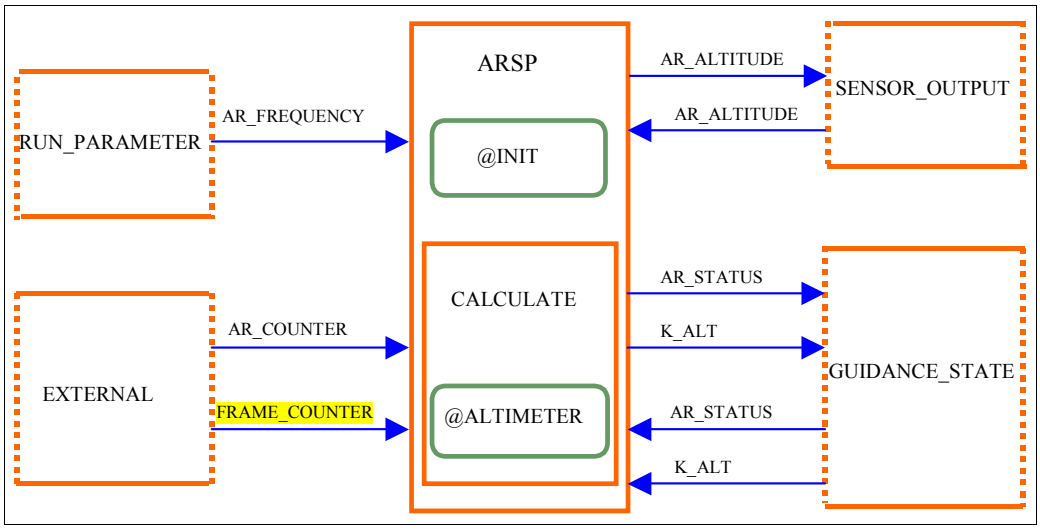

Figure 5. ARSP activity-chart

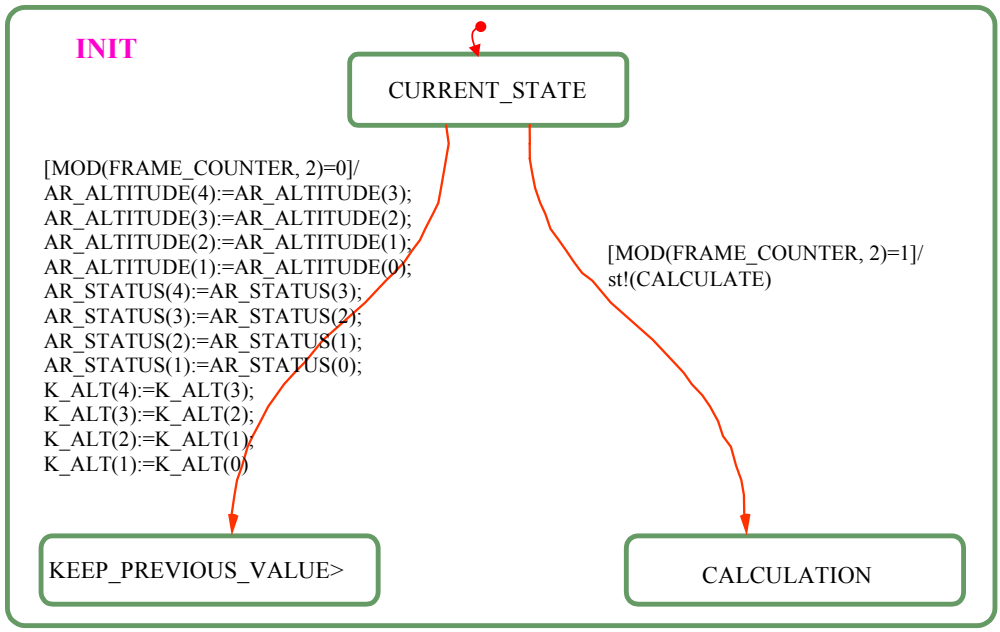

Figure 6. INIT statechart

The Altimeter Statechart (Figure 7) is represented by the "@ALTIMETER" control activity in the ARSP activity chart (Figure 5). The ODD state is activated by the default transition when the CALCULATION activity (in 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 array are set to "healthy." When this transition begins the AR_STATUS and K_ALT values are updated as described by predicate ❻ of Figure 4. The 0 (zero) value of the AR_STATUS means "healthy" which corresponds to the value given in the SRS data dictionary [2].

The transition from the ODD state to the CALCULATE_ALTITUDE state begins when the AR_COUNTER is positive, which is equivalent to predicate ❺ of Figure 4. 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 ❼ in Figure 4. 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

❷ (Figure 4). The transaction from the KEEP_PREVIOUS state to the DONE state denotes the predicate ❹ (Figure 4) operation.

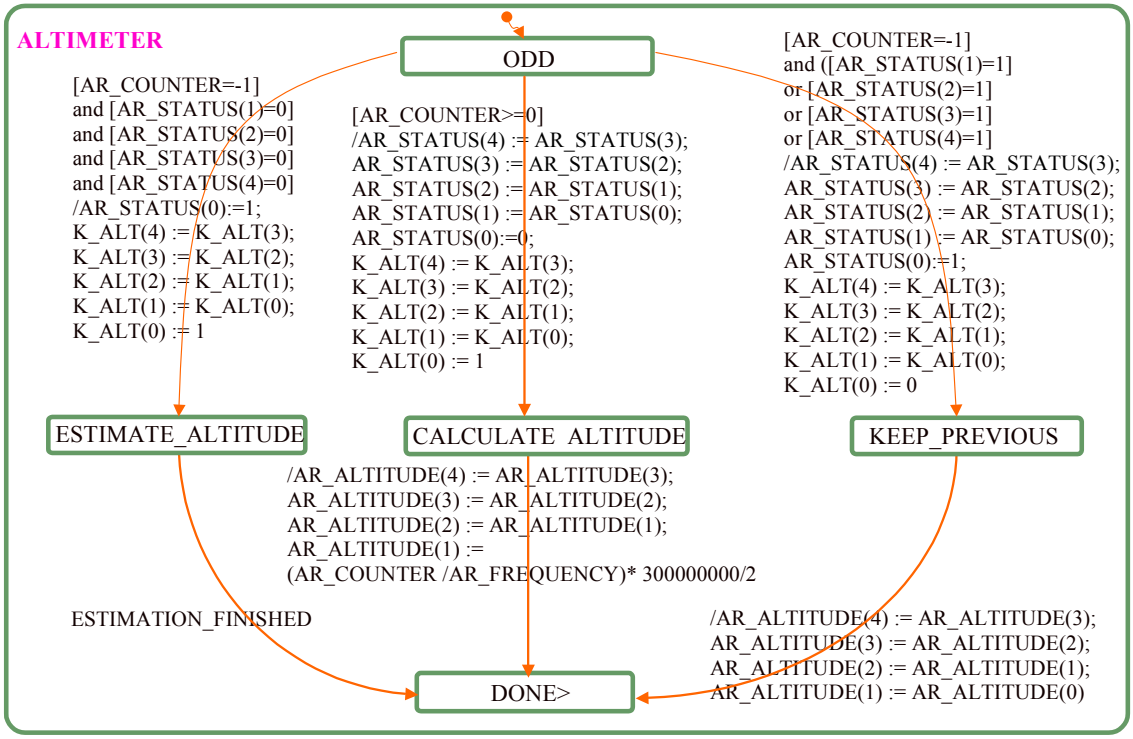

Figure 7. ALTIMETER statechart

4.3 Specification Tests

The statechart models described here and in the appendix are validated for completeness and consistency using symbolic simulation. Two specification test results (based on approaches described in Section 3.3) are presented in this section.

4.3.1 Finite State machine approach

There are four possible paths for activity/state transitions in the ARSP Statecharts model. *Path 1* represents the ARSP module's processing when the `FRAME_COUNTER` is even. *Path 2* represents the condition when the updated `FRAME_COUNTER` is an odd number, the radar echo pulse is *not* yet received, and all the `AR_STATUS` elements' values are healthy. *Path 3* is taken when the updated `FRAME_COUNTER` is an odd value, the radar echo pulse has been received, and all the `AR_STATUS` elements' values are healthy. *Path 4* describes the condition when the updated `FRAME_COUNTER` value is odd, the echo has not arrived, and one or more of the `AR_STATUS` elements' values are not healthy.

The simulation results in Table 2 show the order of the activities/states entered for each path. One can conclude that the ARSP Statecharts model does not have any absorbing states or activities and the module is complete indicating that the SRS is complete (at least for the ARSP submodule).

Table 2. 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.

4.3.2 Data Item approach

Five test cases (Case 1-5) as shown in Table 3 were defined to test the Statecharts. They represent the way the Z schemas were visualized and evaluated. The input and output values were calculated based on the equations in the SRS. The AR_FREQUENCY variable is used to determine the AR_ALTITUDE value (represented as a state transition from the "CALCULATE_ALTITUDE" state to the "DONE>" state in Figure 7). The AR_FREQUENCY variable is defined as a real number with a large range. Consequently, it 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 1.5e9 for all test cases. Table 3 and 4 show how each of the conditions was evaluated and this should help to convince the reader that the ARSP subunit (one of six different sensor units which make up the complete GCS platform) is significantly complex.

The values of the ARSP input/output variables are given in Table 3. The contents of Table 4 represent the highlighted column of Table 3 in detail. In 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 value 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 depend on the module inputs and their value before the execution. The expected values of the output variables are determined prior to the simulation. The "After the execution" values (shown in Table 4) represent the actual outputs from the Statecharts model simulation. The test results are correct when the expected values and the after execution values match. The actual output values for all the test cases match the expected output values (as shown in Table 3). Therefore, the result of this simulation shows that the Z specification was developed correctly.

Table 3. 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 4. 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.

4.4 Fault Injection

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 environment.) For example, one can alter a system state variable (e.g., FRAME_COUNTER) at a certain state (e.g., CURRENT_STATE) during the simulation for Case 1. Table 5 gives the fault injection results of the FRAME_COUNTER alteration at CURRENT_STATE. The expected values of the output variables are not the same as the actual values of the output due to the state variable change (depicted as the highlighted **x** mark in Table 6).

Table 5. 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 6. 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.

Table 6 shows 120 fault injection results. The "CURRENT_STATE" does not tolerate any of the injected faults. In addition, fault injection in the CALCULATION and ODD system states produces erroneous outputs. Therefore, one can conclude that these three system states are the most vulnerable.

The Statecharts approach has a better chance of predicting possible faults in the system. The Z specification cannot provide a way to predict the transitions from state to state. (i.e., Z is not executable.) Three new issues arose during the fault injection process. (1) Some correct inputs produced incorrect outputs; (2) Some weak points were found where faults were

hidden (e.g., errors described in Appendix C in [26]); (3) During the execution of the model, some errors such as memory overflow were uncovered. Finding the correct formulation is a process of refinement and validation, which was facilitated using this approach.

4.5 Reformulated Requirements

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 SRS were revealed during the construction of Z schemas. When a misinterpreted specification in Z was uncovered during the execution of the Statecharts model, Z specification was refined using the test results.

Based on the simulation results using fault injection, the SRS was discovered to be 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: $1 \leqslant AR_FREQUENCY \leqslant AR_COUNTER * 75000$, or $AR_COUNTER = -1 \vee (0 \leqslant AR_COUNTER \leqslant AR_FREQUENCY/75000)$. In other words, one of these two relational expressions must evaluate as true.

5 CONCLUSION

This paper discusses the methods and results as they relate to the ARSP submodule which was part of a larger system specification. The actual study covered additional submodules as well as the overall structure of the GCS. The other submodules specified and analyzed include the ARSP (Altimeter Radar Sensor Processing), GP (Guidance Processing), RECLP (Roll Engine Control Law Processing), and CP (Communication Processing). The choice of these submodules for specification was made based on the GCS run-time schedule.

Z was used first to detect and remove ambiguity from this portion of the 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 executable specification to probe how the system would perform. Missing or incorrectly specified requirements were found during the process. The integrity of the SRS was assessed in this

manner. This approach can help avoid the problems that result when incorrectly specified artifacts (i.e., in this case requirements) force corrective work.

The results of this analysis indicated that the portion of the GCS SRS under study was inconsistent, incomplete and not completely fault-tolerant. The findings indicate that one can better understand the implications of the system requirements using this approach (Z-Statecharts) as the basis for their specification and analysis. The time involved generating the Z specification (considering all variables and functional specifications) is a major concern. Naturally, the amount of time necessary for generating a formalization of a NL-based specification will vary based on the inherent complexity of the SRS.

In the long run we envision this approach will be useful in a more general sense as a means to avoid incompleteness and inconsistencies. Undoubtedly, the dynamic behavioral analysis is useful in avoiding major design flaws. Refinement between these two formalisms gives pertinent analysis of the problem – i.e., operational errors between states, functional defects, lack of such properties such as fault tolerance, etc.

6 REFERENCES

1. Kotonya G, Sommerville I. Requirements Engineering: Process and Techniques. Wiley, 1998.
2. NASA. Software Requirements - Guidance and Control Software Development Specification Ver 2.2 with the formal mods 1-8. NASA, Langley Research Ctr 1993.
3. Shaw AC. Real-Time Systems and Software. Wiley, 2001.
4. Leveson N. Safeware - system safety and computers. Addison Wesley, 1995.
5. Czerny B. Integrative Analysis of State-Based Requirements for Completeness and Consistency. PhD dissertation in Computer Science, Michigan State University. 1998.
6. Pradhan DK. Fault-Tolerant Computer System Design. Prentice Hall. 1996. pp 428-477.
7. Gaudel M-C, Bernot G. The Role of Formal Specifications. In: Astesiano E, et al. (eds). IFIP State-of-the-Art Report: Algebraic Foundations of Sys Spec. Springer. 1999.
8. Vliet HV. Software Engineering: Principles and Practice. Wiley, 2000.
9. Sannella D, Tarlecki A. Algebraic Preliminaries. In: Astesiano E, et al. (eds). IFIP State-

of-the-Art Reports: Algebraic Foundations of Sys Spec. Springer. 1999.

10. Fabbrini F, et al. An Automatic Quality Evaluation for Natural Language Requirements. 7th Int. Workshop on Req. Eng.: Foundation for SW Quality (REFSQ) 2001. Accessed from: www.ifi.uib.no/conf/refsq2001/papers/p3.pdf. Accessed on: Mar. 25, 2002.
11. Heitmeyer C, et al. Using Abstraction and Model Checking to Detect Safety Violations in Requirements Specification. IEEE Trans on SW Eng 1998; 24 (11): 927-948.
12. Heimdahl MPE, Leveson NG. Completeness and consistency in Hierarchical State-Based Requirements. IEEE Trans on SE 1996; 22 (6): 363-377.
13. He X. PZ nets - a formal method integrating Petri nets with Z. Info and SW Tech 2001; 43: 1-18.
14. Hierons RM, Sadeghipour S, Singh H. Testing a system specified using Statecharts and Z. Info and SW Tech 2001; 43 (2001): 137-149.
15. Bussow R, Weber M. A Steam-boiler Control Specification with Statecharts and Z. LNCS 1165. 1996.
16. Grieskamp W, Heisel M, Dorr H. Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components. LNCS 1382. 1998.
17. Damm W, et al. Statecharts - Using Graphical Specification Languages and Symbolic Model checking in the Verification of a Production Cell. LNCS 891. 1995.
18. Bussow R, Geisler R, Klar M. Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study. LNCS 1382. 1998.
19. Castello R. From Informal Specification to Formalization: an Automated Visualization Approach. PhD dissertation in Computer Science, University of Texas at Dallas. 2000.
20. Woodcock J, Davies J. Using Z: Specification, Refinement, and Proof. Series of Comp Sci. Prentice Hall Int, 1996.
21. Harel D. Statecharts: A Visual Formalism for Complex Systems. Sci of Comp Prog 1987; 8: 231-274.
22. Harel D, Politi M. Modeling Reactive Systems with Statecharts. McGraw-Hill, 1998.
23. Bogdanov K, Holcombe M. Statechart testing method for aircraft control systems. Software Testing, Verification & Reliability 2001; 11 (1): 39-54.

24. Voas J, et al. A Crystal Ball for Software Liability. IEEE Computer 1997; 30 (6): 29-36.
25. Sheldon FT, Kim HY, Zhou Z. A Case Study: Validation of the Guidance Control Software Requirements for Completeness, Consistency, and Fault Tolerance. Proc of IEEE 2001 Pacific Rim Intl Symp on Dependable Comp. 2001. Seoul, Korea. IEEE Computer Society. pp 311-318.
26. Sheldon FT, Kim HY. Validation of Guidance Control Software Requirements Specification for Reliability and Fault-Tolerance. Proc of Annual Reliability and Maintainability Symp. 2002. Seattle, WA. USA. IEEE. pp 312-318.
27. Sommerville I. Software Engineering. 6th edn. Addison-Wesley, Reading, MA 2000.

APPENDIX[5]

[5] The appendix gives the total system architecture. The statecharts are included as background material. We do not plan to include this appendix in the final version. The final proved Z schemas can be obtained from http://www.eecs.wsu.edu/seds/hkim_thesis_final_ilogix.pdf or the http://www.ilogix.com university page.

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.

Figure A-1 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 A-1). 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 A-2).

The choice of parts for this study is made based on its run-time schedule (Table A-1). The GCS has a predetermined running time frame that consists of three subframes. Each subframe has specific submodules to run. The partial specification in this study 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 must run last.

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 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 relay to Earth. 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.

Table A-1. Functional unit schedule [2]

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

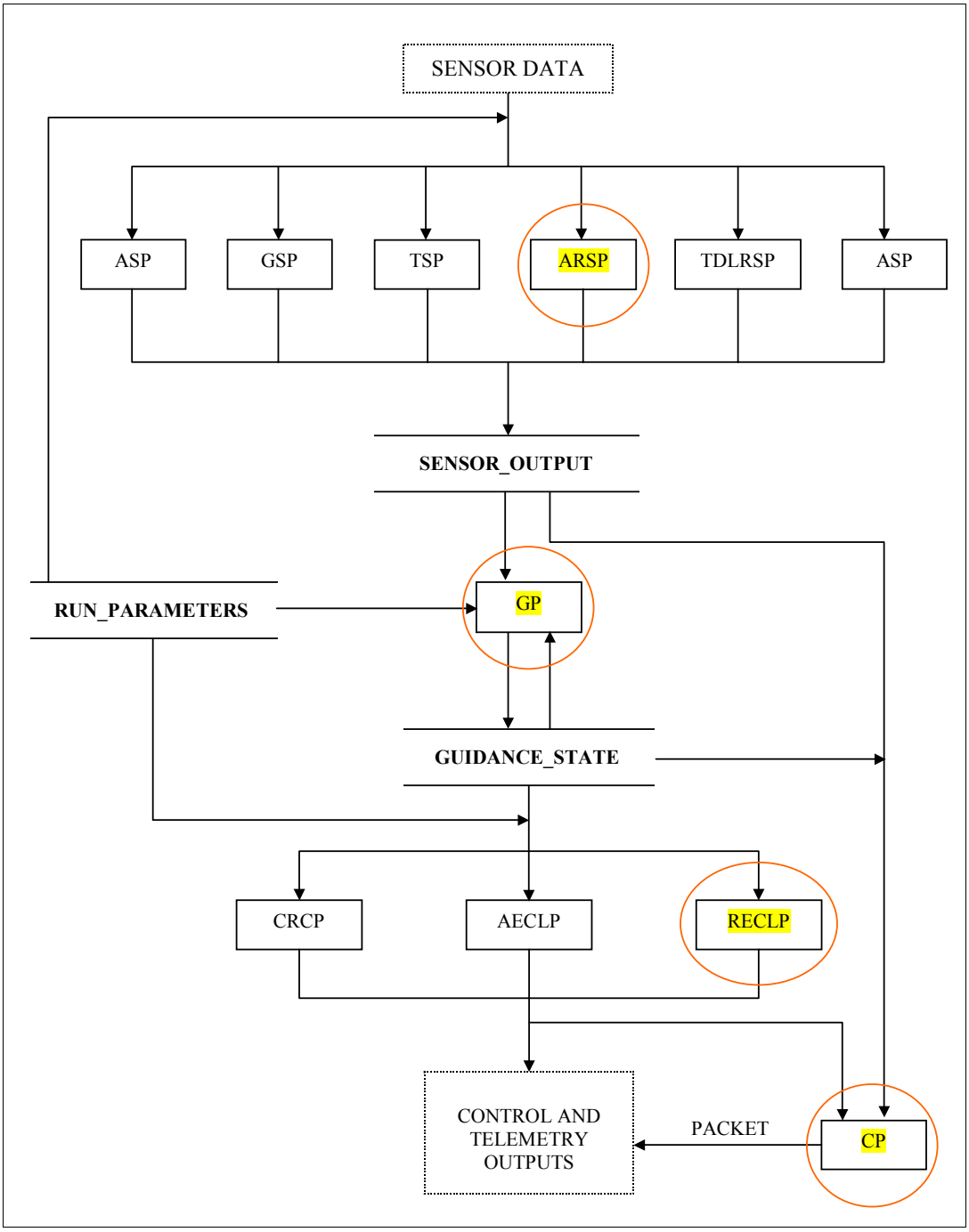

Figure A-1. GCS system structure

The module chart presented in the Figure A-3 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 A-1.

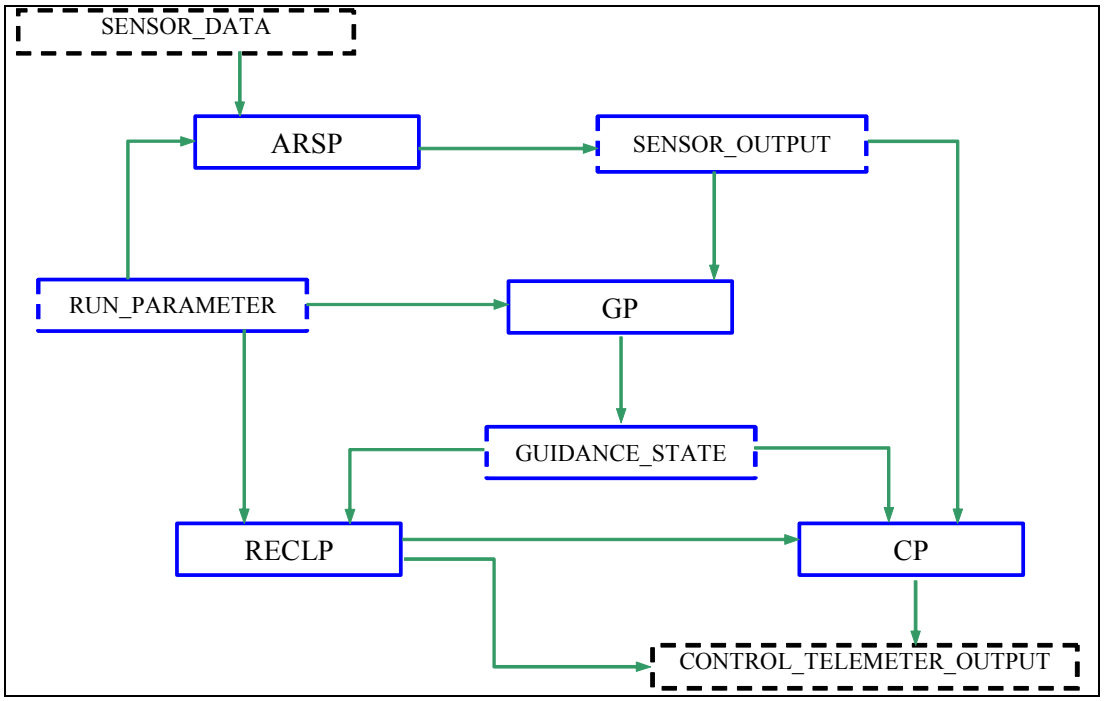

Figure A-2. A module chart of GCS excerpt

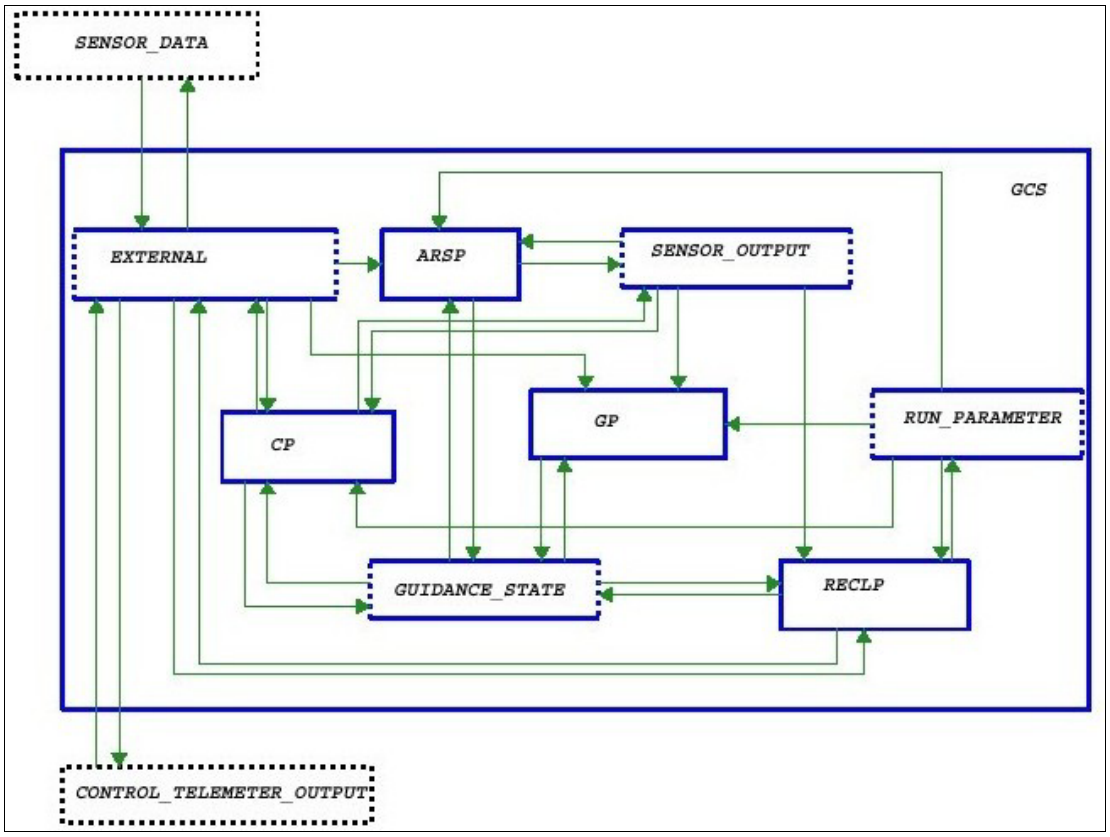

Figure A-3. Actual module chart of the GCS excerpt

In a GCS project created in the Statemate, the GCS activity chart is developed. Figure A-

4 shows the GCS activity chart with four data stores which contains the data definitions. The GCS activity is representing the GCS schemas. 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 having only one control state.

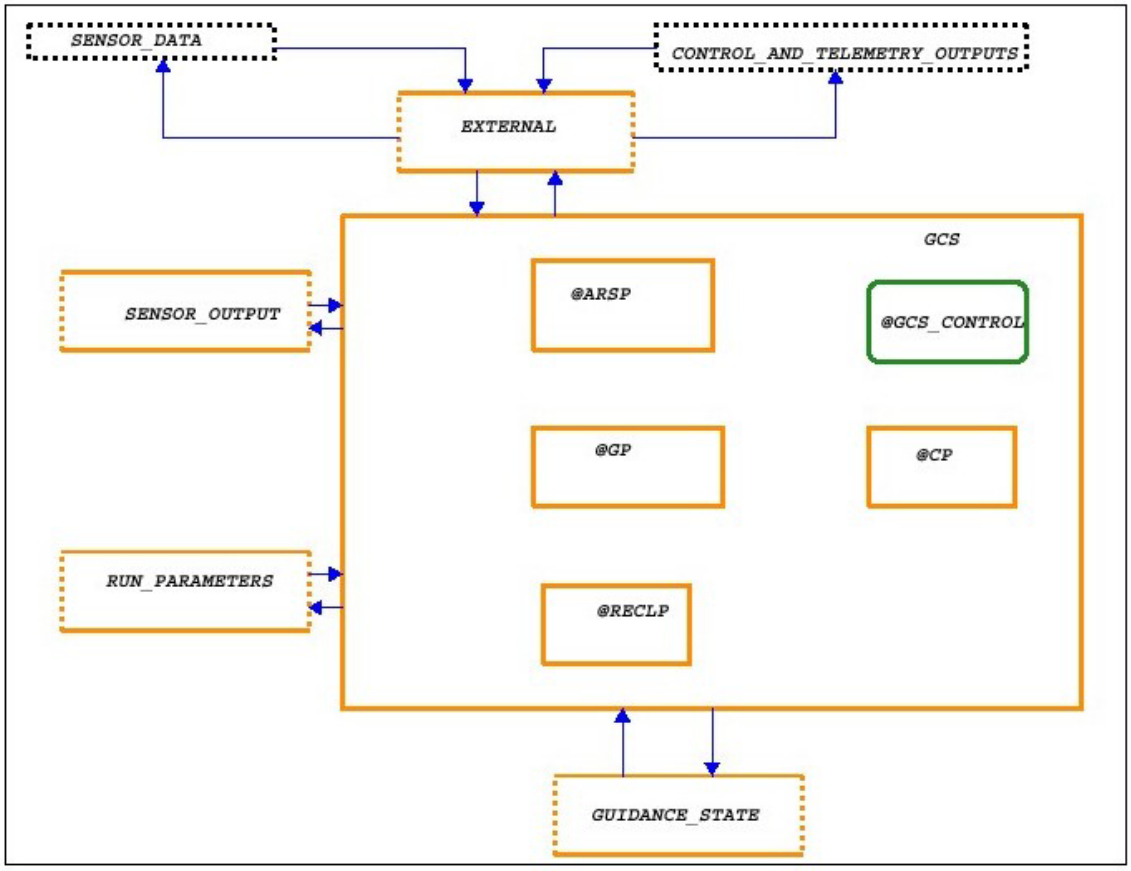

Figure A-4. GCS activity chart

The GCS_CONTROL statechart (Figure A-5) represents the GCS_CONTROL schemas. The default transition represent the moment START_SIGNAL? input for the GCS schema is 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 (Figure A-6 to A-8) linked to the superstate.

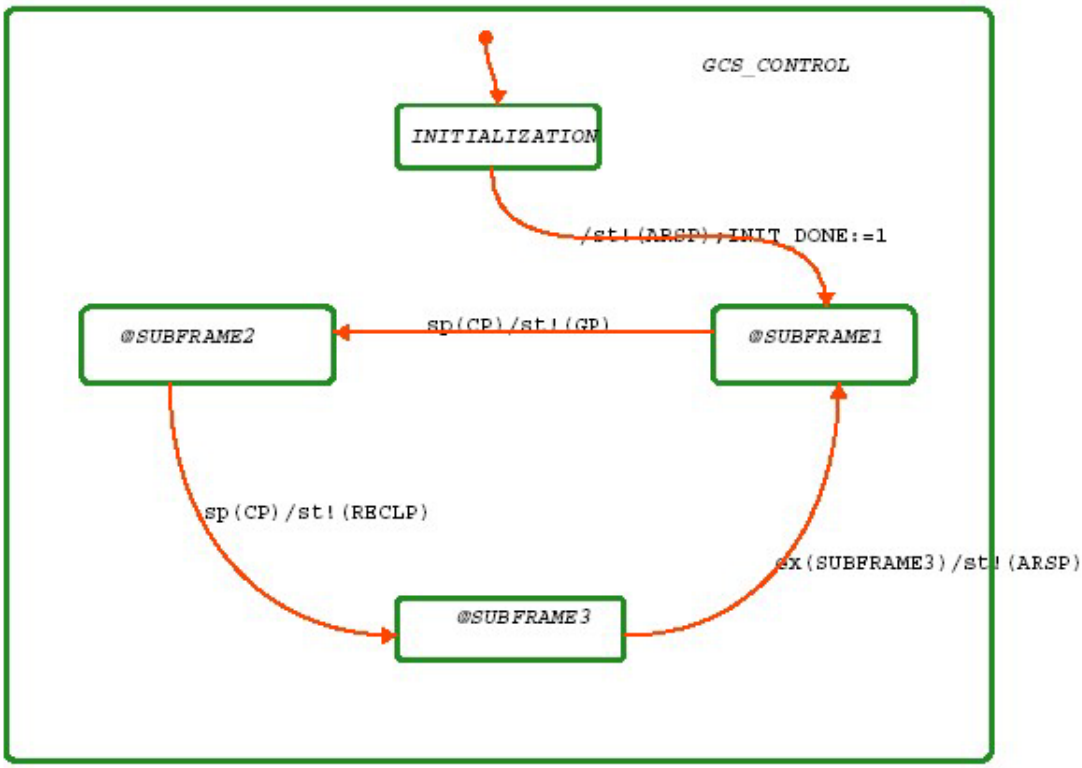

Figure A-5. GCS_CONTROL statechart

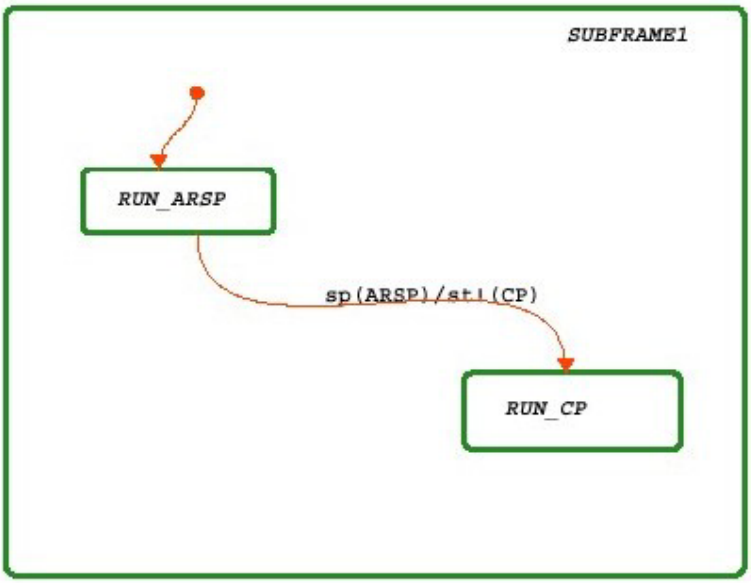

Figure A-6. SUBFRAME1 statechart

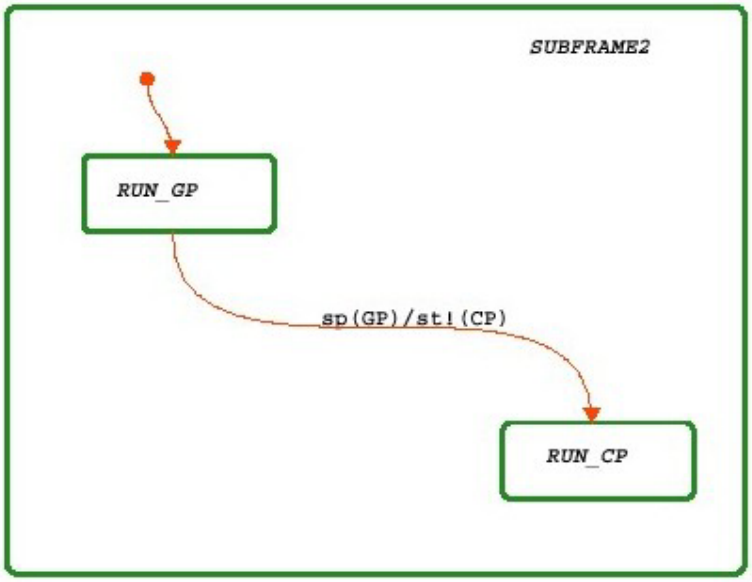

Figure A-7. SUBFRAME2 statechart

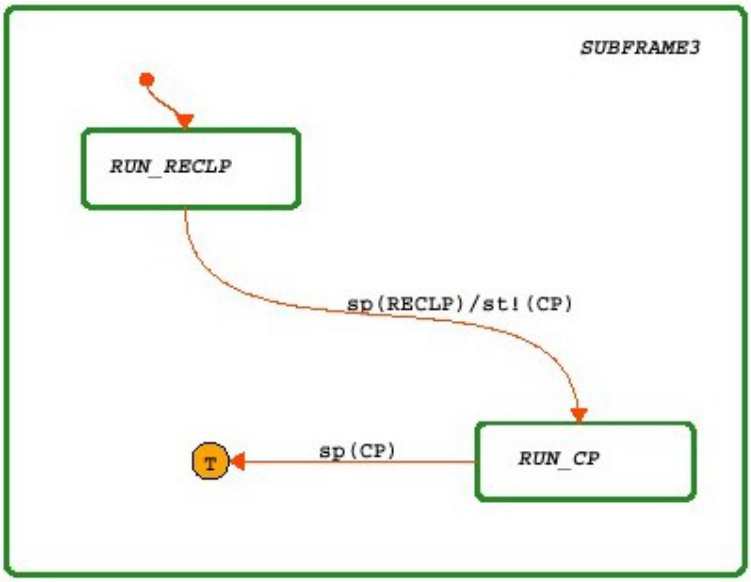

Figure A-8. SUBFRAME3 statechart

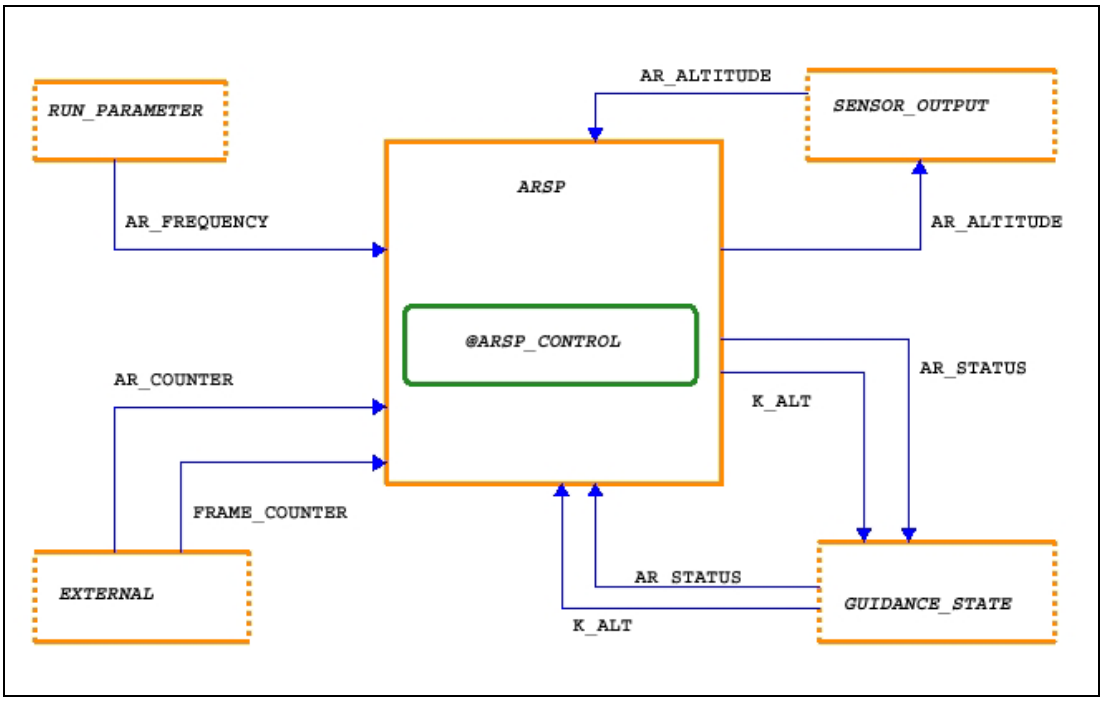

Figure A-9. ARSP activity chart

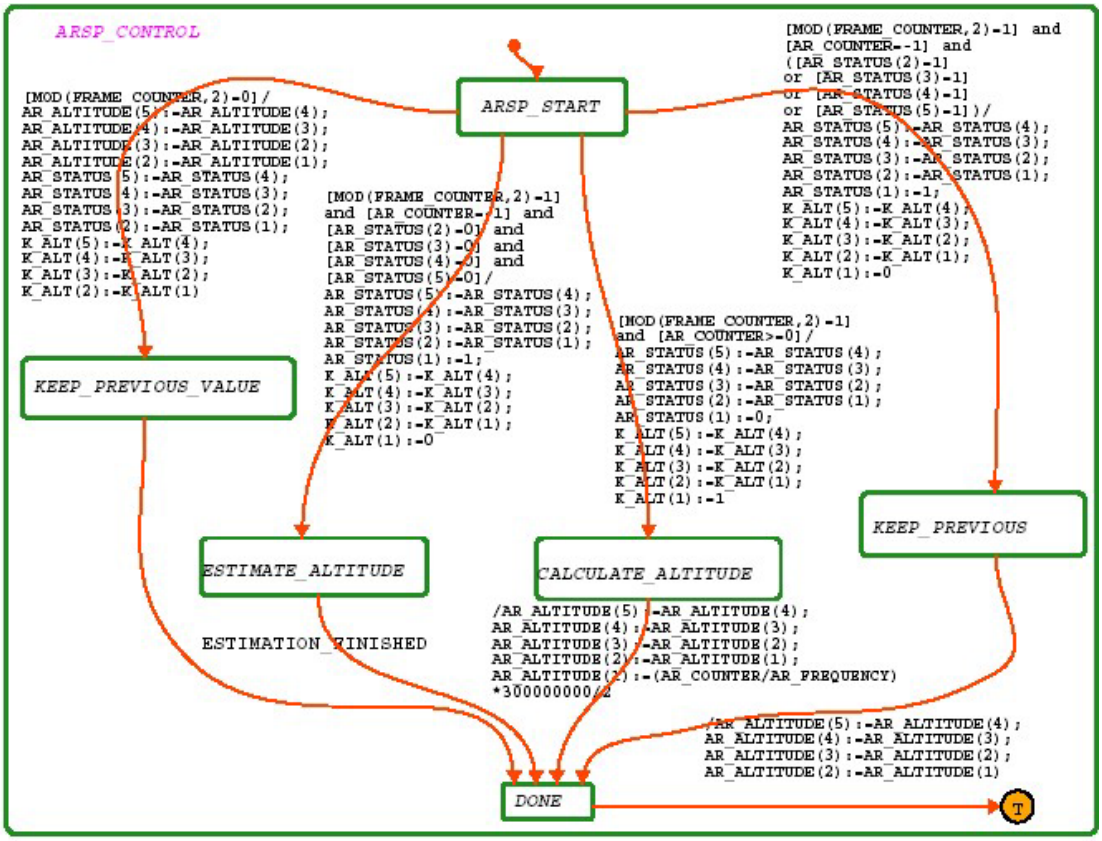

Figure A-10. ARSP_CONTROL state chart

The Figure A-9 and Figure 5 are the equivalent activity charts. Figure A-10 represents the

Z specification of the ARSP submodule shown in Figure 7. The Statecharts model in the Section 4 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 of the thesis (accessible by downloading from http://www.eecs.wsu.edu/seds/hkim_thesis_final_ilogix.pdf).

This ARSP model has 4 distinctive paths. The simulation results of the state transition path are as presented in the Table A-2.

Table A-2. ARSP specification simulation results

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 Section 4.3.2. The fault injection results are described in the Table A-3.

Table A-3. 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
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	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔

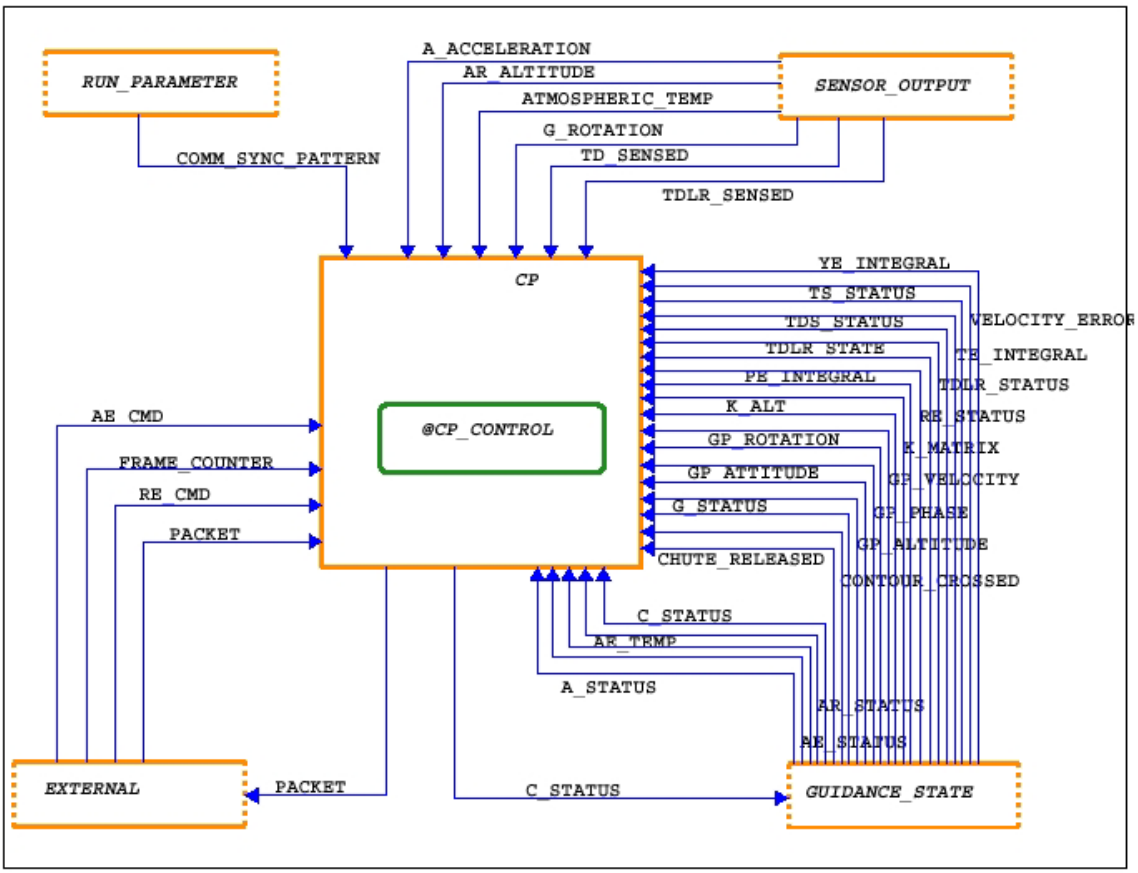

Figure A-11. CP activity chart

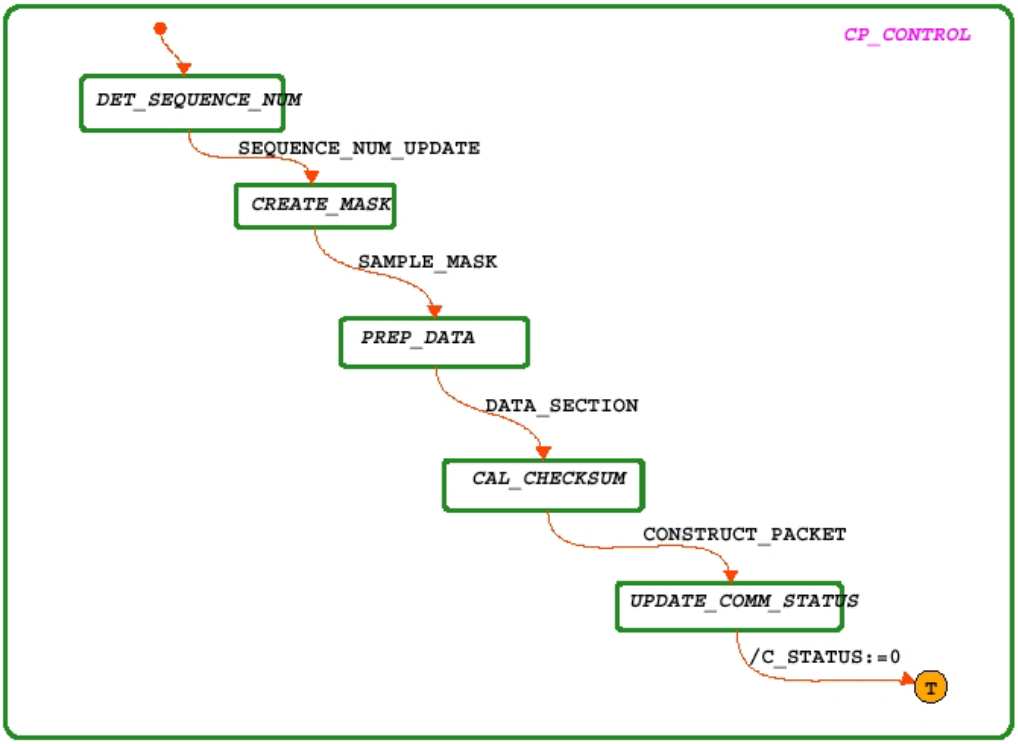

Figure A-12. CP_CONTROL statechart

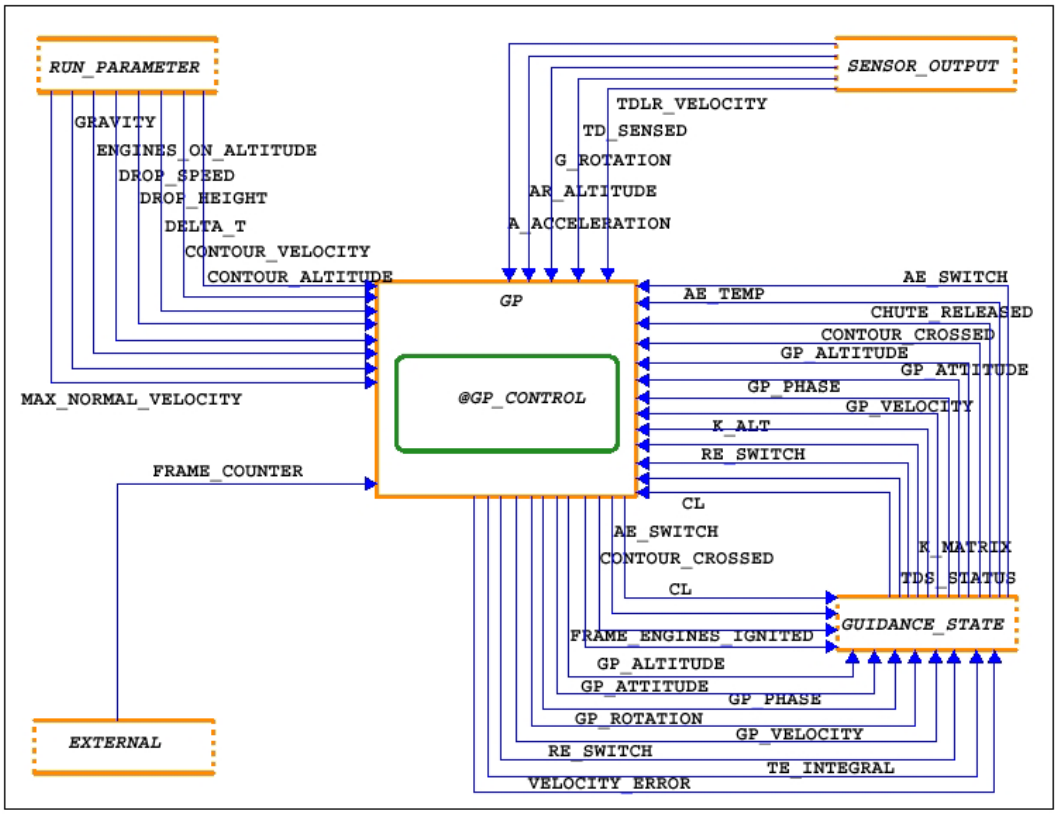

Figure A-13. GP activity chart

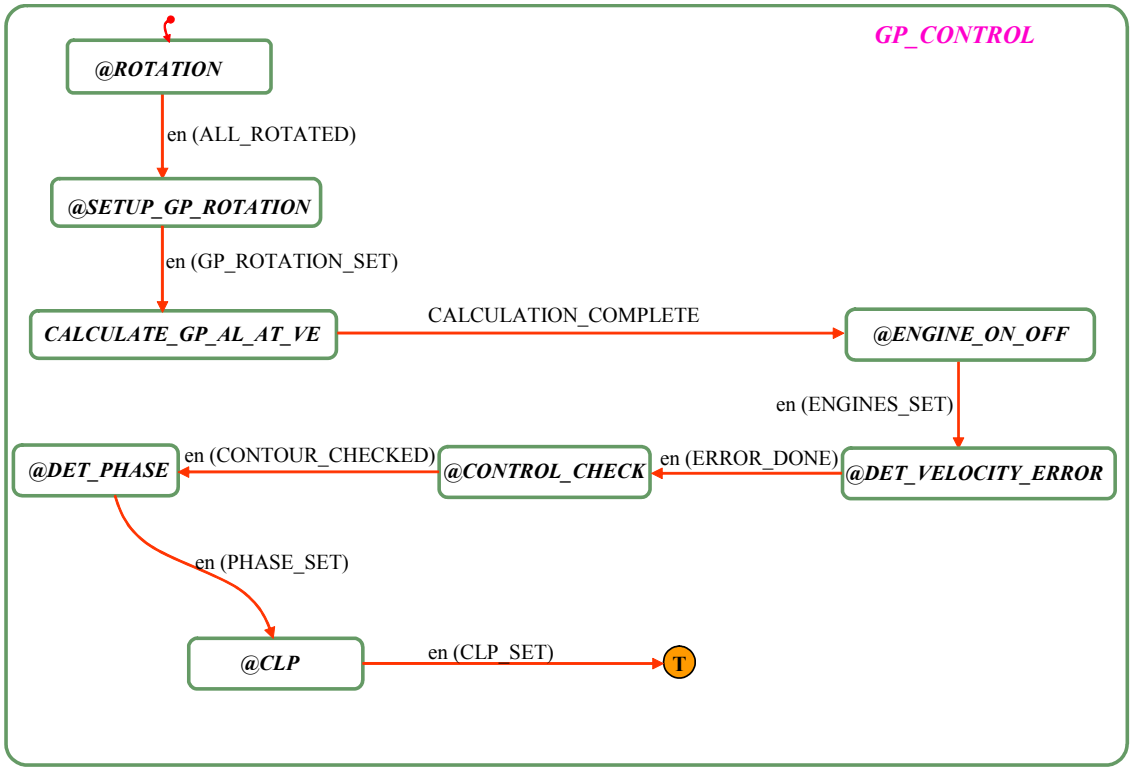

Figure A-14. GP_CONTROL statechart

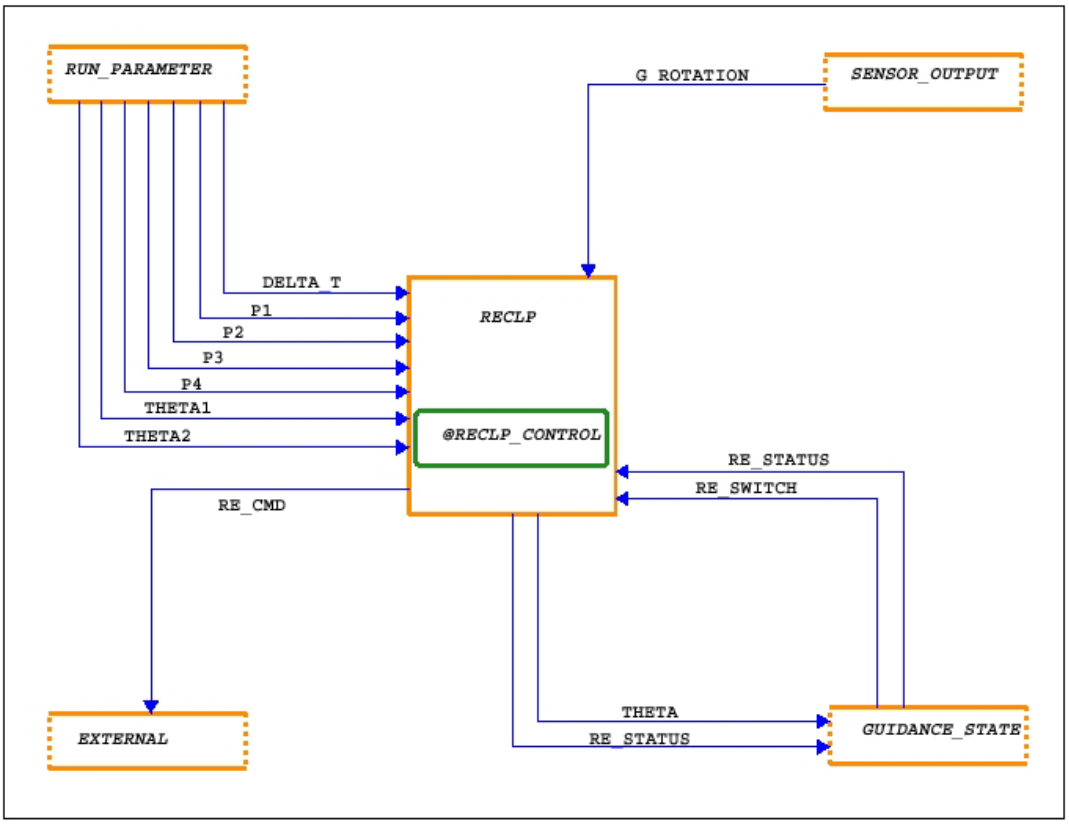

Figure A-15. RECLP activity chart

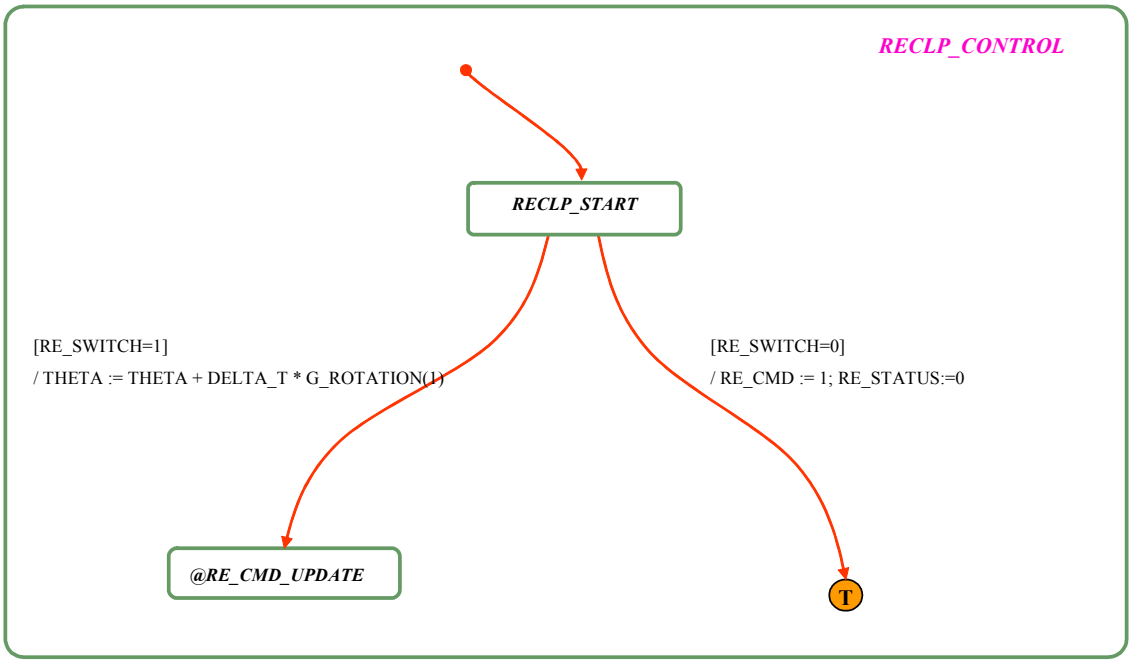

Figure A-16. RECLP_CONTROL statechart

The CP submodule is too inconsistent to develop complete Statecharts model. Moreover,

the bit wise transactions needed 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 model (Figure A-11 and A-12) is built with events that represent the functional sequences that CP is required to follow. The CP has only one state transaction path which is tested using the finite state machine approach (FSMA). The fault injection and data item approach (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 (Figure A-13 and A-14). 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 on the entire GP model while the data item approach test is performed on some parts of GP model that uses only the variables processed inside of the GCS excerpt.

The FSMA and DIA test and Fault injections are performed on the RECLP submodule (Figure A-15 and A-16). The test results are as presented in Table A-4 and A-6 to A-9. Table A-5 shows the system constants for the simulation.

Table A-4. RECLP submodule simulation result

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	E_4	E_4	E_4	E_4	E_4	E_4
RE_CMD_UPDATE	SET_RE_CMD	-	E_5	E_5	E_5	E_5	E_5	E_5	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

Table A-5. Variable values (constants) used for simulation

Variable name	Values
DELTA_T	0.005
P1	0.005
P2	0.010
P3	0.015
P4	0.020
THETA1	0.010
THETA2	0.020

Table A-6. RECLP submodule specification test input and output (1)

	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

Table A-7. RECLP submodule specification test input and output (2)

	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

Table A-8. RECLP submodule specification test input and output (3)

	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

Table A-9. RECLP submodule specification test input and output (4)

	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