

Validation of Guidance Control Software Requirements Specification for Reliability and Fault-Tolerance

Frederick T. Sheldon • Washington State University • Pullman

Hye Yeon Kim • Washington State University • Pullman

Key Words: Requirement Engineering, Verification and Validation, Executable Specifications, Specification Testing, Consistency, Completeness.

SUMMARY & CONCLUSIONS

A case study was performed to validate the integrity of a software requirements specification (SRS) for Guidance Control Software (GCS) in terms of reliability and fault-tolerance. A partial verification of the GCS specification resulted. Two modeling formalisms were used to evaluate the SRS and to determine strategies for avoiding design defects and system failures. Z was applied first to detect and remove ambiguity from a part of the Natural Language based (NL-based) GCS SRS. Next, Statecharts and Activity-charts were constructed to visualize the Z description and make it executable. Using this formalism, the system behavior was assessed under normal and abnormal conditions. Faults were seeded into the model (i.e., an executable specification) to probe how the system would perform. The result of our analysis revealed that it is beneficial to construct a complete and consistent specification using this method (Z-to-Statecharts). We discuss the significance of this approach, compare our work with similar studies, and propose approaches for improving fault tolerance. Our findings indicate that one can better understand the implications of the system requirements using Z-Statecharts approach to facilitate their specification and analysis. Consequently, this approach can help to avoid the problems that result when incorrectly specified artifacts (i.e., in this case requirements) force corrective rework.

1. INTRODUCTION

Highly reliable systems demand rigorously engineered software. A failure in the control software of mission critical systems can be disastrous. It is difficult to create a reliable 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. 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 downstream activities?

The completeness of a specification is defined as a lack of

ambiguity in 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 (Ref. 1). Consistency means that the specification is free from conflicting requirements and undesired nondeterminism (Ref. 2).

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. 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 (Ref. 3). The usefulness of the requirements specification is diminished by not being understandable to the diverse set of stakeholders. Nevertheless, to avoid the confusion caused by ambiguity, we investigated the merits of two different mathematically based RSLs.

Consequently, in this case study Z was used to clarify intentions, identify assumptions and explain correctness in light of ambiguous statements found in the SRS. Statecharts were chosen to model the Z specifications because a key goal was visualization, testability and pre-development evaluation. A clear distinction of our approach as compared to others is that we did not combine Z and Statecharts together. We translated the SRS into Z completely and then translated the Z specification into Statecharts. Next, we evaluate the usefulness of this approach by applying it to a small but critical part of the SRS.

2. RELATED WORKS

There have been numerous studies conducted that combine a Z representation with some formal method for the benefit of visualization and dynamical assessment. Xudong He suggests using a hybrid formal method called PZ-nets that combine

Petri nets and Z notations (Ref. 4). 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 (Ref. 5). 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 poses properties that can be utilized during test generation. These properties help solve the problem of setting up the initial state and checking the final state of a test to assist in test automation. Both the dynamic behavior specified in Statecharts and the individual operations are checked using this method.

Bussow and Weber present a mixed method consisting of Z notations and Statecharts (Ref. 6). 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 Statechart semantics were not fully formalized. In addition, there are several other case studies that utilized Z for defining data while Statecharts were used as a behavioral description method (Ref. 7, 8, 9).

3. THE CHOICE OF METHODS

A 2-step process was performed using Z/Statecharts. First, the NL-based GCS specification was transformed using the Z notation. Z Schemas were abstracted from the GCS SRS. This compositional process helped to clarify ambiguities. Second, the Schemas were transformed into Statecharts/Activity-charts and symbolically executed to assess the model's behavior based on the GCS-specified mission profile.

3.1. Z : a Mathematical Language of Logic, Sets, and Relations

The Z notation is a mathematical language equipped with an underlying theory of refinement 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. Refinement yields a new Z description that provably conforms to its predecessor and is closer to executable code (Ref. 10). Schema's are the main structuring mechanism used to create patterns and objects. The notation is used to model system states. In this work, the state of the system and the relationship between the ARSP and the state of various components were explained. The production of such a specification helps us to understand requirements, clarify intentions, and construct proofs (i.e., identify assumptions and explain correctness). These facilities were useful and essential in clarifying ambiguities and solidifying our understanding of

the requirements.

3.2. Statecharts: State-based Formal Diagrammatic Language

Statecharts constitute a visual formalism for describing states and transitions in a modular fashion, enabling clustering orthogonality (i.e., concurrency) and refinement, and supporting capability for moving between levels of abstraction. Technically speaking, 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 (Ref. 11).

Statecharts (using STATEMATE¹) 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 (Ref. 12). The Activity-charts and Statecharts were used to specify our conceptual system model for symbolic simulation. With the simulation method, we verified our assumptions, injected faults, and identified hidden errors that represent inconsistencies or incompleteness in the specification.

4. TRANSFORMATION OF THE DIFFERENT SPECIFICATIONS

We now discuss the transformation from the SRS to the Statecharts representations via Z. The Altitude Radar Sensor Processing (ARSP) module specification showing inputs, outputs, and subsystem processing descriptions was chosen for the purpose of our study. The SRS provides a data dictionary with variable definitions, type, and units, and a brief description of variables and functions. This descriptive information is shown in Appendix A. We abstracted the NL-based module specification into Z, preserving variable names, operations (i.e., functionality), dependency and scope. Figure 1 provides an example using the FRAME_COUNTER input variable that illustrates the complete translation from Z to Statecharts. 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

¹ STATEMATE Magnum – product of i-Logix, was used for this case study.

dictionary.

In translating from the NL-based SRS to Z, four ambiguously specified requirements were identified. The first one 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 (see Appendix A). 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.

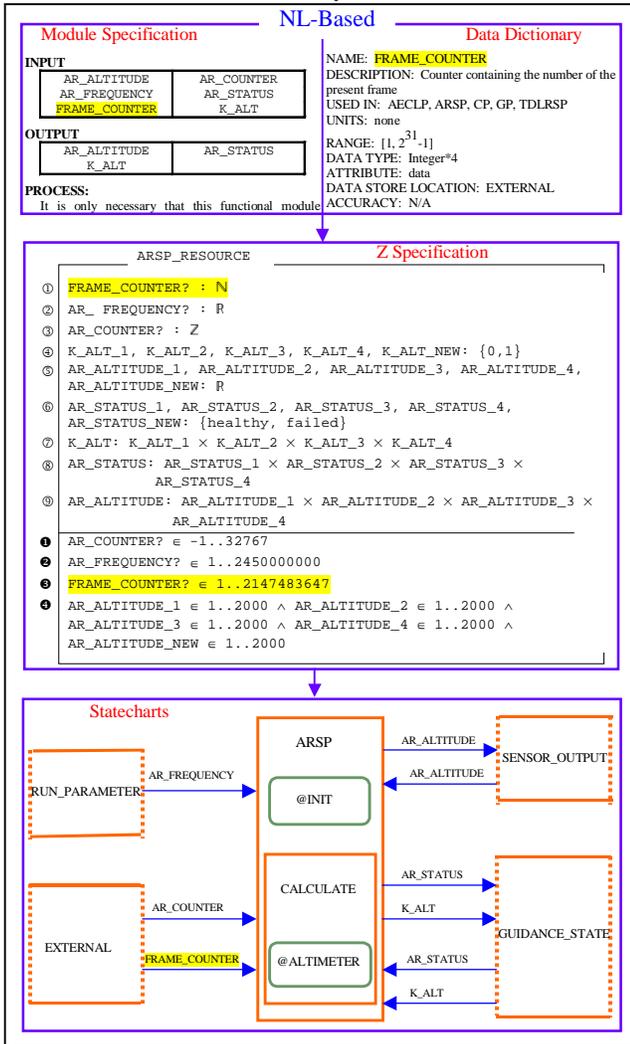


Figure 1. Mapping example from NL-based to Statecharts

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, we defined the Z version of the ARSP specification 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 (Ref. 3). Therefore, the interpretation of Scenario One is inconsistent with the SRS.

On the other hand, in Scenario Two 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 was considered as a complete and consistent transformation. Consequently, Statecharts were developed based on Scenario Two. In this way, Statechart could be used to analyze a model that properly conformed to its requirements, which would be useful in feeding back into the results of our assessment (i.e., symbolic simulation). We also wanted to confirm what we had seen using Z with this other type of formalism, namely Statecharts, and determined if indeed our reformulation revealed similar ambiguity. The detailed Z specification for Scenario Two is described in Appendix B.

5. THE TRANSFORMATION FROM Z TO STATECHARTS

An ARSP project was created within the Statechart framework. Graphic editors were used to create Statecharts and Activity-charts. Once the graphical forms were characterized, state transition conditions and data items were defined.

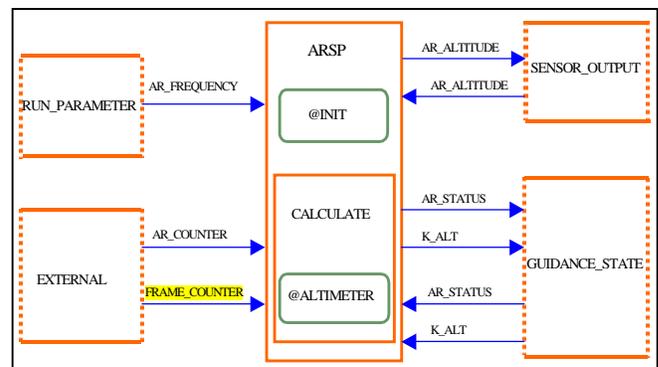


Figure 2. ARSP activity-chart generated with Statechart

These items and/or conditions trigger activities and state transitions that occur within the Statechart model based on definitions within the “data dictionary” and/or the “data bank browser.” The Activity-chart (shown in Figure 2) and Statecharts (see Appendix C) reflect all variables/conditions defined in our Z formulation. During simulation, various color changes help to show the sequence of state changes that occur to validate the system according to its specified structure (based on our Schema signatures) and constraints (based on our Schema predicates). We changed initial (and current) values and conditions while at the same time rerunning and/or resuming the simulation in the process of verifying our assumptions against the Statechart specification. In this way, we exercised the Statechart-based model and generated C code

directly from the charts.

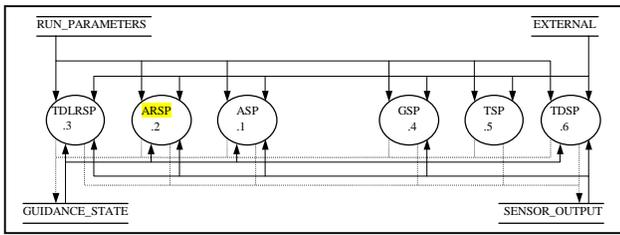


Figure 3. DFD 2.1 SP- Sensor Processing (Ref. 13)

The ARSP Activity-chart shows the data flow between the data stores and the ARSP module based on the information in Figure 2. The direction of the data flow given by Figure 3, which shows what parameters go where, follows the information from the SRS data dictionary (Ref. 14).

6. SPECIFICATION TESTING

Now we discuss the results of our validation effort based on a symbolic simulation of the ARSP Statechart model. In effect, we verified that the ARSP subunit requirements are complete and consistent by running the simulation against all of the Activity/Statecharts. The data used in the simulation is provided in Table 1.

Five conditions (Case 1-5) as shown in Table 1 were defined to test the statecharts. They represent the way we visualized and were able to scrutinize the Z specification. The AR_FREQUENCY value was fixed at 1,500,000,000 to calculate the value of AR_ALTITUDE for all test cases. In the material presented below, we'll explain how each of the conditions was evaluated, this should help to convince the reader that the ARSP subunit is significantly complex (one of six different sensor units used by the GCS).

Table 1. 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, 1, 0, 0]
	AR_COUNTER	-1	19900	-1	20000	-1
Output	AR_STATUS	KP	KP	[1, 0, 0, 0]	[0, -, -, -]	[1, 0, 1, 0]
	K_ALT	KP	KP	[1, 1, 1, 1]	[1, -, -, -]	[0, 1, -, 1]
	AR_ALTITUDE	KP	KP	[*, -, -, -]	[2000, -, -, -]	KP

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

The values of the ARSP output variables are given in Table 1 (KP indicates that the first two element values of the output are the same). All of the output values are the same as expected. All the transitions, activities, and states in the charts were activated precisely as expected. All of the variables were updated as expected. The expected values were calculated based on the given equations in NL-based SRS. Therefore, the result of this simulation shows the previous Z specification was developed correctly. We used simulation of the specification for discovering hidden faults and their location. To accomplish this, faults were injected into the model to simulate memory corruption (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, we found some weak points where faults were lurking (e.g., errors described in Appendix C); (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.

Table 2 shows the specification testing results using fault injection. It describes what state variable is altered at what system state. The system states are the states defined in the Statecharts model. Obviously, the starting state "CURRENT_STATE" shows as a weak system state because any module, improperly initialized, will produce an erroneous output. According to the result table, the CALCULATION and ODD system states are the most vulnerable states to incur failures.

Table 2. 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	✓	✓
ODD	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x	✓	✓
ESTIMATE_ALTITUDE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	N/A	✓	✓	✓	✓
CALCULATE_ALTITUDE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓	✓	✓
KEEP_PREVIOUS	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
DONE	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

x incorrect outputs, ✓ no defect

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

7. CONCLUSION

The result of our analysis revealed that it is beneficial to construct a complete and consistent specification using this method (Z-to-Statecharts). In the process, we uncovered some ambiguity issues associated with our interpretation of the NL-based specification.

The outputs from the ARSP module were examined and shown to be consistent with our 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 way to determine the consistency of the requirements.

The output values from the simulation were checked and compared against the requirements found to be valid. After

running various simulations using fault injection, we uncovered several issues indicating that the SRS is incomplete. In addition, several vulnerable states were identified because faults were injected into the Statecharts and tested. Though the GCS NL-based SRS did not specify fault tolerance, we conclude that the system would not be able to tolerate certain system faults. Through the whole process of this case study, we found that the SRS for the ARSP module was consistent yet not complete and not fault-tolerant. Therefore, our findings indicate that one can better understand the implications of the system requirements using this approach (Z-Statecharts) to facilitate their specification and analysis. Consequently, this approach can help to avoid the problems that result when incorrectly specified artifacts (i.e., in this case requirements) force corrective rework.

APPENDIX A: NL-BASED ARSP SPECIFICATION

The specifications provided below are exactly as they appear in the GCS SRS (Ver. 2.2). The material provided here in Appendix A has been reproduced exactly as it appears in the original SRS. This is done to document and preserve the basis form, which this case study was conducted.

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 perform its normal calculations every other frame, namely on the odd-numbered frames; however, it is required that this functional module execute every frame. The reason for this is that during its normal processing it must rotate history variables. This means that during the frames when it does not need to calculate new outputs, namely the even-numbered frames, it must still rotate its history variables and set its new or current values equal to the previous values, thus creating double entries for each rotated variable. By doubling the entries, consistency of time histories will be maintained at the expense of keeping two copies of each value in these variables, and forcing the functional module to execute every frame.

The processing of the altimeter counter data (AR_COUNTER) into the vehicle's altitude above the planet's terrain depends on whether or not an echo is received by the altimeter radar for the current time step. The distance covered by the radio pulses emitted from the altimeter radar is directly proportional to the time between transmission and reception of its echo. 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.

- **ROTATE VARIABLES:**

Rotate AR_ALTITUDE, AR_STATUS, and K_ALT.

- **PERFORM ALTERNATE PROCESSING:**

If FRAME_COUNTER is an even number, insure that the current values of AR_ALTITUDE, AR_STATUS, and K_ALT are equal to the previous values of AR_ALTITUDE, AR_STATUS, and K_ALT respectively.

- **DETERMINE ALTITUDE:**

- If an echo is received, convert the AR_COUNTER value to a distance to be returned in the variable AR_ALTITUDE according to the following equation:

$$AR_ALTITUDE = \frac{AR_COUNTER \cdot 3 \times 10^8 \frac{m}{sec}}{AR_FREQUEN\ CY \cdot 2} \quad (1)$$

- If an echo is not received, compute AR_ALTITUDE as follows:
 - If all four previous values of AR_STATUS are healthy: In order to smooth the estimate of altitude; fit a third-order polynomial to the previous four values of AR_ALTITUDE. Use this polynomial to extrapolate a value for AR_ALTITUDE for the current time step.
 - If any of the previous four values of AR_STATUS is failed: Set the current value of AR_ALTITUDE equal to the previous value of AR_ALTITUDE.
- **UPDATE STATE:** Set the current values for AR_STATUS and K_ALT according to the following table.

Table 3: Determination of Altitude Status

CURRENT STATE		ACTIONS TO BE TAKEN	
ECHO RETURNED?	All 4 previous AR_STATUS values healthy?	AR_STATUS	K_ALT ²
yes	don't care	healthy	1
no	yes	failed	1
no	no	failed	0

APPENDIX B: Z ARSP 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 1) defines the ARSP module input and output variables. The FRAME_COUNTER? ³ (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

² The K_ALT value is used in the Guidance Processing (GP) module to determine the correction term value of GP_ALTITUDE variable. If K_ALT = 0, the correction term is set to zero. Otherwise, a non-zero value is used in the correction term.

³ The "?" notation in Z represents a variable as an input. The NL-Based SRS defined some variables as both input and output. Z does not provide a way to describe this. So, those variables were treated as variables with neither "?", nor "!" notation.

defined as a set of real numbers that altitude as 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.

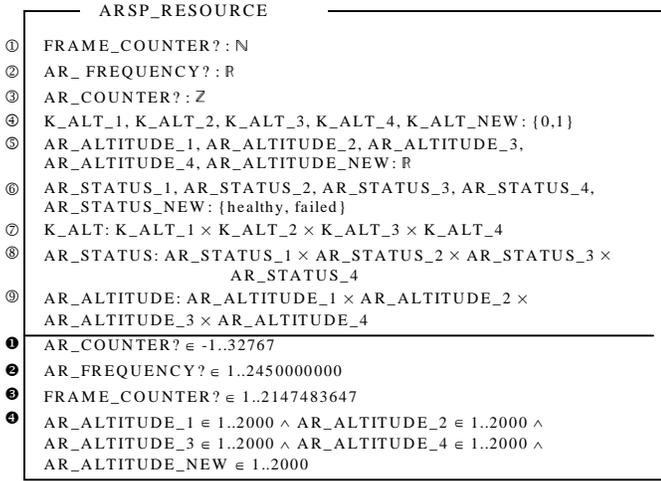


Figure 1. ARSP_RESOURCE Schema

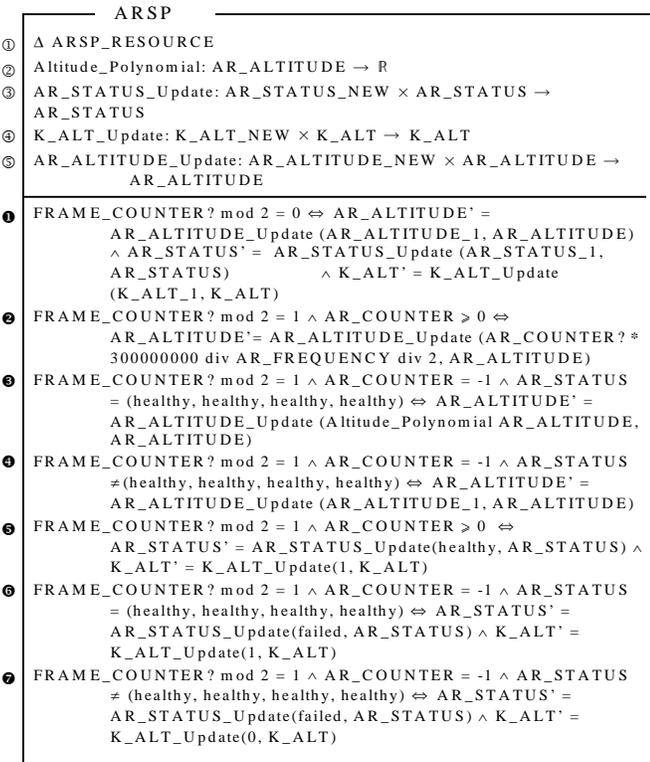


Figure 2. ARSP Schema

The ARSP schema (Figure 2) is the main functional schema of the ARSP module. The ARSP_RESOURCE schema is imported (and is modified) in the Signature ⑩. The Altitude_Polynomial function (Sig. ②) 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.

Predicate ① 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 ② defines 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 ③ 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.

The AR_STATUS, AR_ALTITUDE, and K_ALT variables were defined as a 4-element array in the SRS. Z does not have a specific array construct so these variables are designed as 4-element Cartesian products. The array can also be represented as a 4-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 ①, ②, and ③ represent the variables ranges. The predicate ④ defines the values for the sets in the Signature ⑤.

Predicate ④ 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 ⑤ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd and the AR_COUNTER? is -1. Predicate ⑥ 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 ⑦ 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.

APPENDIX C: STATECHARTS

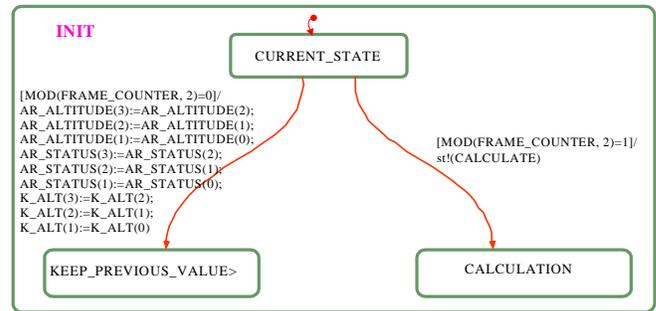


Figure 3. INIT Statechart

The “@INIT” control activity in the ARSP activity chart represents the link to the INIT Statechart. INIT Statechart shows the initialization of the ARSP module and a portion of the ARSP operational schema (Fig.2). 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 ① of Fig.2. 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 Fig. 2.

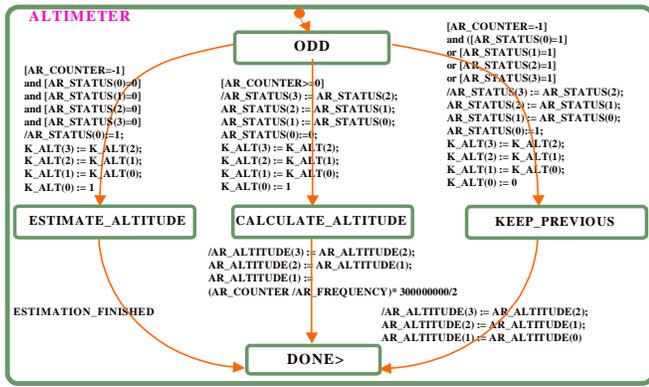


Figure 4. ALTIMETER Statechart

The Altimeter Statechart (Figure 4) 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 ⑥ of Fig.2. The 0 (zero) value of the AR_STATUS means “healthy” which corresponds to the value given in the SRS data dictionary (Ref. 14).

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 ⑤ of Fig.2. 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 ⑦ in Fig.2. The transition from the ESTIMATE_ALTITUDE state to the DONE state happens when the ESTIMATION_FINISHED event occurs. We represented this process as an event because the transaction was described as an undefined third-order polynomial estimation in the SRS⁴. The transaction from the CALCULATE_ALTITUDE state to the DONE state denotes predicate ② (Fig.2). The transaction from the KEEP_PREVIOUS state to the DONE state denotes the predicate ④ (Fig.2) operation.

REFERENCE

1. Leveson, N., *Safeware - system safety and computers*. 1995: Addison Wesley.
2. Heimdahl, M.P.E., Leveson, Nancy G., *Completeness and consistency in Hierarchical State-Based Requirements*. IEEE Trans on SE, 1996. Vol. 22. (NO.6, June 1996).
3. Sommerville, I., *Software Engineering*. 6th ed. 2000, Reading, MA: Addison-Wesley.
4. He, X., *PZ nets - a formal method integrating Petri nets with Z*. Information and Software Technology, 2001. Vol. 43.
5. Hierons, R.M., Sadeghipour, S., Singh, H., *Testing a system specified using Statecharts and Z*. Information and Software Technology, 2001. Vol. 43. (Feb).

⁴ Statemate does not provide predefined mathematical functions, which in this case, would need to support solving a differential equation to estimate the AR_ALTITUDE value.

6. Bussow, R., Weber, M., *A Steam-boiler Control Specification with Statecharts and Z*. Lecture Notes in Computer Science, 1996. Vol. 1165.

7. Grieskamp, W., Heisel, M., and Dorr, H., *Specifying Embedded Systems with Statecharts and Z: An Agenda for Cyclic Software Components*. LNCS 1382, 1998.

8. Damm, W., Hungar, H., Kelb, P., and Schlor, R., *Statecharts - Using Graphical Specification Languages and Symbolic Model checking in the Verification of a Production Cell*. LNCS 891, 1995.

9. Bussow, R., Geisler, R., and Klar, M., *Specifying Safety-Critical Embedded Systems with Statecharts and Z: A Case Study*. LNCS 1382, 1998.

10. Woodcock, J., and Davies, J., *Using Z: Specification, Refinement, and Proof*. Series of Computer Science. 1996: Prentice Hall International.

11. Harel, D., *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 1987. Vol. 8.

12. Harel, D., and Politi, M., *Modeling Reactive Systems with Statecharts*. 1998: McGraw-Hill.

13. *Software Requirements - Guidance and Control Software Development Specification Version 2.2 with formal mods 1-26.*, NASA, Langley Research Center, 1993.

14. *Software Requirements - Guidance and Control Software Development Specification Version 2.2 with the formal mods 1-8*, NASA, Langley Research Center, 1993.

BIOGRAPHY

Frederick T. Sheldon, PhD, AP
 School of Electrical Engineering and Computer Science
 Washington State University
 EME 102 Spokane Street
 Pullman, WA 99164-2752 USA
 Internet (e-mail): Sheldon@acm.org

Frederick T. Sheldon is an Assistant Professor at the Washington State University teaching and conducting research in the area of software engineering. His research is concerned with developing and validating methods and supporting tools for the creation of safe and correct software (focused on verification and validation of systems using modeling and analysis of both logical and stochastic properties). Dr. Sheldon received his Ph.D. at the University of Texas at Arlington (UTA) and has worked at NASA Langley and Ames Research Centers in various capacities since 1993. Prior to that, he worked as a Software Engineer in the area of avionics and diagnostics software development for the YF-22, F-16 and Tornado aircraft programs at General Dynamics and Texas Instruments. He is a member of the IEEE Computer and Reliability Societies, ACM, AIAA, and The Tau Beta Pi and Upsilon Pi Epsilon societies.

Hye Yeon Kim
 School of Electrical Engineering and Computer Science
 Washington State University
 EME 102 Spokane Street
 Pullman, WA 99164-2752 USA
 Internet (e-mail): hkim@eecs.wsu.edu

Hye Yeon Kim is a Graduate Student at Washington State University pursuing her Master’s degree in Software Engineering. Her research interests are Formal methods used in software requirements specification analysis and validation. She had an experience doing research on the Object Oriented Software Metrics on quality aspects. Ms. Kim received her first BS in Science Education from Dankook University and her second BS in Computer Science from Washington State University. She had been a member of the Samsung Electronics Software Membership as an Education Software Developer. She is currently a student member of IEEE Computer Society and ACM. She has been working as a research assistant to Dr. Frederick Sheldon.