Software Requirements Specification and Analysis Using Zed and Statecharts

Frederick T. Sheldon and Hye Yeon Kim
Washington State University
Pullman, Washington 99164-2752, USA
+1 509 335 6138 | +1 509 335 5856
sheldon@acm.org | hyekim@ieee.org

Abstract

This paper presents a prototypical study, of an embedded system requirement specification, used to establish the basis for a complete case study. We are interested in comparing different specification methods that accommodate the notion of state.

A partial modeling of a NASA provided Guidance and Control Software (GCS) development specification was employed. The GCS describes, in natural language, how software is used to control a planetary landing vehicle during the terminal phases of descent. Our ultimate goal is to develop a complete software requirement specification based on the IEEE Standard 830-1998.

The first step in the study was to derive a Zed description for a small portion of the system (Altitude Radar Sensor Processing [ARSP]). The ARSP module reads the altimeter counter provided by the radar and converts the data into a measure of distance to the planet surface.

In the second step, Statecharts were developed to model and graphically visualize the Zed specified ARSP. Using Statemate we analyzed the specification for completeness and consistency. This was accomplished through the generation of activity-charts and simulations.

We present the results of this work and discuss the issues associated with comparing the two methods in terms of their ability to ascertain consistency and completeness of the final products. A more comprehensive assessment of tools publicly available for the specification, modeling and analysis of embedded systems is envisioned.

Keywords: Natural language software specifications, Zed, Statecharts, requirements analysis, reliability

1 PROBLEM DEFINITION

Our greatest need today, in terms of future progress rather than short-term coping with software engineering projects, is not for new languages or tools to implement our inventions, but for more in-depth understanding of whether our inventions are effective and why or why not [1]. Space-born expedition demands very highly engineered systems. A failure in the control software of these systems can be economically and politically disastrous and/or safety critical. To avoid problems in the latter development phases and reduce life-cycle costs it is crucial to ensure that the specification be reliable. By reliable, we mean, is the specification correct, precise, unambiguous, complete, and consistent? Can the specification be trusted to the extent that design and implementation can commence while minimizing the risk of costly errors?

It is difficult to create a reliable specification because such control software tends to be highly complex. Natural language (NL) specifications, though common, have the problem that they are often subject to multiple interpretations, which only complicate the correctness-checking task. Even when NL specifications are developed systematically, it is difficult to ensure their integrity without some form of correctness checking. On the other side, automated 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. This posses a serious concern to industry because there are many different classes of requirements. Different stakeholders typically represent different ways of looking at the problem (or problem viewpoints). Thus, in regards to requirements specification, a multi-perspective analysis is important, as there is no single correct way to analyze system requirements [2]. The usefulness of the requirements specification is diminished by not being understandable to the diverse set of stakeholders. Nevertheless, to address the need to break free of the uncertainty of NL, we investigated the merits of two different mathematically based RSLs.

2 MOTIVATION

Although some members of the software engineering community are quick to announce the latest breakthrough in software engineering technology based on individual success stories, many researchers concur that computer science, especially the software side, needs an epistemological foundation to separate the general from the accidental results [3, 4]. According to Wiener [5], “we need to codify standard practices for software engineering—just as soon as we discover what they should be. Regulations uninformed by evidence, however, can make matters worse.” Clearly, scientific experimentation is needed to supply the empirical evidence for evaluating software engineering methods.

To confront the growing complexity and quantity of software used in commercial aircraft, government regulatory agencies such as the FAA and the DoD have required the use of certain software development processes and techniques. However, no software engineering method(s) (or combination) has been shown to consistently produce reliable, safe software. In fact, little quantitative

evidence exists to show a direct correlation of software development method to product quality. Software verification is the subject of considerable controversy. No general agreement has been reached on the best way to proceed or on the effectiveness of various methods [6, 7]. Moreover, the knowledge base for software engineering has not reached maturity [8].

Computer software allows us to build systems that would otherwise be impossible and provides the potential for great economic gain [1]. The logical constructs of software provide the capability to express extremely complex systems. In fact, computer programs are ranked among the most complex products ever devised by humankind [6]. The complexity intensifies the difficulty of enumerating, much less understanding, all possible states of the program, and thus results in unreliability [9]. Identifying unusual or rare states are particularly problematic. Unfortunately, its those rare software error(s) in a critical system that may cost a human life(s) [10].

Statemate was chosen to model the Zed version of the specification because the key goal of this modeling is testability and pre-development evaluation of the specification itself. We focused on the following issues throughout the process:

- Can ambiguous expressions be found during the process of this study?
- Can the reliability of the end product (i.e., the code) be predicted given the operational environment?
- Is specification level testing (i.e., without implementation) feasible/possible?

3 TOOL BASED ANALYSIS

There are some tools that can be used to test and assess the quality of the software specifications. In this section, three of those tools are briefly reviewed. Consider BEACON, a tool designed for specifying embedded applications. It has some graphical features that are used to create object-based documentation. The design specifications provide a way to graphically visualize the system, are executable and can be used to generate code (i.e., C, Ada, and Fortran). In addition, BEACON supports test case generation and is especially well suited for reverse engineering projects by reason of its legacy code interfacing feature [11].

Workbench is a general-purpose modeling and simulation tool for use in designing sophisticated systems of various types. Workbench evaluates the correctness and performance of a system design. Performance evaluation is done by simulating the model derived from the system specification. Correctness is evaluated by executing, during the simulation, assertions (consistency constraints) that a user attaches to each design specification component [12]. Workbench provides a unique graphical notation for representing the design specification. Workbench is particularly well suited for specifying and evaluating complex systems involving a high degree of concurrency. Using SES Workbench, one can sample data throughout the simulation. Its differentiating features include a statistical navigator and automatic documentation [13].

Statemate [14] also provides a popular and intuitive graphical specification language (in addition to automated documentation and testing capabilities). Users create activity-charts and state charts to represent system operation using a graphical editor. A simulation tool allows one to visualize the system's functionality without creating a physical realization (i.e., code). It also provides a code generator (e.g., C and Ada). Using the above two features the specification can be more thoroughly subjected to evaluation and analysis for such properties as consistency and completeness [15]. By executing the (generated) code one can "debug" the design specifications however the simulation feature provides a better means of fine tuning the design.

4 METHODS

A two-step process was performed using Zed and Statecharts. Our goal was to develop a more "reliable" (i.e., complete and consistent) software requirements specification (SRS). We used an objectified approach to evaluating the results of our two-step process. First, the NL based GCS specification was transformed into a Zed specification. This step necessitated we interpret and make the specification precise (to clarify any ambiguities). Zed proved useful in this regard.

The Zed notation is a mathematical language with a powerful structuring mechanism. In combination with natural language, it can be used to produce a formal specification. We may reason about this specification using the proof techniques of mathematical logic[1]. We may also refine a specification, yielding another description that is closer to executable code [16][2].

In the second step, Statecharts were used to describe the system behaviors combined with activity-charts, which were used to describe the system activities (i.e., its functional building blocks, capabilities, and objects) and the data that flows between them. By using these two languages and Statemate, we developed a conceptual system model. These languages are highly diagrammatic in nature, constituting full-fledged visual formalisms, complete with rigorous semantics [17][3].

---

[1] Additionally, for system designers and managers who do not have a good understanding of Zed, it is necessary to provide English descriptions of Zed structures to convey an intuitive understanding of the specification.

[2] Even though there are currently many free or commercial tools that are available for checking Zed specifications, our experience from a pragmatic view was problematic. To summarize, we were blocked from using automated methods for checking the correctness of the Zed specification.

[3] The conceptual model can be linked with the system's physical (or structural) model, which is described using module-charts (a third language).

5 METHODS APPLICATION

Our experiment focused on applying the methods described above to the Altimeter Radar Sensor Processing (ARSP) module of the Guidance and Control Software (GCS) Development Specification [18]. The GCS, an embedded real-time software system, is at the heart of the Viking Mars Lander. This system was designed to provide software control of the embedded sensors and actuators during the terminal decent phase of the Viking Mars mission. The ARSP module reads data provided by the altimeter radar sensor to determine the lander's altitude from the Mars surface.

The NL specification for the ARSP module, a part of the GCS, is the starting point in this study. Figure 1 shows the location of the ARSP module in the entire GCS system. The next step transformed the B5 into Zed as a concrete specification form. Zed is thus presented as an intermediate step. Subsequently, the ARSP was represented (and evaluated) as Statecharts and activity-charts. The outcome from this evaluation is therefore described below as the final result of our analysis.

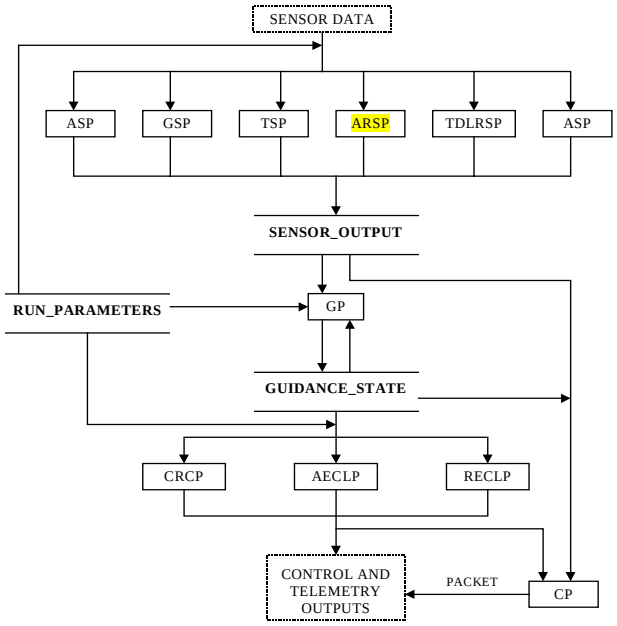


Figure 1 Process 2. RUN_GCS

5.1 B5 – THE NL BASED SPECIFICATION

The GCS Development Specification indicates that it was developed based on the RTCA/DO178A. The DO178 provides guidelines for Software Considerations In Airborne System and Equipment Certification. It does not provide any standards (recommended practice/guidelines) concerning the actual requirements specification of the software unlike the IEEE Std 830-1998. The IEEE standard outlines the specific material and format needed. The GCS however has a concrete and systematic format that it uses to present its content. The format is resembles the old B5 style format used in various Military projects during the Cold War. Hence, we choose to refer to the GCS Development Specification as the "B5[19]." However the B5 clearly states that it is one part of the life cycle data required to fulfill the RTCS/DO178A. In Version 2.2 of the B5, it is clear that this data is considered to be the "Software Requirements" document. While in the Version 2.3 of the B5 (which refers to Version B of RTCS/DO178), the wording "Software Requirements" have been removed. In any case, we used Version 2.2 (with Mods 1-8).

The ARSP module is provided below exactly as it appears within the complete GCS specification [18, 20].

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:
  a. 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:

     Equation 1: AR_ALTITUDE Element calculation

$$\text{AR\_ALTITUDE} = \frac{\text{AR\_COUNTER} \bullet 3 \times 10^8 \frac{m}{\text{sec}}}{\text{AR\_FREQUENCY} \bullet 2}$$

  b. If an echo is not received, compute AR_ALTITUDE as follows:

     1) 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.

     2) 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 1: Determination of Altitude Status.

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

Table 1 gives the state-action constraints. In Table 1, 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.

5.2 Zed Specification

The Zed version of the ARSP module is shown and described at length in this section [16, 21][4]. There are some issues concerning ambiguity that needed to be clarified. The first issue concerns the exact meaning (i.e., purpose) of the rotate variables because the rotational direction of the variables is unclear. The second issue concerns the type assigned to the AR_COUNTER variable. Issue number three is about the fact that an undefined 3rd order polynomial is used to determine the AR_ALTITUDE value. Finally, there is some question about where the AR_COUNTER should be modified.

Three variables are identified that are to be rotated. In this context, it could mean exchanging the values among one another (e.g., Temp:=AR_ALTITUDE, AR_ALTITUDE:= AR_STATUS, etc.) or as was assumed in the Zed version (Fig. 5), rotation occurs within the variable since each variable is a four element array.

As explained above, the AR_STATUS, AR_ALTITUDE, and K_ALT array element values are rotated. The rotation direction could be left shift or right shift. We decided to shift the array element to the right. In Figure 2, "New" means the currently processed value. E1 is the newest and E4 is the oldest value of the array.

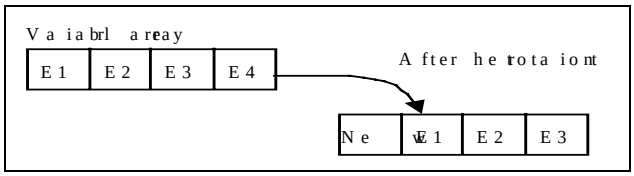


Figure 2 Variable rotation direction

The B5 ARSP module specification describes the AR_COUNTER as a 16-bit binary number. In the data dictionary it is described differently (i.e., as a 2-byte integer). In the Zed version we assumed the data dictionary was correct. In addition, a third-order polynomial is used for estimating the AR_ALTITUDE value however no definition was given anywhere. Accordingly, an undefined function was specified in Zed to represent the fact that such a function is required for estimating the altitude value that would need to be output (i.e., AR_ALTITUDE).

In addition, there are two confusing interpretations about how the AR_COUNTER value is processed. In the first interpretation, if a timely echo (i.e., off the surface of Mars) is received from the altimeter radar then the value of the AR_COUNTER is considered to be the number of pulses from whence the echo was first transmitted. Otherwise, a constant value is assumed (i.e., -1). This is not just an input variable but depends on the state of the radar altimeter and hence can be thought of as an internal ASRP variable. Therefore the actual value of AR_COUNTER is determined internally as part of the ASRP processing. However, the B5 gives the impression that this counter should not be updated inside the ARSP since it is declared as an input variable! Additionally, the B5 claims that AR_COUNTER accumulates the radar pulse cycles from the time of the radar pulse transmission. Which means that the AR_COUNTER value must be a positive number after the transmission whether or not the echo is received in time.

Given this description two possible cases were considered. The first case considers the AR_COUNTER to

[4] The "?" notation in Zed represents a variable as an input. One problem was that the B5 defined variables as both input and output. Zed does not provide a way to describe this. So, those variables were treated as variables with neither "?", nor "!" notation.

be updated inside the ARSP module. The other case assumes that the AR_COUNTER value is ready to be used (and will not be updated by the ARSP processing). The first case is presented as a Zed specification in section 5.2.1. The other case is presented in section 5.2.2.

5.2.1 Case 1: AR_COUNTER updated inside the ARSP

In case one, two conditions are considered. Because we assume the AR_COUNTER value is updated within the ARSP module, it should not represent the arrival of the radar echo pulse. Accordingly, a Boolean flag expressing this condition is introduced (timely arrival or not). To determine the AR_COUNTER value, the time between the initial radar pulse transmission and reception of the return echo is needed. The B5 does not distinguish the difference between the Boolean condition (of a timely echo arrival) and the time value represented by the counter. We believe these should be considered separately for the purpose of clarity. The time value and the echo flag are mentioned inside of the B5 but they are not treated as separate entities. Consequently, in this study we chose to define separate variables with the appropriate types to avoid coupling a double meaning to one variable.

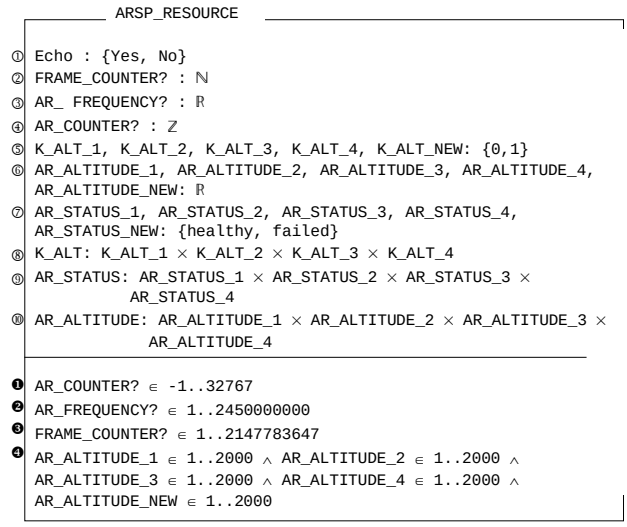


Figure 3 ARSP_RESOURCE Schema

The ARSP_RESOURCE schema (Figure 3) defines the ARSP module input and output variables[5]. Echo (Sig. ①) is a Boolean variable that represents whether the return pulse (radar echo pulse) has been received or not. The FRAME_COUNTER? (Sig. ②) is an input variable giving the present frame number and is typed as a natural number.

AR_FREQUENCY? (Sig. ③) represents the rate at which the AR_COUNTER? has been incremented and is typed as a real number. The AR_COUNTER? (Sig. ④) 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. ⑤, also see Table 1) variables are defined as a set of binary elements. The AR_ALTITUDE_1, AR_ALTITUDE_2, AR_ALTITUDE_3, AR_ALTITUDE_4, and AR_ALTITUDE_NEW (Sig. ⑥) are 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. ⑦) 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 (Sigs. ⑧-⑩) arrays hold the previous 4 values of their elements respectively.

The AR_STATUS, AR_ALTITUDE, and K_ALT variables were defined as a 4-element array in the B5 specification. Zed does not have a specific array construct so these variables are designed as 4-element Cartesian products. The array can be also 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 though the sequence. The predicate ❶, ❷, and ❸ represent the variables ranges. The predicate ❹ constrains the values for the sets defined in the Signature ⑥.

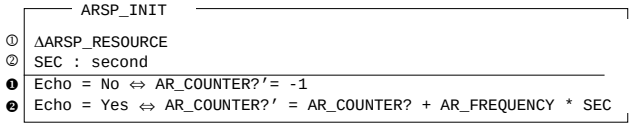


Figure 4 ARSP_INIT Schema

The ARSP_INIT schema (Figure 4) defines the initial state of the ARSP module. It initializes the AR_COUNTER? value to –1 meaning the radar sent out a pulse but has not yet received the echo. Otherwise, the AR_COUNTER? value will be updated as defined by predicate ❷. The SEC (Sig. ②) variable represents seconds from the point the initial radar pulse was transmitted to the present time. It is defined in ARSP_INIT instead of ARSP_RESOURCE because it is not defined as an input by the B5.

The ARSP schema (Figure 5) is the main functional schema. The ARSP_RESOURCE schema is imported for changing in the Signature ①. 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. The AR_STATUS_Update (Sig. ③) is a function that obtains its current value (AR_STATUS_NEW) and the AR_STATUS as input and updates the AR_STATUS array. The K_ALT_Update (Sig. ④) is a function that modifies the K_ALT array by assigning the K_ALT_NEW the new value (a correction term needed by the guidance process).

The AR_ALTITUDE_Update (Sig. ⑤) is a function that updates the AR_ALTITUDE variable by shifting the 1st, 2nd, and 3rd elements value into 2nd, 3rd, and 4th elements

[5] All data types given in the B5 are defined to satisfy the constraints described in the data dictionary [18].

respectively. The current value of AR_ALTITUDE_NEW is pushed into the first element place. The "FRAME_COUNTER? mod 2" is used to represent whether the FRAME_COUNTER? values for the rest of the predicate part are odd or even.

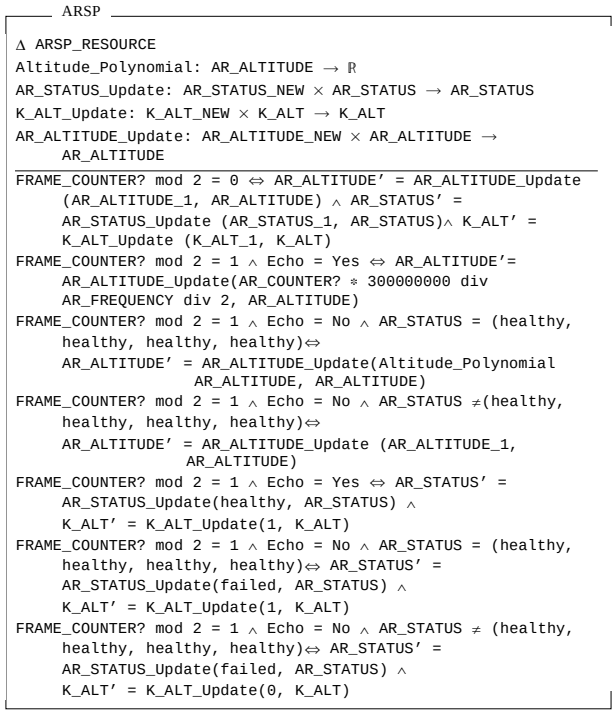


Figure 5 ARSP Schema

Predicate ❶ requires that the current AR_ALTITUDE, AR_STATUS, and K_ALT values remain the same as their previous value when the FRAME_COUNTER? is even.

Predicate ❷ defines the AR_ALTITUDE update. The update takes the current value calculated by Equation 1 when FRAME_COUNTER? is odd and the radar pulse echo has arrived on time.

Predicate ❸ requires that newest AR_ALTITUDE value be estimated by the Altitude_Polynomial function when the FRAME_COUNTER? is odd, the radar echo pulse did not arrive on time, and all AR_STATUS elements are healthy.

Predicate ❹ requires that the newest AR_ALTITUDE value be the same as the previous value when FRAME_COUNTER? is odd, the radar pulse echo did not arrive, and any of the AR_STATUS elements are not healthy.

Predicate ❺ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd and the radar echo pulse is received on time.

Predicate ❻ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd, the echo of the radar echo pulse has not yet been received, 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, the radar echo pulse has not yet been received, and any of the AR_STATUS elements are not healthy.

5.2.2 Case 2: AR_COUNTER, the input variable

In case two, only two schemas are defined and the ARSP_RESOURCE schema of this case is different than in the case one (i.e., the Echo variable is not included). Also, the case one ARSP_INIT schema is not needed. The only assumption in this case 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, it means that the echo of the radar pulse arrived at the time indicated by the value of the counter.

The ARSP_RESOURCE schema (Figure 6) defines the input and output variables for the ARSP module. All of the Zed data types were defined based on their data dictionary entry in the B5 [18]. The FRAME_COUNTER? (Sig. ①) and the AR_FREQUENCY? (Sig. ②) are defined the same as in case one (no need to repeat here). However, the AR_COUNTER? (Sig. ③) is an input variable that contains the counter value of elapsed time from the time of the initial radar pulse transmission toward Mars.

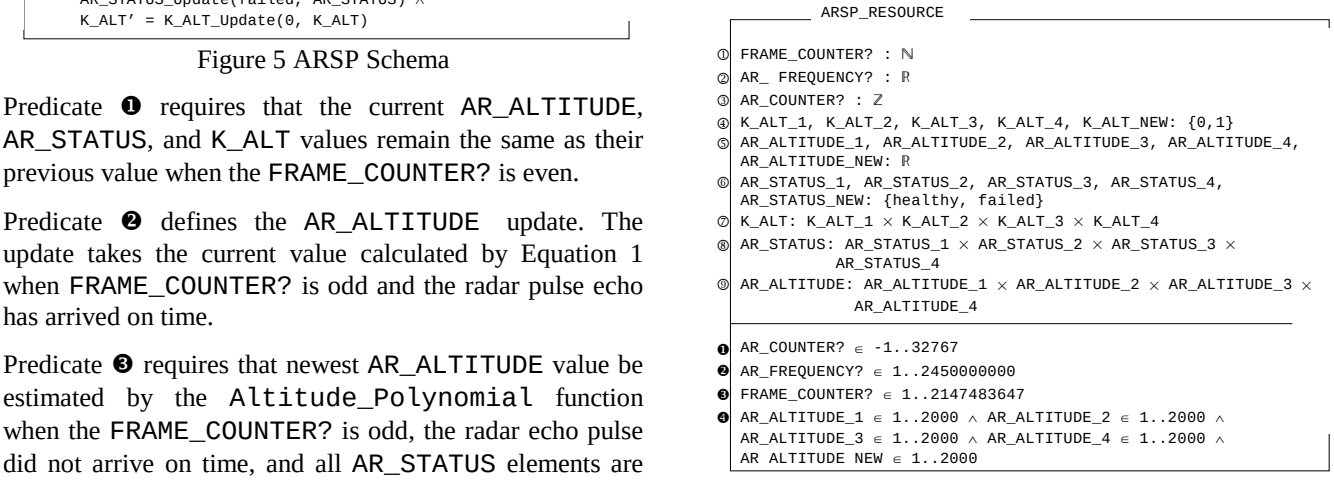


Figure 6 ARSP_RESOURCE schema

The K_ALT_1, K_ALT_2, K_ALT_3, K_ALT_4, and K_ALT_NEW (Sig. ④), *and* AR_ALTITUDE_1, AR_ALTITUDE_2, AR_ALTITUDE_3, AR_ALTITUDE_4, and AR_ALTITUDE_NEW (Sig. ⑤), *and* the AR_STATUS_1, AR_STATUS_2, AR_STATUS_3, AR_STATUS_4, and AR_STATUS_NEW (Sig. ⑥) *as well as* AR_STATUS, AR_ALTITUDE, and K_ALT (Sigs. ⑦-⑨) are all defined the same as in case one. Predicate ❶, ❷, and ❸ represent value ranges of the variables and predicate ❹ defines the possible element values of the predefined sets in Signature ⑤ as was true for case one.

The ARSP schema (Figure 7) is the main functional schema of the ARSP module. The ARSP_RESOURCE

schema is imported (and is modified) in the Signature ①. 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, which is the same as in case one. Similarly, the following signatures are unchanged: AR_STATUS_Update (Sig. ③), K_ALT_Update (Sig. ④), and AR_ALTITUDE_Update (Sig. ⑤). Also, the FRAME_COUNTER? is used in the same fashion specified in case one. The difference lies in how the predicates are specified below.

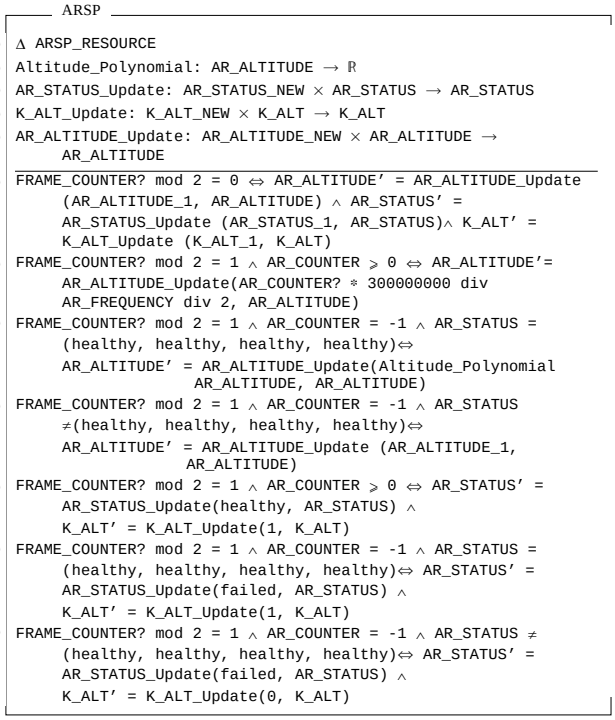


Figure 7 ARSP schema

Predicate ❶ requires that the current AR_ALTITUDE, AR_STATUS, and K_ALT element values be the same as the predecessors when FRAME_COUNTER? is even.

Predicate ❷ defines the AR_ALTITUDE update. The update takes the current value, calculated by the Equation 1, when FRAME_COUNTER? is odd and AR_COUNTER? is greater than or equal to zero.

Predicate ❸ states that the AR_ALTITUDE value is updated (i.e., estimated) by the Altitude_Polynomial function. This is done when FRAME_COUNTER? is odd, AR_COUNTER? is -1, and all the AR_STATUS elements are healthy.

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

Predicate ❺ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd and the AR_COUNTER? is -1.

Predicate ❻ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd, the AR_COUNTER? is -1, and all of the AR_STATUS elements are healthy.

Predicate ❼ requires that the updates to AR_STATUS and K_ALT occur when FRAME_COUNTER? is odd, AR_COUNTER? is -1, and any of the elements in AR_STATUS indicate the Altimeter Radar is not healthy.

5.2.3 Discussion

Let's compare the two cases. Case one presumes that the two separate constraints (i.e., two values with different types) defined in the B5 be represented by two separate variables (i.e., Echo and AR_COUNTER). In the B5, the sign bit of AR_COUNTER represents whether the radar echo pulse is received on time. In case one this condition is split off into the Echo variable while in case two the Echo variable is not introduced. The Zed specification is consistent with the B5 as long as this newly defined Echo variable does not affect any processing unit outside of the ARSP module. This could be the case if, by chance, the sign bit is accessed by some other process. The Echo variable should be treated as an additional input to the ARSP module because otherwise there is no way to determine if the radar echo pulse has been received. This variable was therefore considered an input to the ARSP module.

This leaves the problem of where the Echo, as an input to the ARSP module, will come? Accordingly, we had to revise the Zed version of the ASRP specification to account for this problem. This revision impacts the whole approach to how we planned to specification. Therefore, the interpretation of case one is inconsistent with the B5. However, it reflects the principle that mandates decoupling data [2]. Case 2 does not define any additional variables. Case 2 inherits only the variables defined in the B5 and all the requirements specified in B5 were covered.

Therefore, this reformulation of the B5 is a consistent and complete transformation. For this reason, case 2 was chosen as the basis from which to build the Statecharts. In this way Statemate could be used to analyze, visualize and determine if indeed the reformulation was consistent.

5.3 Statecharts

The Zed version of the ARSP was translated into Statecharts. An ARSP project was created within the Statemate framework first to enable the process. Graphic editors were used to create Statecharts and activity-charts. Once the graphical forms were characterized, state transition conditions and data items were defined. These items and/or conditions trigger activities and state transitions that occur within the Statemate model based on definitions within the "data dictionary" and/or the "data bank browser." The Statecharts and activity-chart are shown in Figure 8, 10, and 11. Once all variables and

possible conditions had been defined, a simulation could proceed. Statemate's Color changes are used to animate how the actions modify the state of the system (i.e., evolve the system defined by all of the various charts). The specification was checked by changing initial (and current) values and conditions and rerunning (and resuming) the simulation. Statemate was used to simulate the translated charts and generate C code directly from the charts.

Figure 8 ARSP activity-chart generated with Statemate

The ARSP activity-chart shows the data flow between the data stores and the ARSP module. This chart is based on the information in Figure 9.

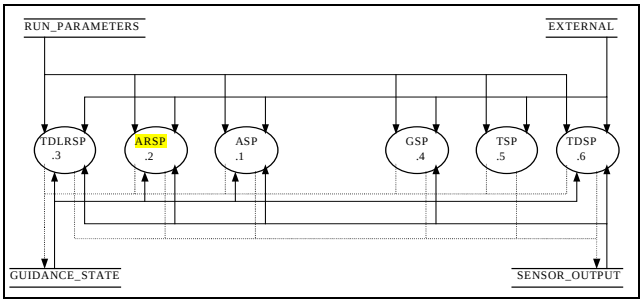


Figure 9 DFD 2.1 SP- Sensor Processing [20]

Figure 9 shows the information flow between the data store and processes, but it does not show which parameters go where. The direction of the data flow in Figure 8 follows the information described in the B5 data dictionary [18].

The "@INIT" control activity in the ARSP activity-chart (Figure 8) represents the link to the INIT Statechart (Figure 10). Figure 10 shows the initialization of the ARSP module and a portion of the ARSP schema (Figure 7) operation. The FRAME_COUNTER_UPDATE is an event that triggers the activity. The transition from the CURRENT_STATE state to KEEP_PREVIOUS_VALUE state describes the predicate ❶ from the ARSP Schema. 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. This was described as "FRAME_COUNTER mod 2 = 1" in the ARSP Schema.

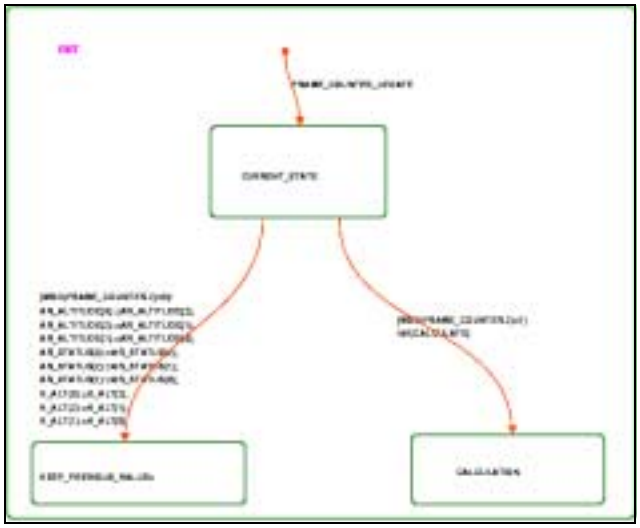


Figure 10 INIT Statechart generated with Statemate

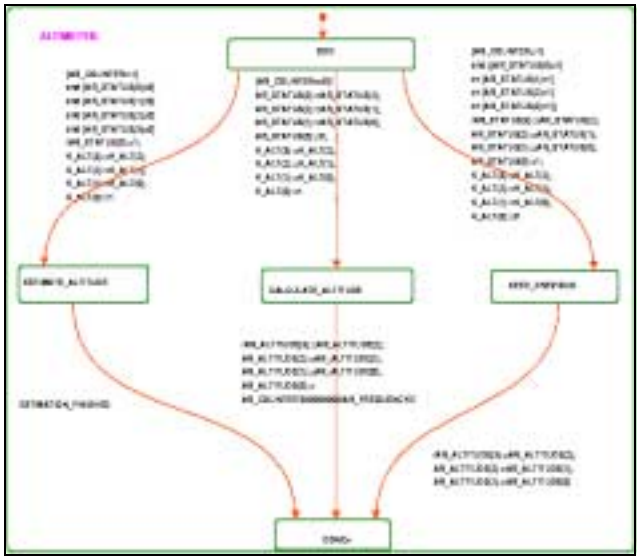


Figure 11 ALTIMETER Statechart generated with Statemate

The ALTIMETER Statechart (Figure 11) 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 ❻ of the ARSP Schema. The 0 (zero) value of the AR_STATUS means "healthy." It corresponds to the value given in the B5 data dictionary [18].

The transition from the ODD state to the CALCULATE_ALTITUDE state begins when a positive value of the AR_COUNTER is given. This transition process is equivalent to the predicate ❺ of the ARSP Schema. 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 AR_STATUS elements are not healthy. This transition has the same meaning as predicate ❼.

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 this transaction was described as an undefined third-order polynomial estimation in B5, and an undetermined function in Zed (i.e., predicate ❸ of the ARSP Schema). 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. The transaction from the CALCULATE_ALTITUDE state to the DONE state denotes predicate ❷. The transaction from the KEEP_PREVIOUS state to the DONE state denotes the operation of predicate ❹.

We tested these three charts by running the Statemate simulator. The data used in the simulation is provided in Table 2, 3 and 4 while the simulated system itself is specified by the activity and statecharts shown in Figure 8, 10, and 11.

Table 2: ARSP Specification Simulation Conditions

Variable	Condition 1	Condition 2	Condition 3	Condition 4	Condition 5	Condition 6
FRAME_COUNTER_UPDATE	-	X	X	X	X	X
FRAME_COUNTER	DC	2	2	1	1	3
AR_STATUS	DC	DC	DC	[0, 0, 0, 0]	DC	[0, 1, 0, 0]
AR_COUNTER	DC	-1	19900	-1	20000	-1

X Event occured, DC Don't Care.

Six conditions are defined as shown in Table 2. These conditions represent test cases against the charts we developed. They represent the way we visualized and were able to inspect the specification we derived from the B5 using our Zed-to-Statecharts method. The AR_FREQUENCY value was fixed at 1,500,000,000 to calculate the value of AR_ALTITUDE for all test cases.

Condition 1 represents the situation where the FRAME_COUNTER value is not updated. The ARSP module is scheduled to run once every frame. This test case covers the time period between frame updates. The expected results are the process blocking at ARSP (see the ARSP activity-chart Fig. 8) with no data changes.

Condition 2 and 3 cover the ARSP module's reaction based on the reception status of the radar echo pulse when the FRAME_COUNTER is even. In both cases' the results should not be different.

Condition 2 covers the case when the FRAME_COUNTER value is changed, its an even value, and the radar echo pulse is *not* yet received. The process is expected to stop in the KEEP_PREVIOUS_VALUE state. Variable rotation should occur in the AR_STATUS, K_ALT, and AR_ALTITUDE array variables. The first two elements for each of these should be same for this test case.

Condition 3 covers the case when the FRAME_COUNTER value is changed, its an even value, and the radar echo pulse *has been* received. The process is expected to stop in the KEEP_PREVIOUS_VALUE state. Variable rotation should occur to the AR_STATUS, K_ALT, and AR_ALTITUDE. The first two elements of these should be same after the process.

Condition 4 is when the updated FRAME_COUNTER is an odd value, the radar echo pulse is *not* yet received, and all the AR_STATUS elements' values are healthy. The process should reach the DONE state by traversing through the ESTIMATE_ALTITUDE state. The AR_STATUS, K_ALT, and AR_ALTITUDE should be updated with new values. The value one (healthy satuts value) should be in the first element of the updated AR_STATUS, and K_ALT variables. AR_ALTITUDE's new value should be an estimated value from the third order polynomial as shown in the Figure 7 predicate ❸.

Condition 5 is when the updated FRAME_COUNTER is an odd value, the radar echo pulse is received, and all the AR_STATUS elements' values are healthy. The process should reach the DONE state through the CALCULATE_ALTITUDE state. The AR_STATUS, K_ALT, and AR_ALTITUDE should be updated with new values. The zero value should be the first element value of the AR_STATUS, and the one value for the K_ALT. AR_ALTITUDE's new value should be a value calculated based on the process shown in the Figure 7 predicate ❷.

Condition 6 is when the updated FRAME_COUNTER value is odd, the echo is not arrived, and one or more of the AR_STATUS elements' values are not healthy. The AR_STATUS and K_ALT variables should be updated with new values and the AR_ALTITUDE variable should have the previous value. The new value for AR_STATUS should be one, and for K_ALT it should be zero. AR_ALTITUDE's first two elements should be the same value after the process because it is keeping the previous value as the current value.

Table 3: ARSP Specification Simulation Result

Name of Chart	Activity / State Name	Condition 1	2	3	4	5	6
ARSP	ARSP	A	A	A	A	A	A
	CALCULATE	-	A	A	A	A	A
INIT	CURRENT_STATE	-	A	A	A	A	A
	KEEP_PREVIOUS_VALUE>	-	A	A	-	-	-
	CALCULATION	-	-	-	A	A	A
ALTIMETER	ODD	-	-	-	A	A	A
	ESTIMATE_ALTITUDE	-	-	-	A	-	-
	CALCULATE_ALTITUDE	-	-	-	-	A	-
	KEEP_PREVIOUS	-	-	-	-	-	A
	DONE>	-	-	-	A	A	A

A Activated, - not activated.

Table 3 and 4 show the results of the simulation. Activation of the states and activities as specified in the charts are shown as an "A" in Table 3.

At condition 1, the ARSP activity is activated but is blocked before the CALCULATE control activity. This is the expected reaction of the system for this condition.

At condition 2 and 3, activity/state activation order is ARSP, CALCULATE, CURRENT_STATE, and KEEP_PREVIOUS_VALUE. This is the correct order as expected.

At condition 4, the activation order is ARSP, CALCULATE, CURRENT_STATE, CALCULATION, ODD, ESTIMATE_ALTITUDE, and DONE. This is the correct order as expected.

At condition 5, the activation order is ARSP, CALCULATE, CURRENT_STATE, CALCULATION, ODD, CALCULATE_ALTITUDE, and DONE. This is the correct order as expected.

At condition 6, the activation order is ARSP, CALCULATE, CURRENT_STATE, CALCULATION, ODD, KEEP_PREVIOUS, and DONE. This is the correct order as expected.

Table 4 ARSP Outputs from the Simulation

Variable	Condition 1	Condition 2	Condition 3	Condition 4	Condition 5	Condition 6
AR_STATUS	NA	KP	KP	[1, 0, 0, 0]	[0, -, -, -]	[1, 0, 1, 0]
K_ALT	NA	KP	KP	[1, 1, 1, 1]	[1, -, -, -]	[0, 1, -, 1]
AR_ALTITUDE	NA	KP	KP	[*, -, -, -]	[2000,-,-,-]	KP

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

The values of the ARSP output variables are shown in Table 4. The outputs under condition one are not applicable because no data processing occurred. KP in Table 4 means that the first two element values of the output are same. All the output values are the same (as expected).

All the transitions, activities, and states in the charts were activated precisely. All of the variables were updated as expected. The result of this simulation show the previous specification was developed correctly. Debugging the C code generated by the code generator feature in Statemate from these charts is another way to test those specifications.

6 CONCLUSION

Even though the entire GCS specification was not evaluated by this method, the result of the completed partial analysis reveals that it is possible to develop a complete and consistent specification with this method (Zed-to-Statecharts). We uncovered some ambiguity issues associated with our interpretation of the B5 specification. The outputs from the ARSP module were examined and shown to be consistent with our expectations by running the simulation. In this context the simulation has provided a means for determining the consistency (i.e., a specification level test) of the requirements.

Our prototypical study has shown that it is possible (albeit time consuming) to use both Zed and Statecharts combined to verify the consistency of software requirements. Using these two RSLs provides and alternative approach to correctness checking on a NL requirements specification. Consequently, this approach can help to avoid the waste problem that results in redevelopment effort from incorrectly specified products.

Reference

1. N. Leveson, "High-Pressure Steam Engines and Computer Software," Computer, 1994.
2. I. Sommerville, "Software Engineering," Addison-Wesley, 2000.
3. W. Gibbs, "Software's Chronic Crisis," Sci. American, 1994.
4. C. M. Holloway, "Software Engineering and Epistemology," ACM SIGSOFT Software Engineering Notes, 1995.
5. L. R. Wiener, "Digital Woes - Why We Should Not Depend on Software," Addison-Wesley, 1993.
6. I. Peterson, "Fatal defect-Chasing Killer Computer Bugs," Random House, Inc, 1995.
7. R. W. Butler, and Finelli, G.B., "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," IEEE Trans. on Software Engineering, 1993.
8. K. J. Hayhurst, "Framework Small-Scale Experiments in Software Engineering (Guidance and control software Project: Software Engineering Case Study)," NASA Langley Research Center, 1998.
9. F. P. Brooks, Jr., "No Silver Bullet - Essence and Accidents of Software Engineering," IEEE Computer, 1987.
10. T. Williams, ed., "It Takes More Than a Keen Nose to Track Down Software Bugs.," Computer Design, 1993.
11. ADI, "Software Development," 2000.
12. SES, "SES/Workbench Creating Models," 1994.
13. Hyperformix.com, "SES/workbench®," 2000.
14. iLogix, "Product Overview," 2000.
15. iLogix, "Statemate Magnum Tutorial," 1999.
16. J. Woodcock, and Davies, J., "Using Z: Specification, Refinement, and Proof," Series of Computer Science, Prentice Hall International, 1996.
17. D. Harel, and Politi, M., "Modeling Reactive Systems with Statecharts," McGraw-Hill, 1998.
18. NASA, "Software Requirements - Guidance and Control Software Development Specification Version 2.2 with the formal mods 1-8," National Aeronautics and Space Administration, Langley Research Center, 1993.
19. DoD, "DOD-STD-2167A," 1988.
20. NASA, "Software Requirements - Guidance and Control Software Development Specification Version 2.2 with formal mods 1-26.," National Aeronautics and Space Administration, Langley Research Center, 1993.
21. J. M. Spivey, "The Z Notation: A Reference Manual," Prentice Hall Int'l, 1992.

Software Requirements Specification and Analysis Using Zed and Statecharts

Formal Descriptions and Software Reliability

Oct. 7th, 2000.

Frederick T. Sheldon and Hye Yeon Kim

School of Electrical Engineering and Computer Science

Washington State University

Research Agenda

- Goal:
  Develop a complete software requirement specification based on the IEEE Standard 830-1998.
  - Determine completeness and consistency.
  - Compare methods.
- Target specification:
  A NASA provided Guidance and Control Software (GCS) development specification for the Viking Mars Lander.
- Analysis Approach:
  Zed, and Statecharts.
- Summary of present research status and future study.

Focus: Testing the Requirements

- Can ambiguous expressions be found during the process of this study?
- Can the reliability of the end product (i.e., the software system) be predicted given the operational environment?
- Is specification level testing (i.e., without implementation) feasible/possible?

Completeness

An SRS is complete if, and only if, it includes the following elements:

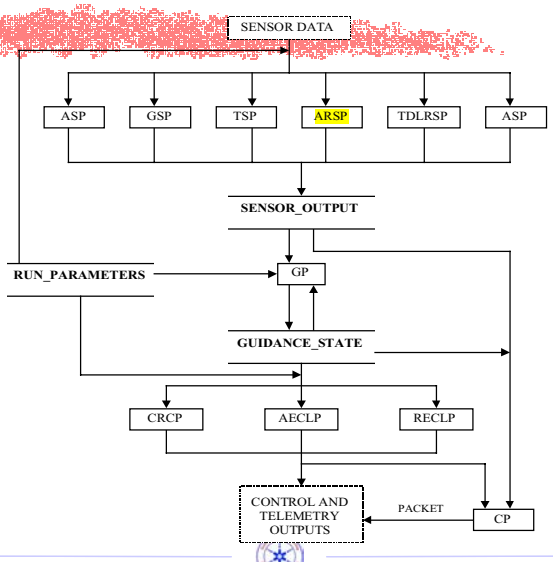
- All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interface. In particular, any external requirements imposed by a system specification should be acknowledged and treated.
- Definition of the response of the software to all realizable classes of input data in all realizable classes of simulations. Note that it is important to specify the responses to both valid and invalid input values.
- Full labels and references to all figures, tables, and tables and diagrams in the SRS and definition of all terms and units of measure.

- IEEE STD 830-1998, pp.5-6.

Guidance and Control Software

- Software Requirements - GCS Development Specification.
- This system was designed to provide software control of the embedded sensors and actuators of the Viking Mars Lander during the terminal decent phase of the mission.
- The ARSP module reads data provided by the altimeter radar sensor to determine the lander's altitude from the Mars surface.

Process 2: RUN_GCS



Ambiguity issues

- The exact meaning of the rotate variables, and direction of the rotation *was unclear*
- The type assigned to the `AR_COUNTER` variable *was unclear*
- An *undefined* 3rd order polynomial
- Where the `AR_COUNTER` should be modified? (dividing point of the cases I and II)

Zed – *Case 1* continue

ARSP_INIT

```
ΔARSP_RESOURCE
SEC : second
```

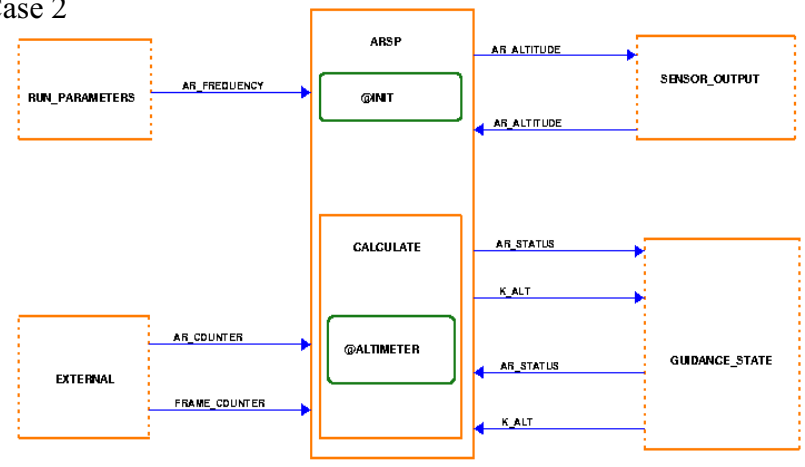
1. `Echo = No ⇔ AR_COUNTER?' = -1`
2. `Echo = Yes ⇔ AR_COUNTER?' = AR_COUNTER? + AR_FREQUENCY * SEC`

Zed – Discussion

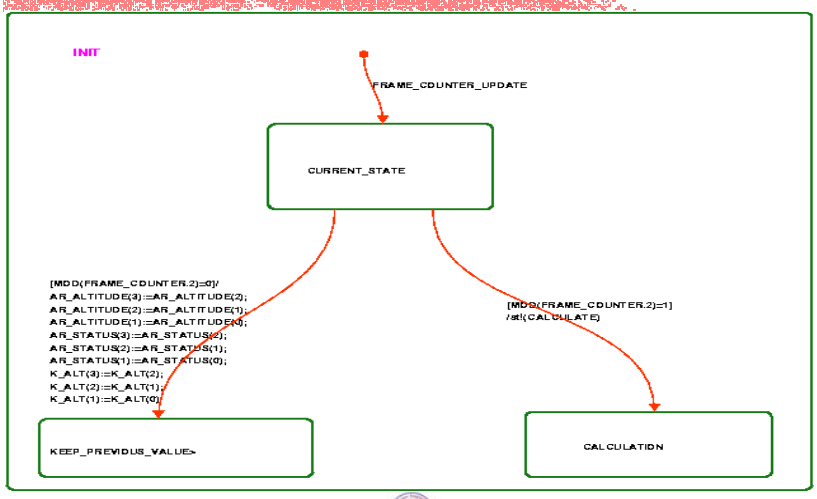
- `Echo` and `AR_COUNTER`
- Case 1 is inconsistent with B5 due to use of two newly defined variable (Echo and SEC) (requires revision of whole specification)
- Case 2 completely models the B5 definitions and requirements without any addition (verbatim interpretation)
- Case 2 was used to build the *Statecharts*

Statemate's – Activity Charts

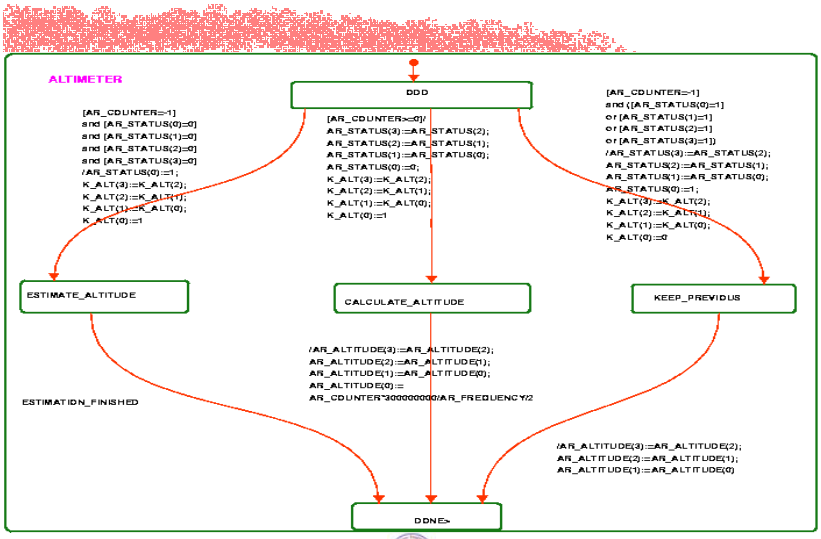
Case 2



Statecharts continue



Statecharts continue



Simulation Conditions (inputs to Sim)

Condition / Variables	1	2	3	4	5	6
FRAME_COUNTER_UPDATE	-	X	X	X	X	X
FRAME_COUNTER	DC	2	2	1	1	1
AR_STATUS	DC	DC	DC	[0,0,0,0]	DC	[0,1,0,0]
AR_COUNTER	DC	-1	19900	-1	20000	-1

X: Event occurred, DC: Don't Care

- AR_FREQUENCY value was fixed @ 150,000,000 for all cases.

ARSP Activity/State Activation Results

Name of Chart	Activity/State Name	Conditions					
		1	2	3	4	5	6
ARSP	ARSP	A	A	A	A	A	A
	CALCULATE	-	A	A	A	A	A
INIT	CURRENT_STATE	-	A	A	A	A	A
	KEEP_PREVIOUS_VALUE>	-	A	A	-	-	-
	CLACULATION	-	-	-	A	A	A
ALTIMETER	ODD	-	-	-	A	A	A
	ELSTIMATE_ALTITUDE	-	-	-	A	-	-
	CALCULATE_ALTITUDE	-	-	-	-	A	-
	KEEP_PREVIOUS	-	-	-	-	-	A
	DONE>	-	-	-	A	A	A

ARSP Outputs from Simulation

Condition / Variables	1	2	3	4	5	6
AR_STATUS	NA	KP	KP	[1,0,0,0]	[0,-,-,-]	[1,0,1,0]
K_ALT	NA	KP	KP	[1,1,1,1]	[1,-,-,-]	[0,1,-,1]
AR_ALTITUDE	NA	KP	KP	[*,-,-,-]	[2000,-,-,-]	KP

NA: Not Applicable, -: Don't Care, KP: Keep Previous value, *: An estimated value

Conclusion

- It is possible to develop a complete and consistent specification with Zed-to-Statecharts method.
- We uncovered some ambiguity issues.
- The outputs from the ARSP module were examined and shown to be consistent with our expectations by running the simulation.
- In this context the simulation has provided a means for determining the consistency (i.e., a specification level test) of the requirements.

Future Study

- ARSP model analysis with Petri-net based tool (e.g., UltraSAN or SPNP6).
- GCS scheduling mechanism analysis with a model based method other than Zed and SES Workbench.
- Build complete and consistent GCS SRS with the analysis results.