



# Testing Software Requirements with Z and Statecharts Applied to an Embedded Control System<sup>\*,\*\*</sup>

HYE YEON KIM<sup>\*\*\*</sup>

*Samsung Electronics, S/W Group, Digital Appliance R&D Ctr., 416, Meatan-3Dong, Paldal-Gu, Suwon, Kyounggi-Do, Korea 442-742*

hyekim@ieee.org

FREDERICK T. SHELDON<sup>†</sup>

*Oak Ridge National Laboratory Computational Sciences and Engineering, Oak Ridge, TN 37831-6363, USA*

sheldonft@ornl.gov

**Abstract.** Software development starts by 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 is correct. This paper describes how to model, test and evaluate (i.e., check, examine, and probe) a natural language (NL) SRS using two formalisms (Z and Statecharts). These formalisms are used to determine strategies for avoiding design defects that stem from the requirements that could ultimately lead to system failures. A case study was performed to validate the integrity of a Guidance Control SRS in terms of completeness, consistency, and fault-tolerance. Based on these experiences, the NL-specification  $\rightarrow$  Z  $\rightarrow$  Statechart transformations can be completed in a systematic and repeatable manner that yield valuable insight into the overall integrity of software specifications.

**Keywords:** Z, Statecharts, requirements specification and validation, completeness, consistency, fault-tolerance

## 1. Introduction

Every system of consequence needs good requirements. Project risks increase dramatically without good requirements. The better the requirements, the better people will

\* Kluwer acknowledges that this contribution was co-authored by a contractor or affiliate of the U.S. Government (DOE Contract E-AC05-00OR22725). As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.

\*\* The appendix provides the total system architecture (see Figure A.1) and all of the Statecharts. All of the finalized (i.e., proved) schemas can be obtained from the [www.ilogix.com](http://www.ilogix.com) university page or from the [www.csm.ornl.gov/~sheldon](http://www.csm.ornl.gov/~sheldon) publication page (see Hye Yeon Kim, Thesis [defended May 14, 2002]). The published paper uses gray scale for the figures and charts. A full color version with over 20 colored figures is available from Sheldon's publication page or at Kluwer (<http://www.kluweronline.com/issn/0963-9314/>).

\*\*\* Most of this work was completed while Ms. Kim was a graduate student at The Washington State University (WSU). She is a founding member of the Software Engineering for Dependable Systems Laboratory.

<sup>†</sup> 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 Secure and Dependable Systems) Lab. Some of this work was completed while he was a research staff member at DaimlerChrysler (RIC/AS) on leave from his faculty position at WSU. <http://csm.ornl.gov/~sheldon>

1 understand what they are trying to build. The increasing pervasiveness of embedded 1  
2 software and the fact that software requirements are increasingly complex necessitate 2  
3 the use of formal and rigorous approaches in the specification and validation of re- 3  
4 quirements. Requirements validation is concerned with checking the requirements 4  
5 document (i.e., SRS [Software Requirements Specification]) for consistency, com- 5  
6 pleteness and accuracy (Kotonya and Sommerville, 1998), and ensures the specifi- 6  
7 cation represents a clear description of the system for subsequent design, implementa- 7  
8 tion, refinement and evolution (and ultimately assures that the requirements meet with 8  
9 stakeholders' needs). A more thorough (i.e., formal and rigorous) validation approach 9  
10 should ultimately reduce the risk of failures that lead to costly rework and corrective 10  
11 action. In this sense we validate the SRS of an embedded real-time software—the 11  
12 Viking Mars Lander Guidance Control Software (NASA, 1993). In essence, formal 12  
13 systematic and rigorous approach was employed to ensure the correctness/accuracy, 13  
14 completeness/consistency of the SRS.<sup>1</sup> We seeded faults into the executable model to 14  
15 evaluate the SRS resiliency (tolerance of such faults). 15

16 The notations selected to express requirements or designs can have a very impor- 16  
17 tant impact on the construction time, correctness, efficiency, and maintainability of 17  
18 the target application. One desirable property for these notations is that they be pre- 18  
19 cise and unambiguous, so that stakeholders and developers can agree on the required 19  
20 behavior and observe the actual behavior through some means of simulations. The 20  
21 notation should make it possible to state and reason about the system properties. Con- 21  
22 structing the system in compliance with those specifications, given the proper/sound 22  
23 notational foundation, will thus provide a higher level of confidence that the system 23  
24 will correctly exhibit those important properties and behaviors. This implies that the 24  
25 selected notation be formally defined and amenable to mathematical/logical manipula- 25  
26 tion. Observation of behaviors is particularly convenient if the specification language 26  
27 is executable. Executable specifications are also useful for clarifying and refining re- 27  
28 quirements and designs (Shaw, 2001). 28

29 *Formal methods* (FMs) apply to a variety of methods, from light and agile to heavy 29  
30 weight, used to ensure correctness. Their common characteristic is a mathematical 30  
31 foundation. Our approach combines a model-based method (i.e., using set theory, 31  
32 propositional and predicate logic) with a state-based diagrammatic formalism to visu- 32  
33 alize and simulate the specification (including fault injection). Z (pronounced “Zed”) 33  
34 is employed to prove correctness of the SRS while the behavior of executable specifi- 34  
35 cations is gauged through visualization and simulation using Statecharts. 35  
36 36

### 37 37

#### 38 1.1. Definitions 38

### 39 39

40 Integrity, as applied to the SRS, investigates the questions: (1) Is the specification cor- 40  
41 rect, unambiguous, complete, and consistent? (2) Can the specification be trusted to 41  
42 the extent that design and implementation can commence while minimizing the risk 42  
43 of costly errors? And, (3) how can the specification be defined to prevent the propa- 43  
44 gation of errors into the downstream activities? By evaluating these questions in the 44  
45 context of the set of important requirements we claim that the reliability of the conse- 45  
46 quential system will be improved. In theory, the heavier weight FMs, if tractable and 46

1 affordable, ensure a higher level of reliability. Naturally, we supposed that combining 1  
2 the strengths of two lighter weight methods could indeed be more tractable/affordable 2  
3 while at the same time ensure a higher level of requirements integrity and deployed 3  
4 system reliability. 4

5 Specification completeness can be thought of as the *lack of ambiguity*. If system 5  
6 behavior is not precisely and completely specified then the required behavior for some 6  
7 activity (function), event(s) or condition(s) is omitted or is subject to multiple inter- 7  
8 pretations (Leveson, 1995). The lack of ambiguity in requirements, conflicting re- 8  
9 quirements and *undesired* nondeterminism (Czerny, 1998) and promotes a clear and 9  
10 consistent compilation of requirements (i.e., consistency of the specification). 10

11 Fault-tolerance (FT) is the built-in capability of a system to provide continued cor- 11  
12 rect execution in the presence of a limited number of hardware or software faults. 12  
13 To achieve FT necessitates an implementation methodology and architecture that pro- 13  
14 vides for (1) error detection for fault conditions, and (2) backup routines for continued 14  
15 service to critical functions handle errors that arise during operation of the primary 15  
16 software (Pradhan, 1996). For an SRS to support such a capability, we call for the 16  
17 existence of specified requirements that necessitate such capabilities (i.e., detect errors 17  
18 for all fault conditions which may not be practically achievable). Moreover, the SRS 18  
19 should include requirements that encourage system robustness, software diversity, and 19  
20 temporal redundancy for continuing service of critical system functions when stressed 20  
21 beyond normal operating limits in the presence of failure(s). 21

## 22 2. Related research 22

23 Several categories of analysis methods are introduced for safety/mission critical sys- 23  
24 tem software requirements. The studies presented here seek to ensure the consistency 24  
25 and completeness of an SRS. Numerous studies were reviewed that use Z, among other 25  
26 formal methods that gain benefit from visualization and/or dynamical assessment. 26  
27 27  
28 28  
29 29

### 30 2.1. Formal methods 30

31 Formal methods are a collection of techniques, rather than a single technology, most 31  
32 notably for specifying a software system. The main objective provides for eliminat- 32  
33 ing inconsistency, incompleteness, and ambiguity. Because FMs have an underlying 33  
34 mathematical basis, they provide more rigorous analysis regimen over other more ad 34  
35 hoc reviews. There are several classes of distinguishable formal specification tech- 35  
36 niques. They are property-oriented specifications, model-oriented specifications, and 36  
37 operational specifications (Gaudel and Bernot, 1999). 37  
38 38  
39 39

40 In the property-oriented approaches, known as constructive techniques, one declares 40  
41 a name list of functions and properties. These approaches provide notations that can 41  
42 depict a series of data, and use equations to describe the system behaviors rather than 42  
43 building a model. These property-oriented approaches can be broken into algebraic 43  
44 and axiomatic specifications (Vliet, 2000). The algebraic specification describes a 44  
45 system consisting of a set of data and a number of functions over this set (Sannella 45  
46 and Tarlecki, 1999). The axiomatic specification has its origin in the early work on 46

1 program verification. It uses first-order predicate logic in pre- and post-conditions to  
2 specify operations (Vliet, 2000).

3 The objective of the model-oriented approach, utilizing declarative techniques, is  
4 to build a unique model from a choice of built-in data structures and construction  
5 primitives provided by the specification language (Gaudel and Bernot, 1999). This ap-  
6 proach provides a direct way for describing system behaviors. The system is specified  
7 in terms of mathematical structures such as sets, sequences, tuples, and maps (Vliet,  
8 2000). Model behaviors are compared against the specified functionality as a measure  
9 of correctness (Gaudel and Bernot, 1999). Vienna Development Method (VDM), B,  
10 and Z belong to this category.

11 The operational/executable specification technique is another category. It provides  
12 sets of actions describing the sequence of the system behavior and computational  
13 formulas that describe the performance calculation. Petri nets, process algebra, and  
14 state/activity charts in the STATEMATE<sup>2</sup> environment (Shaw, 2001) fall into this cat-  
15 egory (Gaudel and Bernot, 1999).

## 18 2.2. *Analysis/evaluation/assessment studies*

19 Numerous studies have been conducted with the goal of improving integrity, identi-  
20 fying defects, and removing ambiguities. Fabbrini et al. proposed an automatic eval-  
21 uation method called “Quality Analyzer of Requirements Specification (QuARS)” to  
22 evaluate quality which define testability, completeness, understandability, and con-  
23 sistency as properties of a high quality SRS (Fabbrini et al., 2001). The QuARS  
24 tool parses requirement sentences written in natural language (NL) to detect poten-  
25 tial sources of errors. This is a linguistic, informal evaluation approach rather than  
26 a FM but demonstrates how informal systematic approaches are useful for revealing  
27 errors. This approach may be useful in many domains because QuARS’s dictionaries  
28 are customizable.

29 Heitmeyer et al. used the Software Cost Reduction (SCR) tabular notation to iden-  
30 tify inconsistencies in SRSs. They describe, using their notation/method, how a safety  
31 violation is exposed. Typically, the enormous state space of practical software specifi-  
32 cations render direct analysis impractical (Heitmeyer et al., 1998). They show in their  
33 “Two Pushbutton” abstraction method how to reduce a system state space from infinite  
34 to finite. Two redundant specifications represent the required system behavior using  
35 both Petri net and TRIO specification logic. They abstract and analyze their SRS with  
36 Spin and a simulator developed specifically to support the SCR method.

37 Heimdahl and Leveson used their Requirements State Machine Language (RSML)  
38 to verify requirements specifications for completeness and consistency (Heimdahl and  
39 Leveson, 1996). RSML is a state-based language suitable for the specification of reac-  
40 tive systems. It includes several features developed by Harel for Statecharts. In RSML,  
41 the transitions are represented as relationships between states (i.e., hierarchical, next-  
42 state mappings). The functional framework defined in (Heimdahl and Leveson, 1996)  
43 is used to check the model against every possible input to find conflicting requirements  
44 (i.e., to verify whether the model is deterministic). They used a textual-representation-  
45 based simulator developed for RSML to execute the specification. One advantage is  
46

1 the ability to analyze subparts of the whole system without needing to generate a global  
2 reachability graph.

### 3 4 5 2.3. *Related Z case studies*

6 Numerous studies have been conducted that combine Z with other FMs. Xudong He  
7 proposed a hybrid FM called PZ-nets. PZ-nets combine Petri nets and Z (He, 2001).  
8 PZ-nets provide a unified formal model for specifying the overall system structure,  
9 control flow, data types and functionality. Sequential, concurrent and distributed sys-  
10 tems are modeled using a valuable set of complementary compositional analysis tech-  
11 niques. However, modular and hierarchical facilities are needed to effectively apply  
12 this approach to large systems.

13 Hierons, Sadeghipour, and Singh present a hybrid specification language  $\mu$ SZ (Hi-  
14 erons et al., 2001). The language uses Statecharts to describe the dynamical system  
15 behavior and Z to describe the data and their transformations. In  $\mu$ SZ, Statecharts de-  
16 fine sequencing while Z is used to define the data and operations. They abstract data  
17 from the Z specifications to produce an Extended Finite State Machine (EFSM) rep-  
18 resented with Statecharts. EFSM features can also be utilized for test case generation.  
19 These features automate setting up the initial state and checking the final state for each  
20 test. The dynamic system behaviors specified in Statecharts are checked using these  
21 features.

22 Bussow and Weber present a mixed method consisting of Z and Statecharts (Bussow  
23 and Weber, 1996). Each method is applied to a separate part of the system. Z is used  
24 to define the data structures and transformations. Statecharts are used to represent the  
25 overall system and reactive behavior. The Z notations are type checked with the ESZ  
26 type-checker but the Statecharts semantics are not fully formalized. Several other case  
27 studies utilize Z for defining data while Statecharts are used as a behavioral description  
28 method (Grieskamp et al., 1998; Damm et al., 1995; Bussow et al., 1998).

29 Castello developed a framework for the automatic generation of Statecharts layouts  
30 from a database that contains information abstracted from an SRS (Castello, 2000).  
31 The framework centerpiece is the “statecharts layout” tool. The tool’s output is then  
32 transformed into Z schemas. Data is abstracted from the SRS to generate a database  
33 that provides the basis from which to automatically generate the statecharts layout.  
34 Statecharts are translated one-by-one into Z schemas to validate the correctness. The  
35 Z schemas are exact replicas of the Statecharts (i.e., the Z schema is the text version  
36 of the Statechart). Both the method and the criteria for the SRS abstraction are not  
37 explained (Castello, 2000).

### 38 39 40 2.4. *Contribution from this study*

41 The Statecharts we developed are derived from the Z specification which is in turn de-  
42 rived from the natural language based (NL-based) SRS. Moreover, components of the  
43 SRS (i.e., functions described with in subsections of the various chapters) are trans-  
44 lated completely into Z and then completely into State/Activity charts.<sup>3</sup> The Z speci-  
45 fication is type checked and proved using Z/EVES<sup>4</sup> with reduction/refinement proce-  
46

dures prior to the second translation phase (i.e., into Statecharts). Furthermore, many Z specifications/schemas are expressed in a form that makes them amenable to symbolic evaluation. If an operation defines the outputs and final state variables as functions of the inputs and initial state variables, then it can be symbolically evaluated. In such cases, Z/EVES is used to investigate the results of a sequences of operations, by defining a “test case” schema as a composition of individual operations for example. The Z/EVES prover combines automatic strategies and detailed user steps, allowing for a collaborative effort in completing a proof. Z/EVES can look after mundane details such as side-conditions on proof steps and trivial subgoals, leaving the user free to focus on the main line of argument of a proof. Z/EVES offers some powerful automatic commands for proving theorems (e.g., prove, or reduce). However, these commands will only succeed in proving easy theorems, and then only when the way has been prepared. For example, when a name is defined by an abbreviation definition, axiomatic box, or generic box, it may be necessary for some simple theorems to be stated before the automatic steps can succeed.

In the second phase, 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 further described in Sections 3 and 4.

Z and Statecharts have different kind of precision for revealing inherent SRS flaws. Generally, Z is better suited for defining data types while Statecharts are best at describing the dynamic behavior (i.e., state transitions) (Grieskamp et al., 1998; Damm et al., 1995; Bussow et al., 1998) by giving a state-based visualization. When one uses conjoined methods as in 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 Z/EVES prover. The consistency and completeness of the Statecharts model are verifiable using the model checker and simulations. Refinement between these two different formalisms gives an in-depth understanding of requirements, and reveals different SRS flaws that may exist. Focusing these techniques on the most critical functions initially, as described by the SRS, enhanced the usefulness of this approach.

### 3. Applied methods

As described above, 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 SRS sections as described here in Table 1. The Processing Unit describes the AR\_COUNTER modification

Table 1. NL-based specification for AR\_COUNTER (NASA, 1993)

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

rules and the value ranges. One concludes from the first two sentences that the AR\_COUNTER value 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. These conclusions conflict with the last sentence, which states that the AR\_COUNTER will contain sixteen, one bits representing a negative one ( $-1$ ) as found in the data dictionary definition.

Second, the Schemas are manually transformed into State/Activity charts and symbolically executed to assess the model's behavior based on the 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 (countably) infinite number down to a finite number to enable simulations be performed to ensure no nondeterministic state/activity transitions (i.e., inconsistencies) exist. After checking for inconsistencies, in a second step, all data and transition-conditions are specified (i.e., added in). Simulations are performed again to determine if any new inconsistencies have been added. 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/refinement process is complete, faults are injected into the State/Activity charts. Changing state variable values while running a simulation accomplishes this precisely. 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 failures.

### 3.1. Z

Z is classified as a model-based specification language 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 (Woodcock and Davies, 1996).

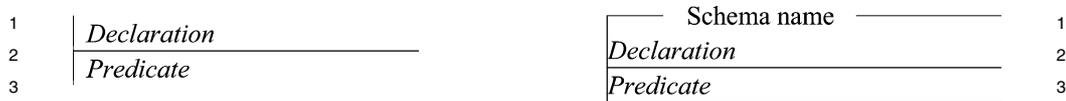


Figure 1. Forms of an axiom and a schema.

Axioms are a common way of defining global objects in Z. There are two parts: a declaration and a 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 by other schemas. These facilities are useful and essential in clarifying ambiguities and solidifying one's understanding of the requirements.

### 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 (Harel, 1987).

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 and fully-matured visual formalisms, complete with rigorous semantics providing an intuitive and concrete representation for inspecting and checking for conflicts (Harel and Politi, 1998). 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 specification inconsistencies and/or incompleteness.

A GCS project was created within the Statemate environment. Graphical editors were used to create State/Activity charts. Once the graphical forms are characterized, state transition conditions and data items are 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

1 conditions are changed while rerunning and/or resuming the simulation in the process  
2 of ensuring consistency and completeness against the Statecharts specification.

### 3.3. *Specification tests*

3  
4  
5  
6  
7  
8 The Statecharts model is examined in two different ways. First, the State/Activity  
9 charts are tested as finite state machines (ensuring state transition conditions and ac-  
10 tivity triggers are deterministic). Next, their functionality is tested. The actual outputs  
11 (values generated by the State/Activity charts simulations) are compared with the ex-  
12 pected output.

13  
14 **3.3.1. *Finite state machine approach.*** This approach identifies absorbing (i.e.,  
15 failure) States/Activities as well as nondeterministic State/Activity transitions. Bog-  
16 danov and Holcombe have discussed how to test Statecharts for an aircraft control  
17 system (Bogdanov and Holcombe, 2001) by examining the underlying finite state ma-  
18 chine(s). We extend their method to evaluate if the Statecharts are behaviorally equiv-  
19 alent to the SRS. In other words, every activity and state transition is exercised as  
20 described in the SRS.

21  
22 **3.3.2. *Data item approach.*** In the data item approach, the state/activity charts are  
23 treated like a software program (i.e., black-box testing). Test cases are generated to  
24 evaluate if the Statecharts model produces the correct data outputs. Input and expected  
25 output values are determined based on the information from the data dictionary and  
26 consistent with the SRS/Z schemas. This test assures that there are no inconsistent or  
27 unspecified data driven operations.

### 3.4. *Fault injection*

28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000

After injecting faults into the model, we observe the behavior to see 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 fault-tolerance using criticality analysis and priority basis.

#### 1 4. Application example 1

2  
3 In this section, an example is presented to explain how we applied the methods de- 3  
4 scribed above (Sheldon et al., 2001; Sheldon and Kim, 2002). This section shows one 4  
5 small part (i.e., ARSP submodule) of the larger NL-based GCS SRS that was trans- 5  
6 formed. 6

7 The selected Altitude Radar Sensor Processing (ARSP) submodule specification 7  
8 shows inputs, outputs, and subsystem processing descriptions. The SRS provides a 8  
9 data dictionary with variable definitions, type, and units, and a brief description of 9  
10 variables and functions. This module specification was abstracted into Z, preserving 10  
11 the variable names, operations (i.e., functionality), dependency and scope. Figure 2 11  
12 provides an example using the `FRAME_COUNTER` input variable that illustrate the 12  
13 complete translation from the SRS to Z and Statecharts. The top box in the Figure 2 13  
14 represents the SRS. The box in the middle of the Figure 2 represents the Z Speci- 14  
15 fication while the bottom box shows a part of the Statecharts model. In the SRS, 15  
16 the `FRAME_COUNTER` is defined as an integer with range  $[1, 2^{31} - 1]$ . In Z, the 16  
17 `FRAME_COUNTER` is declared as a set of natural numbers in the declaration part, and 17  
18 the range of the variable is represented in the predicate part (lower half of the schema). 18  
19 The Statecharts representation of the `FRAME_COUNTER` variable is presented with the 19  
20 direction of data transfer from EXTERNAL into the ARSP Module. Its type and value 20  
21 range are defined in the Statemate data dictionary. 21

22 In translating from the SRS to Z, four different ambiguous requirements were iden- 22  
23 tified. The first ambiguity committed leaves the rotational direction (i.e., left/right 23  
24 array shifting) undefined as it only indicates “rotate.” Second, an undefined third or- 24  
25 der polynomial was revealed used to estimate the `AR_ALTITUDE` value. The third 25  
26 ambiguity concerns the use of the `AR_COUNTER` variable for two different distinct 26  
27 purposes, which imply that it has two different types. Finally, there is uncertainty re- 27  
28 garding the scope of the `AR_COUNTER` variable and this brings into question which 28  
29 module should modify this variable. 29

30 Given these various issues, two scenarios were considered. The first scenario as- 30  
31 sumes the `AR_COUNTER` is updated within the ARSP module while the second sce- 31  
32 nario assumes that the `AR_COUNTER` is updated outside of the module. Both scenar- 32  
33 ios were constructed separately and compared to understand how Z could be useful in 33  
34 clarifying ambiguity and avoiding conflicts. 34

35 In this first scenario (Scenario One) to properly update `AR_COUNTER` within the 35  
36 ARSP, the two different purposes of the variable should be separated. Accordingly, 36  
37 the Z specification of the ARSP was defined to account for two separate variables 37  
38 (`AR_COUNTER` and Echo). This ensures that the `AR_COUNTER` represent only the 38  
39 pulse counter while Echo represent whether the radar echo pulse is received on time. 39  
40 The Z specification is consistent with the SRS as long as the newly introduced Echo 40  
41 variable does not cause a side effect. The Echo is treated as an additional ARSP input 41  
42 and in turn requires the specification to be revised to satisfy the data decoupling prin- 42  
43 ciple (Sommerville, 2000). The Scenario One interpretation is therefore inconsistent 43  
44 with the SRS. 44

45 Conversely, in Scenario Two (details described in Section 4.1) no additional vari- 45  
46 ables were defined. Only the variables defined in the SRS were modeled (as well as 46

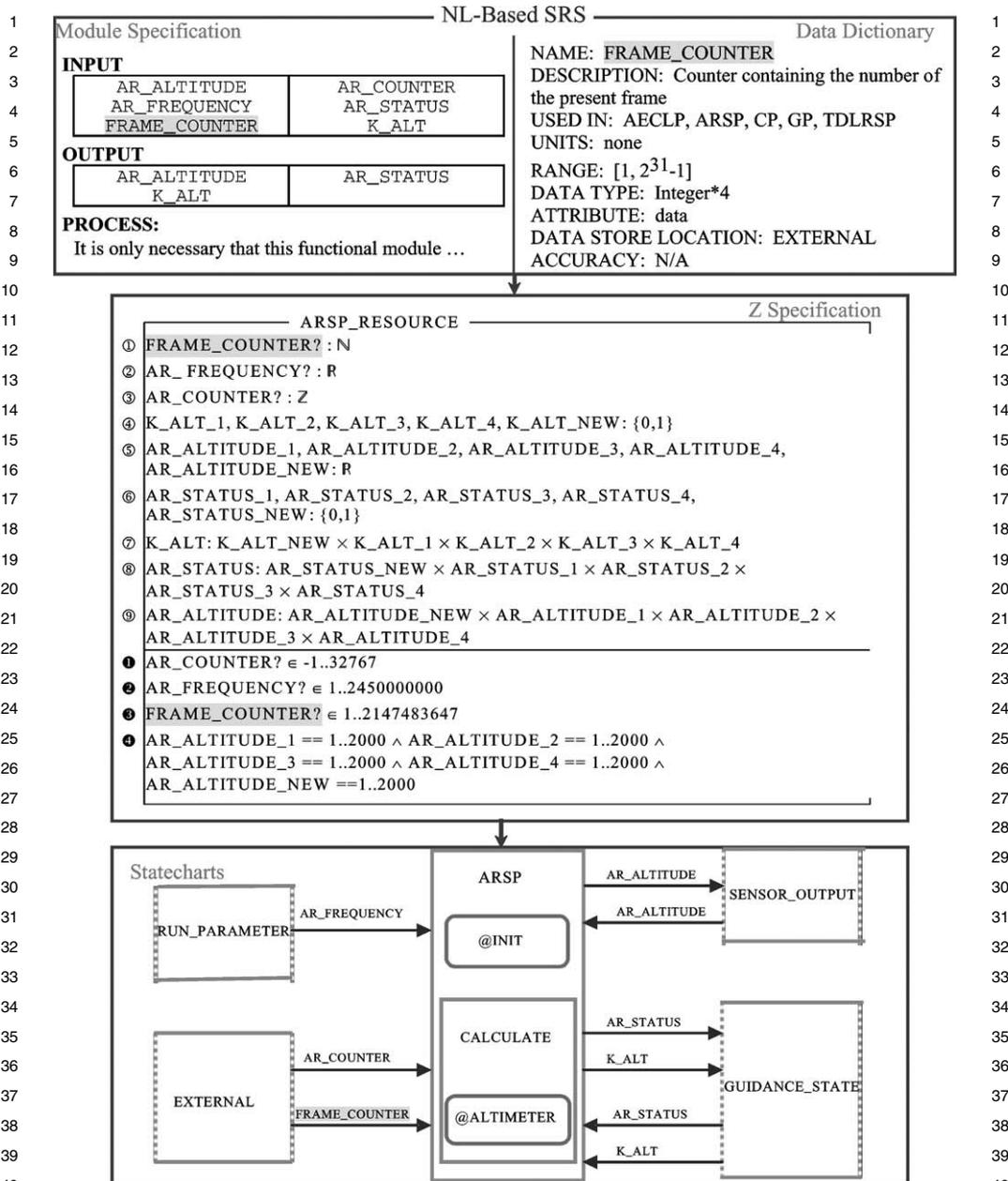


Figure 2. Translation example from NL-based to Statecharts.

covering all of the required ARSP behaviors). We considered this Z formulation to be complete and consistent with that of the SRS. 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 natural number. AR\_FREQUENCY? (Sig②) represents the rate at which the AR\_COUNTER? is incremented and its type is real. The AR\_COUNTER? (Sig③) is an input variable that is used to determine the AR\_ALTITUDE value and its type is 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 determined by the 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.

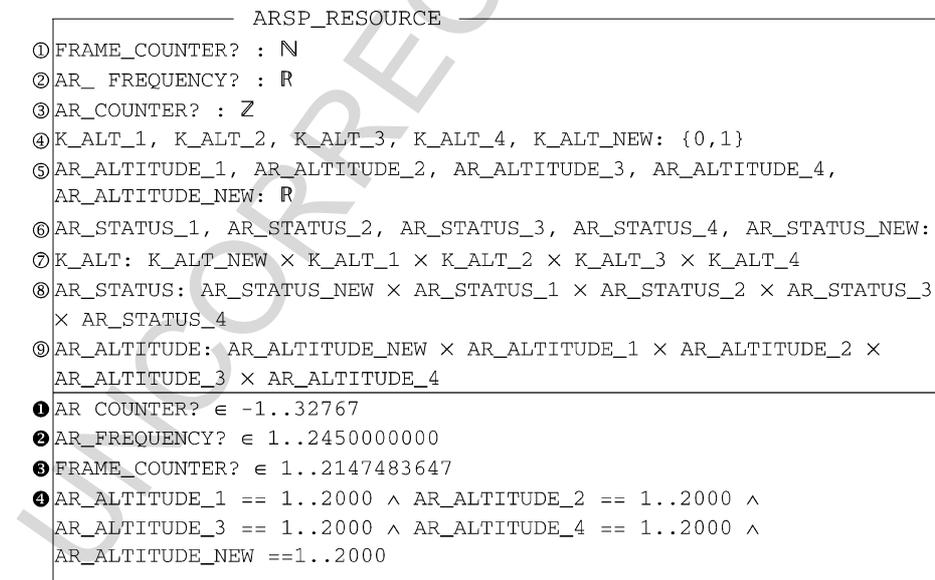


Figure 3. ARSP\_RESOURCE schema.

The array can also be represented as a 5-element sequence. The Cartesian product method was chosen because this composition assumes that any element can be accessed directly without having to search through the sequence. The predicates ①, ②, and ③ represent the variables ranges. The predicate ④ restricts the values for the sets in the Signature ⑤.

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 the 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 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 constrains the AR\_ALTITUDE update. The update takes the current value

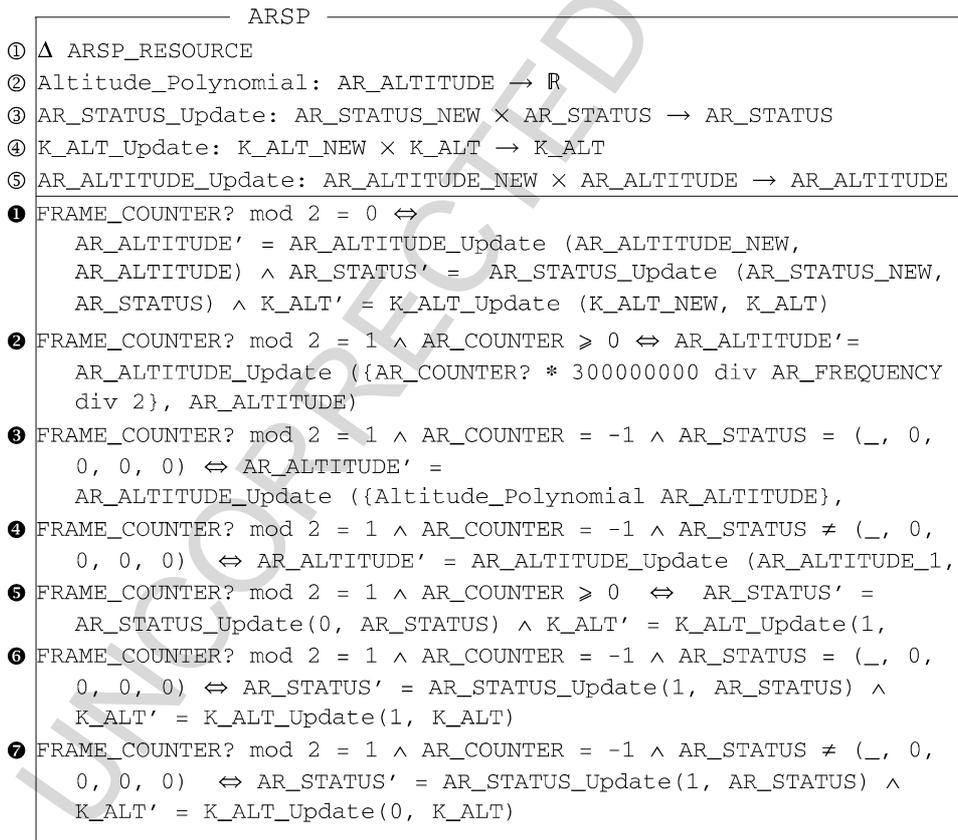


Figure 4. ARSP schema.

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.

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.

#### 4.2. Statecharts

The state/activity charts, derived from the Z specification are now described. Our intention here is to provide a straightforward discourse that precisely identifies the  $Z \rightarrow$  Statechart transformations and convinces the reader that such transformations are systematic and repeatable. We do not provide any explicit guidelines or rules here, which must therefore be inferred from the given example. It is important to understand that the transformations are heavily dependent on human skill.

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 precisely what is specified in the SRS data dictionary. 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.

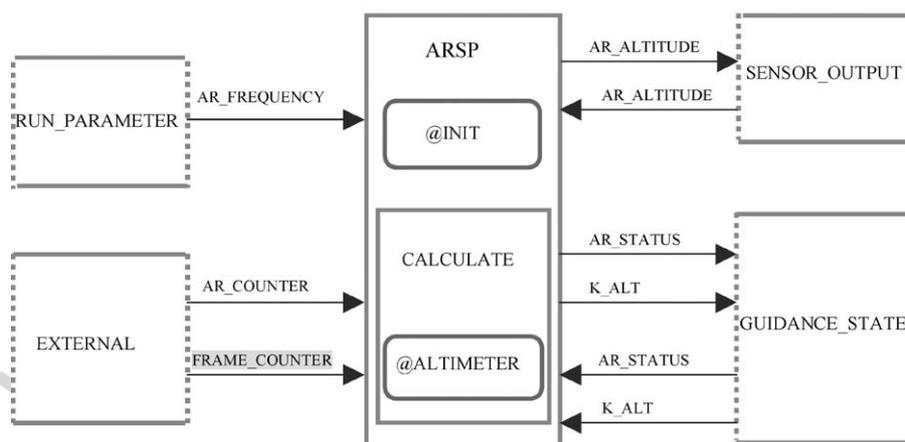


Figure 5. ARSP Activity chart.

1 The INIT Statechart (Figure 6) shows the initialization of the ARSP module and a  
 2 portion of the ARSP operational schema (Figure 4). The default transition activates  
 3 the CURRENT\_STATE when the ARSP activity (in the ARSP activity chart) is begun.  
 4 The transition from the CURRENT\_STATE state to KEEP\_PREVIOUS\_VALUE state  
 5 describe predicate ❶ of Figure 4. The KEEP\_PREVIOUS\_VALUE state is one of the  
 6 module termination states. The termination states are marked with “>” at the end of  
 7 the state name. The transition from the CURRENT\_STATE to the CALCULATION  
 8 state represent a condition where the value of FRAME\_COUNTER is odd, which is  
 9 described by the statement “FRAME\_COUNTER mod 2 = 1” in Figure 4.

10 The Altimeter Statechart (Figure 7) is represented by the “@ALTIMETER” control  
 11 activity in the ARSP activity chart (Figure 5). The ODD state is activated by the default  
 12 transition when the CALCULATION activity (in the ARSP activity chart) is begun.  
 13 The transition from the ODD state to the ESTIMATE\_ALTITUDE state occurs when  
 14 the AR\_COUNTER value is set to -1 and all the elements of the AR\_STATUS array  
 15 are set to “healthy.” When this transition begins, the AR\_STATUS and K\_ALT values  
 16 are updated as described by predicate ❸ of Figure 4. The 0 (zero) value of the  
 17 AR\_STATUS means “healthy” which corresponds to the value given in the SRS data  
 18 dictionary (NASA, 1993).

19 The transition from the ODD state to the CALCULATE\_ALTITUDE state begins  
 20 when the AR\_COUNTER is positive, which is equivalent to predicate ❺ of Figure 4.  
 21 The transition from the ODD to the KEEP\_PREVIOUS state is triggered when the  
 22 AR\_COUNTER value is set to -1 and at least one of the AR\_STATUS elements is  
 23 not healthy. This transition has the same meaning as predicate ❷ in Figure 4. The  
 24 transition from the ESTIMATE\_ALTITUDE state to the DONE state happens when  
 25

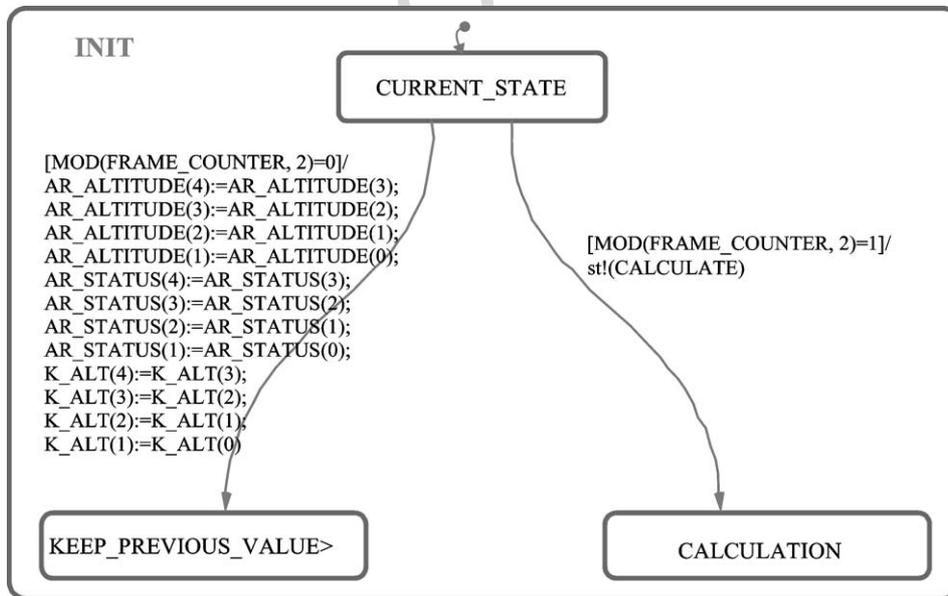


Figure 6. INIT Statechart.

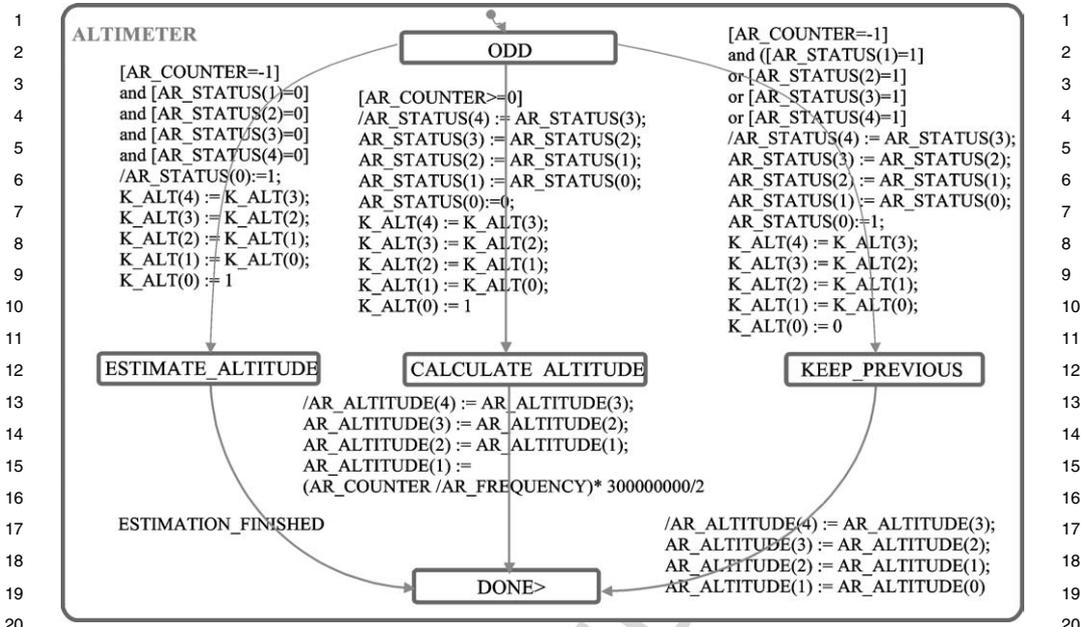


Figure 7. ALTIMETER Statechart.

the ESTIMATION\_FINISHED event occurs. This process is represented as an *event* because the transaction is described as an undefined third-order polynomial estimator in the SRS. The transaction from the CALCULATE\_ALTITUDE state to the DONE state denotes predicate ② (Figure 4). The transaction from the KEEP\_PREVIOUS state to the DONE state denotes the predicate ④ (Figure 4) operation.

#### 4.3. Specification testing

The Statechart models described here and in the appendix are checked for completeness and consistency using symbolic simulation. Two specification test results (using the approaches described in Section 3.3) are presented here.

**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

Table 2. ARSP specification simulation result

Name of chart	Activity/State name	Activity/State transition paths			
		1	2	3	4
ARSP	ARSP	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>
	@INIT	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>
	CALCULATE	-	E <sub>5</sub>	E <sub>5</sub>	E <sub>5</sub>
	@ALTIMETER	-	E <sub>6</sub>	E <sub>6</sub>	E <sub>6</sub>
INIT	CURRENT_STATE	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>
	KEEP_PREVIOUS_VALUE>	E <sub>4</sub>	-	-	-
	CALCULATION	-	E <sub>4</sub>	E <sub>4</sub>	E <sub>4</sub>
ALTIMETER	ODD	-	E <sub>7</sub>	E <sub>7</sub>	E <sub>7</sub>
	ESTIMATE_ALTITUDE	-	E <sub>8</sub>	-	-
	CALCULATE_ALTITUDE	-	-	E <sub>8</sub>	-
	KEEP_PREVIOUS	-	-	-	E <sub>8</sub>
	DONE>	-	E <sub>9</sub>	E <sub>9</sub>	E <sub>9</sub>

E<sub>i</sub>: entered in *i*th order, -: not activated.

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]
Expected output	AR_COUNTER	-1	19900	-1	20000	-1
	AR_STATUS	KP	KP	[1, 0, 0, 0, 0]	[0, -, -, -, -]	[1, 0, 0, 1, 0]
Actual output	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.

absorbing states or activities and the module is complete indicating that the SRS is complete (at least for the ARSP submodule).

**4.3.2. Data item approach.** Five test cases (Cases 1–5) are shown in Table 3 to probe the Statecharts. They represent how the Z schemas were dynamically visualized and evaluated. The input/output values are calculated based on the SRS equations. 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 shown in Figure 7). The AR\_FREQUENCY variable is defined as a real number with a large range. Accordingly, AR\_FREQUENCY 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. Tables 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 Case 1, for example,

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.

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.

input variables for the 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 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.

#### 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 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. 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 (Sheldon and Kim, 2002)); (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:  $1 \leq \text{AR\_FREQUENCY} \leq \text{AR\_COUNTER} * 75000$ , or  $\text{AR\_COUNTER} = -1 \vee (0 \leq \text{AR\_COUNTER} \leq \text{AR\_FREQUENCY}/75000)$ . In other words, one of these two relational expressions must evaluate as true.

## 5. Summary and conclusion

This paper discusses the methods and results as they relate to the ARSP (Altimeter Radar Sensor Processing) submodule, which was part of the larger total system specification. The complete study covered additional submodules as well as the overall structure of the GCS (i.e., other submodules were stubbed off). The submodules analyzed include the ARSP, GP (Guidance Processing), RECLP (Roll Engine Control Law Processing), and CP (Communication Processing). The choice of these submodules was made based on the GCS run-time schedule consisting of three major subframes: (1) sensor processing, (2) guidance processing, and (3) control law processing. One functional unit was chosen from each of the subframes while the CP, which runs in the guidance-processing subframe, was chosen due to its unique functional task.

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 system performability. 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 rework.

The results showed some portions of the GCS SRS to be 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 a pertinent analysis of the problem—i.e., operational errors between states, functional defects, lack of such properties such as fault tolerance, etc.

This paper represents a significant result—it demonstrates these conjoined methods on one real system. Our intention has been to provide a simple straightforward discourse that precisely identifies the NL-specification  $\rightarrow$  Z  $\rightarrow$  Statechart transformations in such a way that shows how these transformations are systematic and repeatable. Explicit or generalizable guidelines/rules are not provided, which therefore must be inferred from the given example. The transformations are heavily dependent on human skill. Consequently, we hope to continue our work with the goal of producing a systematic generalizable approach that can be readily applied to other problems (i.e., a variety of systems) and demonstrate the process empirically.

## Acknowledgment

We would like to especially thank Kelly Hayhurst at NASA LaRC Formal Methods Group for providing the actual NL-based Viking Mars Lander GCS specification (dur-

1 ing Dr. Sheldon's Post-doc), Kshamta Jerath (Ph.D. student now at Microsoft) for 1  
2 her valuable through critiques, Markus Degen and Stefan Greiner (DaimlerChrysler 2  
3 System Safety [RIC/AS Stuttgart]) for reinforcing the need for good specifications es- 3  
4 pecially in critical applications (as well as funding), and the Software Quality Journal 4  
5 reviewers for their valuable critiques and encouragement. 5  
6

## 7 **Appendix<sup>5</sup>** 7

8 **Appendix<sup>5</sup>** 8  
9 9  
10 The Guidance and Control Software (GCS) principally provides control during the ter- 10  
11 minal phase of descent for the Viking Mars Lander. The Lander has three accelerome- 11  
12 ters, one Doppler radar with four beams, one altimeter radar, two temperature sensors, 12  
13 three gyroscopes, three pairs of roll engines, three axial thrust engines, one parachute 13  
14 release actuator, and a touch down sensor. After initialization, the GCS starts sensing 14  
15 the vehicle altitude. When a predefined engine ignition altitude is sensed, the GCS 15  
16 begins guidance and control of the vehicle. The purpose of this software is to main- 16  
17 tain the vehicle along a predetermined velocity-altitude contour. Descent continues 17  
18 along this contour until a predefined engine shut off altitude is reached or touchdown 18  
19 is sensed. 19

20 Figure A.1 shows the overall system architecture of the GCS software. The circled 20  
21 parts are the subunits consisting of the partial specification for this case study. The 21  
22 partial specification that was examined includes one sensor-processing unit, one actu- 22  
23 ator unit, and the two core subunits of the GCS system (circled units in Figure A.1). 23  
24 All other subunits are ignored in this case study except the data stores. Control and 24  
25 data flows between the excerpted modules are the same as they are represented in the 25  
26 Module chart (Figure A.2). 26

27 The choice of parts for this study is made based on its run-time schedule (Table A.1). 27  
28 The GCS has a predetermined running time frame that consists of three subframes. 28  
29 Each subframe has specific submodules to run. The partial specification in this study 29  
30 consists of one submodule from each subframe and a submodule that runs every sub- 30  
31 frame. ARSP (Altimeter Radar Sensor Processing) is running in the first subframe, 31  
32 GP (Guidance Processing) is running in the second subframe, and RECLP (Roll En- 32  
33 gine Control Law Processing) is running in the third subframe. CP (Communication 33  
34 Processing) is running in every subframe. In SRS, CP is specified as the last submod- 34  
35 ules to run for every subframe. The order of the submodules in the same subframe is 35  
36 not declared except CP must run last. 36

37 The ARSP (Altimeter Radar Sensor Processing) is a sensor-processing submodule 37  
38 of the GCS. This functional unit reads the altimeter counter provided by the altimeter 38  
39 radar sensor and converts the data into a measure of distance to the surface of Mars. 39  
40 The CP is a submodule that converts the sensed data into a data packet appropriate 40  
41 for radio transformation. The data packets are relayed back to the orbiting platform 41  
42 for relay to Earth. The GP (Guidance Processing) is the core-processing submodule 42  
43 of the GCS. This module gathers the information from the entire sensor processing 43  
44 subunits and the previous computational results. Then, it manages the vehicle's state 44  
45 during the descent by controlling the actuators. The RECLP (Roll Engine Control Law 45  
46 Processing) is an actuator unit that computes the value settings for three roll engine. 46

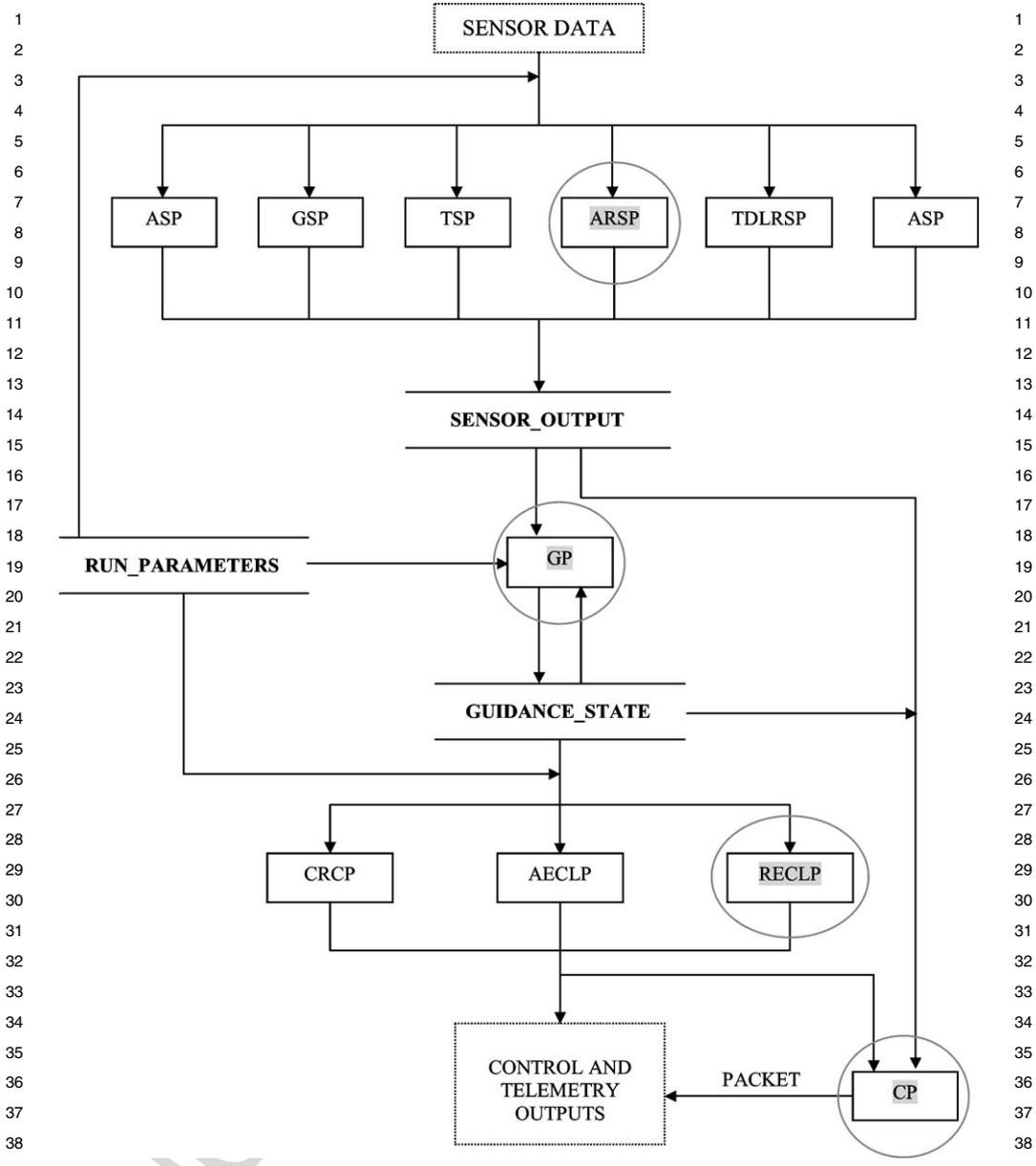


Figure A.1. GCS system structure.

The roll engine value settings are calculated to fix the difference between the vehicle's measured values during operation and the designated trajectory values.

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.

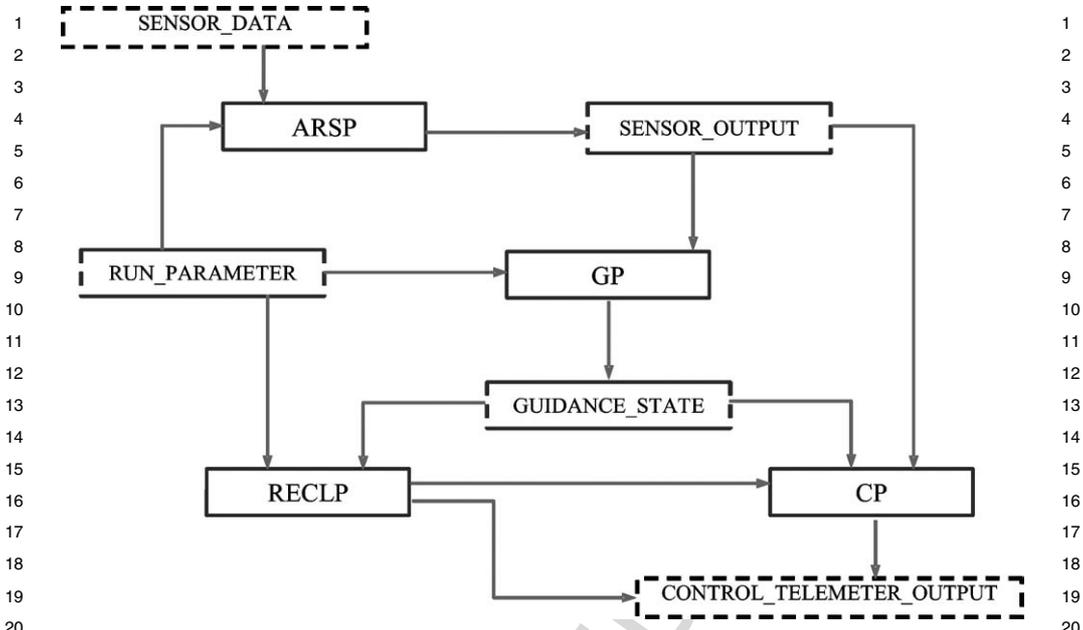


Figure A.2. A module chart of GCS excerpt.

Table A.1. Functional unit schedule (NASA, 1993)

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

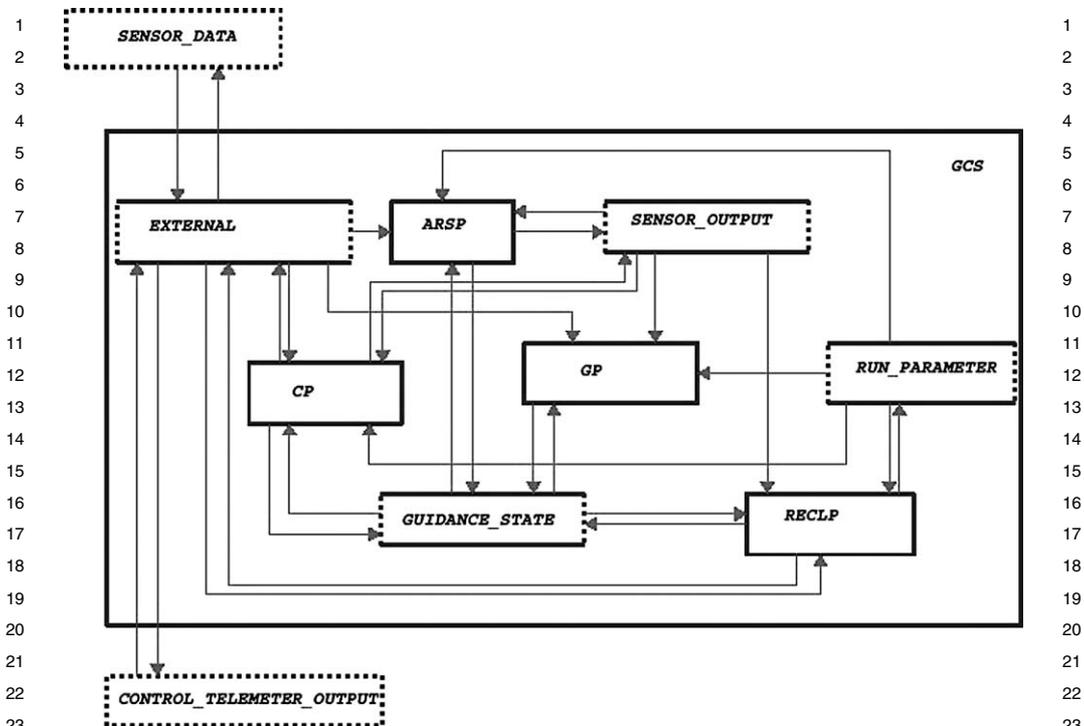


Figure A.3. Actual module chart of the GCS excerpt.

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.

The GCS\_CONTROL statechart (Figure A.5) represents the GCS\_CONTROL schemas. The default transition represents 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 (Figures A.6–A.8) linked to the superstate.

Figures A.9 and 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 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 chapter 5 of the thesis (accessible by downloading from [http://www.eecs.wsu.edu/seds/hkim\\_thesis\\_final\\_ilogix.pdf](http://www.eecs.wsu.edu/seds/hkim_thesis_final_ilogix.pdf)).

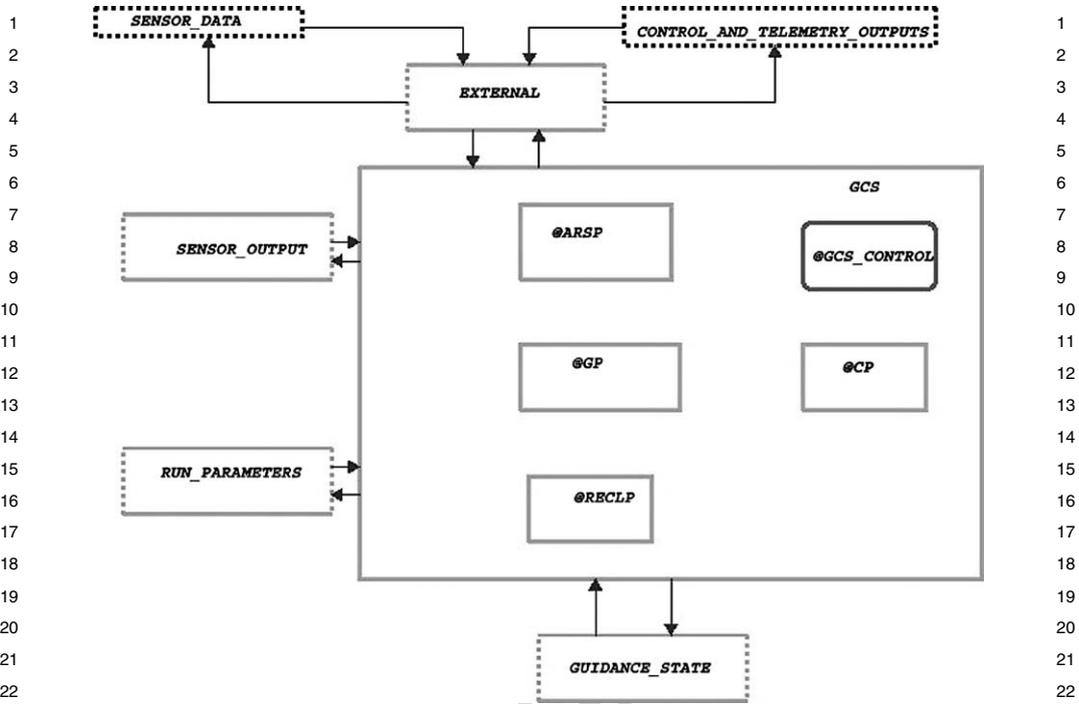


Figure A.4. GCS Activity chart.

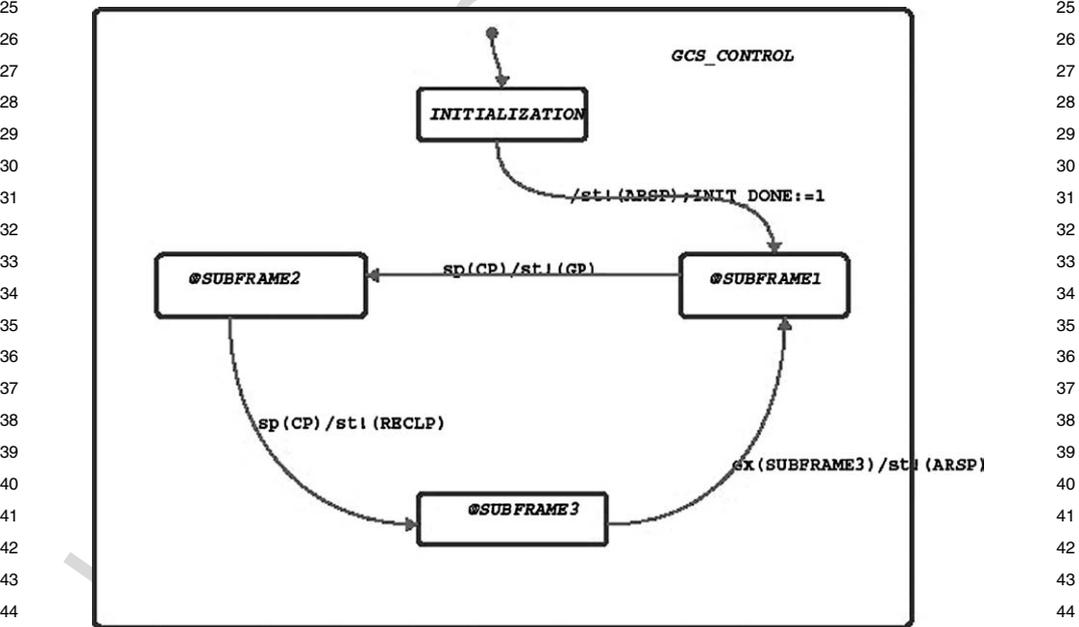


Figure A.5. GCS\_CONTROL Statechart.

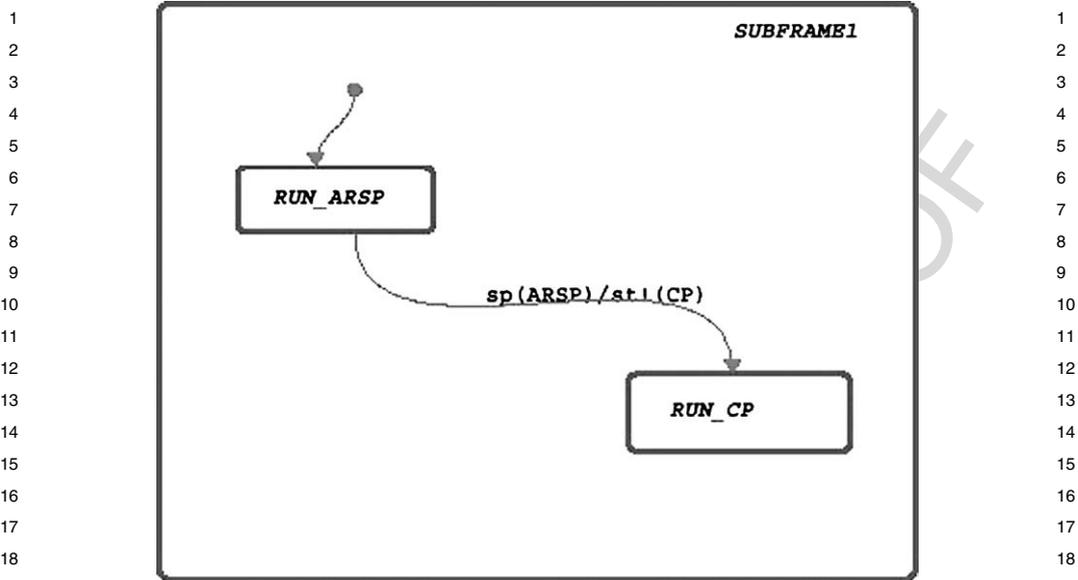


Figure A.6. SUBFRAME1 Statechart.

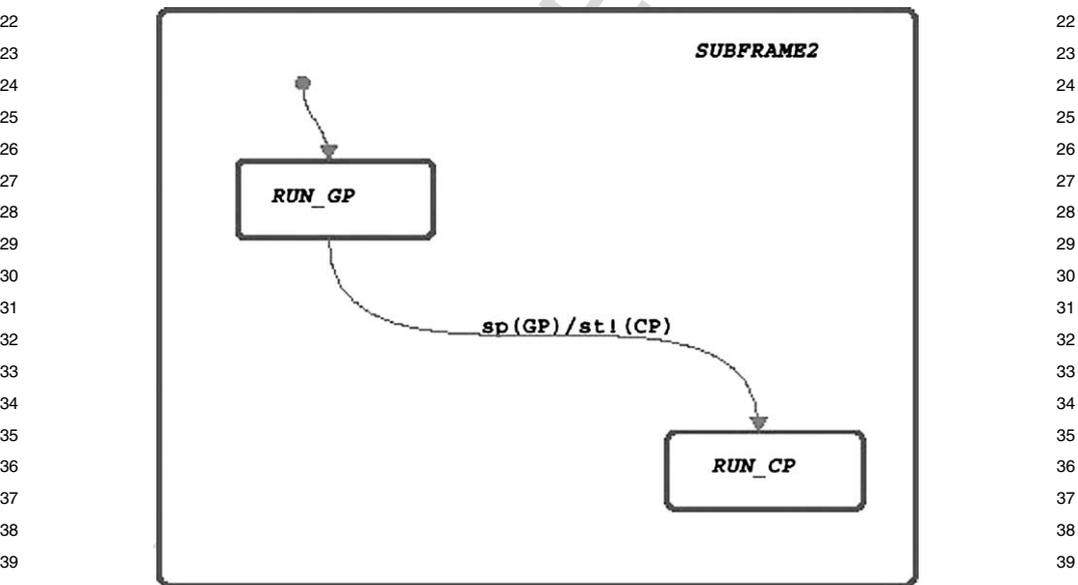


Figure A.7. SUBFRAME2 Statechart.

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

The test results using DIA are the same as shown in the Section 4.3.2. The fault injection results are described in Table A.3.

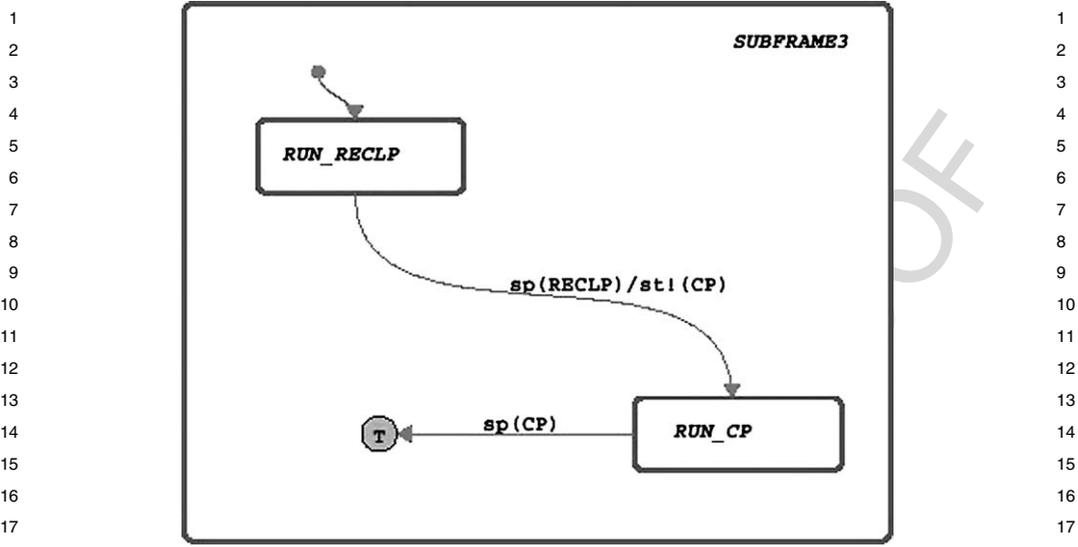


Figure A.8. SUBFRAME3 Statechart.

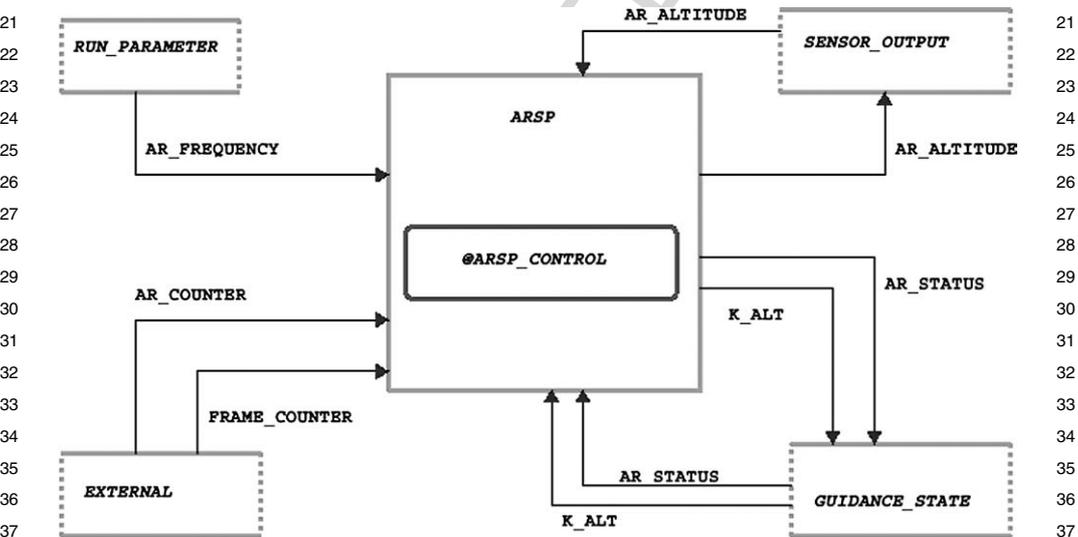


Figure A.9. ARSP Activity chart.

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 (Figures 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

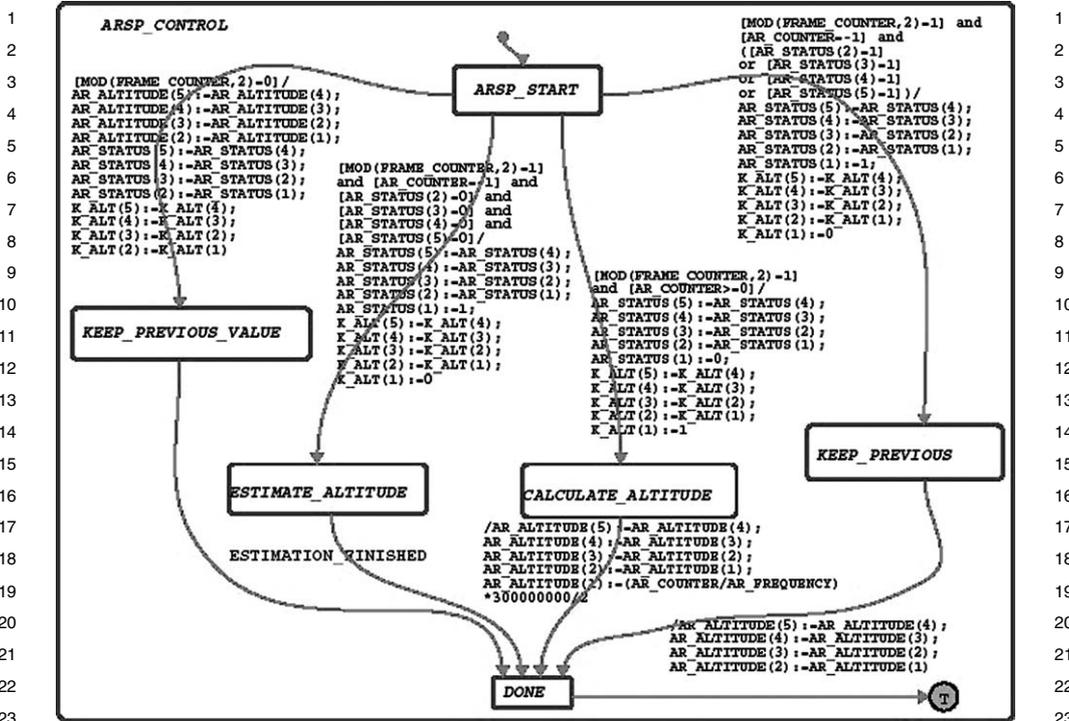


Figure A.10. ARSP\_CONTROL Statechart.

Table A.2. ARSP specification simulation results

Name of chart	Activity/State name	Activity/State transition paths			
		1	2	3	4
ARSP	ARSP	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>
	@ARSP_CONTROL	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>
ARSP_CONTROL	ARSP_START	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>
	KEEP_PREVIOUS_VALUE	E <sub>4</sub>	-	-	-
	ESTIMATE_ALTITUDE	-	E <sub>4</sub>	-	-
	CALCULATE_ALTITUDE	-	-	E <sub>4</sub>	-
	KEEP_PREVIOUS	-	-	-	E <sub>4</sub>
	DONE	E <sub>5</sub>	E <sub>5</sub>	E <sub>5</sub>	E <sub>5</sub>

E<sub>i</sub>: entered in *i*th order, -: not activated.

(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 (Figures 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

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	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

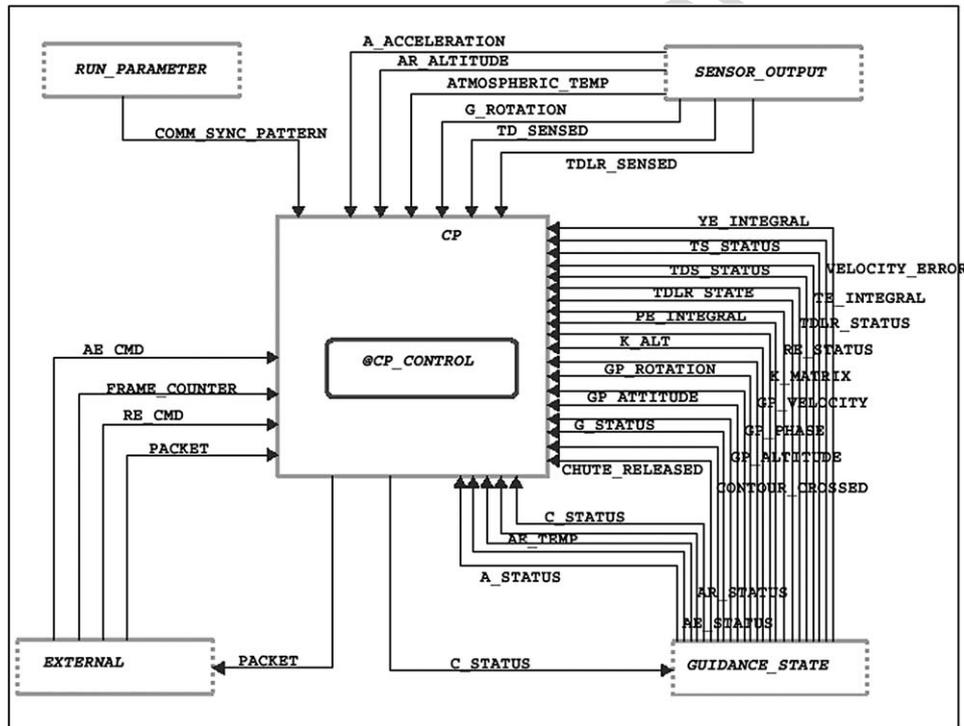


Figure A.11. CP Activity chart.

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 sub-module (Figures A.15 and A.16). The test results are as presented in Tables A.4 and A.6–A.9. Table A.5 shows the system constants for the simulation.

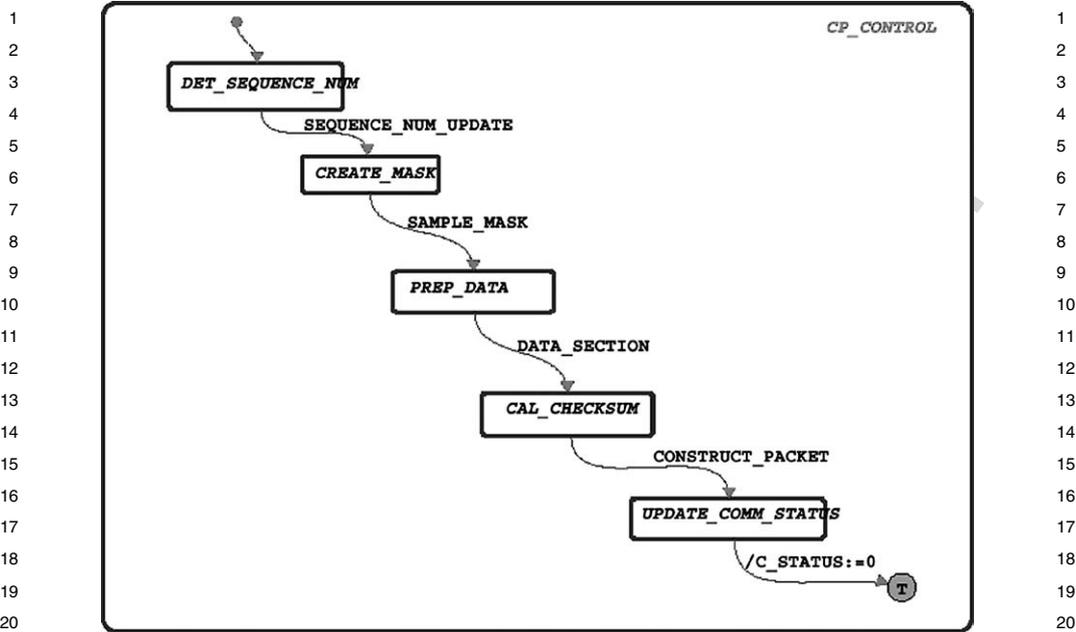


Figure A.12. CP\_CONTROL Statechart.

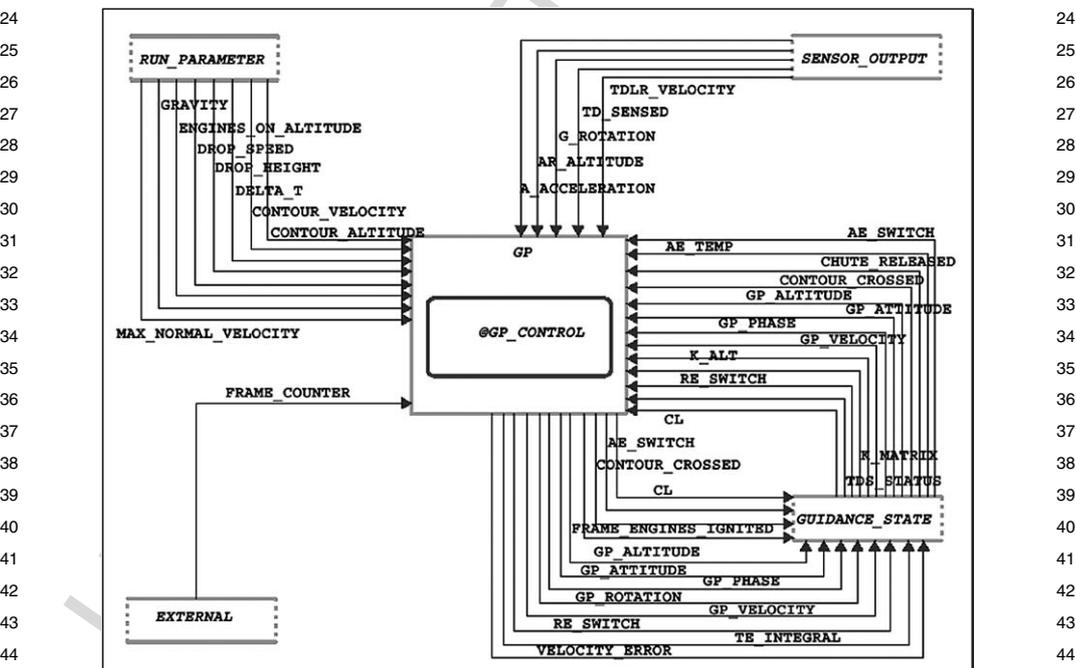


Figure A.13. GP Activity chart.

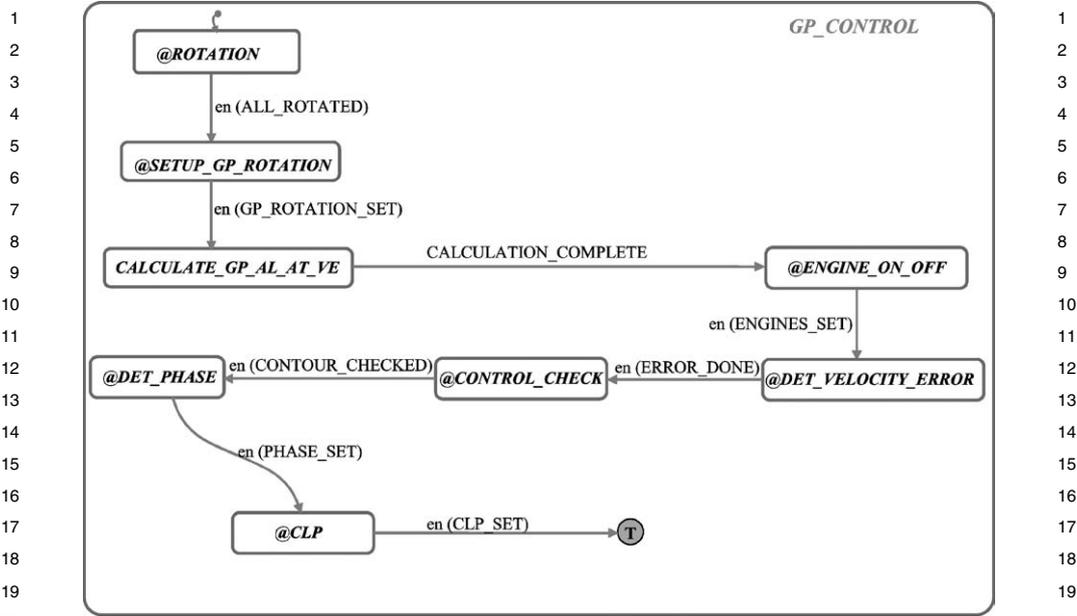


Figure A.14. GP\_CONTROL Statechart.

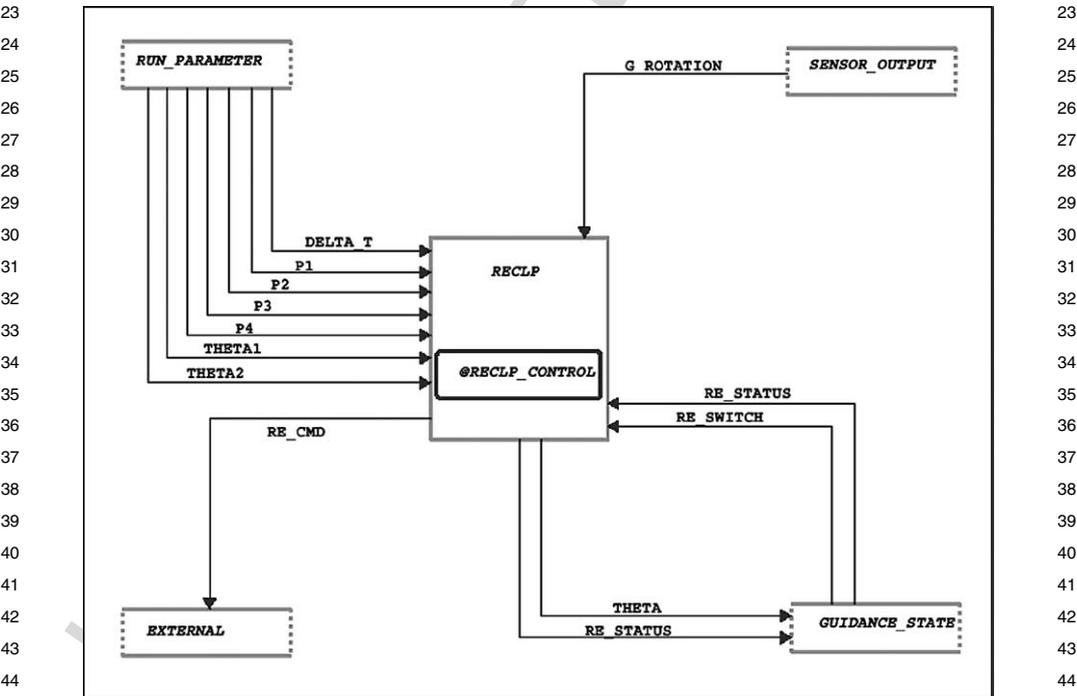


Figure A.15. RECLP Activity chart.

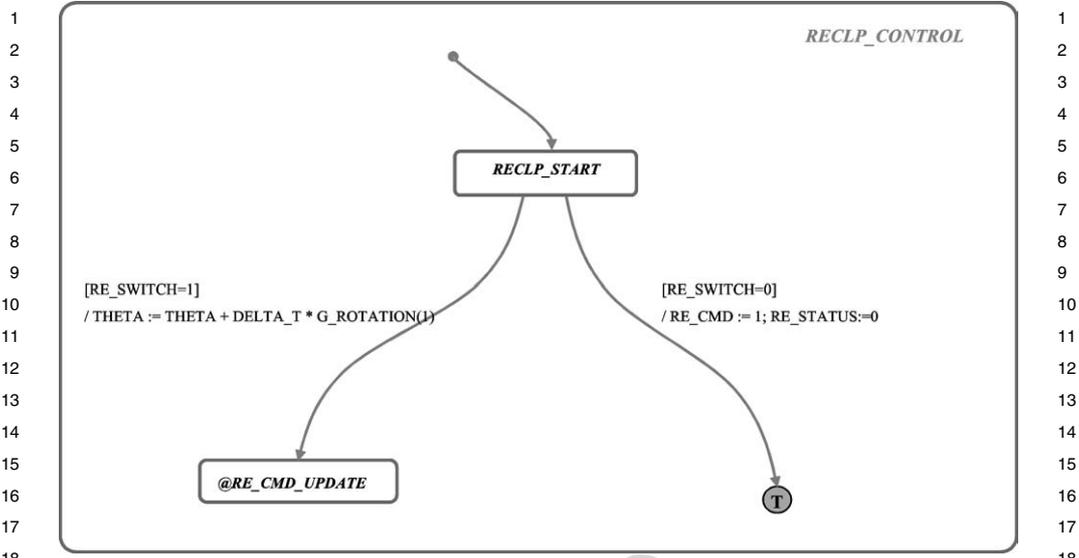


Figure A.16. RECLP\_CONTROL Statechart.

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 <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>	E <sub>1</sub>
	@RECLP_CONTROL	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>	E <sub>2</sub>
RECLP_CONTROL	RECLP_START	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>	E <sub>3</sub>
	@RE_CMD_UPDATE	-	E <sub>4</sub>						
RE_CMD_UPDATE	SET_RE_CMD	-	E <sub>5</sub>						
	RE_CMD1	-	E <sub>6</sub>	-	-	-	-	-	-
	RE_CMD2	-	-	E <sub>6</sub>	-	-	-	-	-
	RE_CMD3	-	-	-	E <sub>6</sub>	-	-	-	-
	RE_CMD4	-	-	-	-	E <sub>6</sub>	-	-	-
	RE_CMD5	-	-	-	-	-	E <sub>6</sub>	-	-
	RE_CMD6	-	-	-	-	-	-	E <sub>6</sub>	-
RE_CMD7	-	-	-	-	-	-	-	E <sub>6</sub>	

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

## Notes

1. The correctness of an SRS cannot be proven in the strict sense used in verification. In the formal definition, a software artifact is correct with respect to another artifact if a well defined relationship exists between the artifacts. The obvious question arises, what can an SRS be proven correct against? We use the terms correctness and accuracy to convey the more general meaning against user/system/mission needs. The system must complete the mission and therefore, the software must continue to function reliably. Our analyses seek to ensure that the SRS unequivocally support the successful mission completion.
2. STATEMATE Magnum—product of i-Logix, was used to conduct the research for this thesis.
3. *Completely* in this context means that each and every variable and function associated with that particular section of the NL-based SRS has been represented/denoted using the respective formalism.
4. Z/EVES is available from ORA, Canada. It provides theorem proving, domain checking, type checking, precondition calculation, and schema expansion for Z specifications.
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](http://www.eecs.wsu.edu/seds/hkim_thesis_final_ilogix.pdf) or the <http://www.ilogix.com> university page.

## References

- Bogdanov, K. and Holcombe, M. 2001. Statechart testing method for aircraft control systems, *Software Testing, Verification & Reliability* 11(1): 39–54.
- Bussow, R., Geisler, R., and Klar, M. 1998. Specifying safety-critical embedded systems with Statecharts and Z: A case study, *Lecture Notes in Computer Science*, Vol. 1382.
- Bussow, R. and Weber, M. 1996. A steam-boiler control specification with Statecharts and Z, *Lecture Notes in Computer Science*, Vol. 1165.
- Castello, R. 2000. *From Informal Specification to Formalization: an Automated Visualization Approach*, PhD dissertation in computer science, University of Texas at Dallas.
- Czerny, B. 1998. *Integrative Analysis of State-Based Requirements for Completeness and Consistency*, PhD dissertation in computer science, Michigan State University.
- Damm, W., Hungar, H., Kelb, P., and Schlor, R. 1995. Statecharts—using graphical specification languages and symbolic model checking in the verification of a production cell, *Lecture Notes in Computer Science*, Vol. 891.
- Fabbrini, F., Fusani, M., Gnesi S., and Lami, G. 2001. An automatic quality evaluation for natural language requirements, *7th Int. Workshop on Req. Eng.: Foundation for SW Quality (REFSQ)*, [www.ifi.uib.no/conf/refsq2001/papers/p3.pdf](http://www.ifi.uib.no/conf/refsq2001/papers/p3.pdf). Accessed on Mar. 25, 2002.
- Gaudel, M.-C. and Bernot, G. 1999. The role of formal specifications, *IFIP State-of-the-Art Report: Algebraic Foundations of Sys Spec.*, eds. E. Astesiano, H.-J. Kreowski, and B. Krieg-Bruckner. Springer.
- Grieskamp, W., Heisel, M., and Dorr, H. 1998. Specifying embedded systems with Statecharts and Z: An agenda for cyclic software components, *Lecture Notes in Computer Science*, Vol. 1382.
- Harel, D. 1987. Statecharts: A visual formalism for complex systems, *Science of Computer Programming* 8: 231–274.
- Harel, D. and Politi, M. 1998. *Modeling Reactive Systems with Statecharts*. McGraw-Hill.
- He, X. 2001. PZ nets—a formal method integrating Petri nets with Z, *Information and Software Technology* 43(1): 1–18.
- Heimdahl, M.P.E. and Leveson, N.G. 1996. Completeness and consistency in hierarchical state-based requirements, *IEEE Transactions on Software Engineering* 22(6): 363–377.
- Heitmeyer, C., Kirby, Jr., J., Labaw, B., Archer, M., and Bharadwaj, R. 1998. Using abstraction and model checking to detect safety violations in requirements specification, *IEEE Transactions on Software Engineering* 24(11): 927–948.
- Hierons, R.M., Sadeghipour, S., and Singh, H. 2001. Testing a system specified using Statecharts and Z, *Information and Software Technology* 43: 137–149.
- Kotonya, G. and Sommerville, I. 1998. *Requirements Engineering: Process and Techniques*. New York, Wiley.
- Leveson, N. 1995. *Safeware—System Safety and Computers*. Reading, MA, Addison Wesley.
- NASA. 1993. *Software Requirements—Guidance and Control Software Development Specification Ver 2.2 with the formal mods 1–8*. NASA, Langley Research Center.
- Pradhan, D. K. 1996. *Fault-Tolerant Computer System Design*, pp. 428–477. Prentice Hall.
- Sannella, D. and Tarlecki, A. 1999. Algebraic preliminaries, *IFIP State-of-the-Art Reports: Algebraic Foundations of Sys Spec.*, eds. E. Astesiano et al. Springer.
- Shaw, A.C. 2001. *Real-Time Systems and Software*. New York, Wiley.
- Sheldon, F.T. and Kim, H.Y. 2002. Validation of guidance control software requirements specification for reliability and fault-tolerance. *Proc. of Annual Reliability and Maintainability Symp.* Seattle, WA, pp. 312–318. IEEE.
- Sheldon, F.T., Kim, H.Y., and Zhou, Z. 2001. 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 Computing*, Seoul, Korea, pp. 311–318. IEEE Computer Society.
- Sommerville, I. 2000. *Software Engineering*, 6th ed. Reading, MA, Addison-Wesley.
- Vliet, H.V. 2000. *Software Engineering: Principles and Practice*. New York, Wiley.

1	Voas, J., McGraw, G., Kassab, L., and Voas, L. 1997. A crystal ball for software liability, <i>IEEE Computer</i> 30(6):	1
2	29–36.	2
3	Woodcock, J. and Davies, J. 1996. <i>Using Z: Specification, Refinement, and Proof</i> , Series of Computer Science.	3
4	Prentice-Hall.	4
5		5
6		6
7		7
8		8
9		9
10		10
11		11
12		12
13		13
14		14
15		15
16		16
17		17
18		18
19		19
20		20
21		21
22		22
23		23
24		24
25		25
26		26
27		27
28		28
29		29
30		30
31		31
32		32
33		33
34		34
35		35
36		36
37		37
38		38
39		39
40		40
41		41
42		42
43		43
44		44
45		45
46		46