SEQUENCE-BASED SPECIFICATION OF CRITICAL SOFTWARE SYSTEMS

Stacy J. Prowell and W. Thomas Swain
The University of Tennessee
203 Claxton Complex, 1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
Email: sprowell@cs.utk.edu

Keywords: Software specification, formal methods, rigorous specification.

ABSTRACT

The plant safety and regulatory requirements for software-based systems mandate rigorous verification and validation to ensure adequate reliability. As software systems become increasingly complex, testing as the sole means to assure confidence in the end product becomes impractical. Modern software engineering provides many tools which have been used successfully to reason about software, including formal systems such as Z and CSP. Many such formal notations have tools to support their use. The use of these notations and tools requires specialized training in software engineering methods. Sequence-based software specification techniques provide a connection between the initial requirements obtained from the domain experts and the formal software derivations, which can be in any appropriate notation.

1. INTRODUCTION

Application of software-based systems to plant control and protection functions offers the potential for improved measurement accuracy and operational flexibility. Additionally, for many applications today it is difficult to find instruments or controllers that do not contain microprocessors. The plant safety and regulatory requirements for such systems mandate rigorous verification and validation to ensure adequate reliability, especially since software can be a source of common mode failure. As software systems become increasingly complex, testing as the sole means to assure a desired level of confidence in the end product becomes impractical. Thorough documentation and intensive review of the transition from requirements to code are labor intensive and still do not guarantee proper operation. It is therefore necessary to focus strongly on correctness in the specification and design phase and to rely on the foundational mathematics of software to prove important properties.

Modern software engineering provides many tools which have been used successfully to reason about software, including formal systems such as Z (Spivey 1992), SCR (Heitmeyer 1983, 1995), and Trace Assertion Method (Bartussek 1978, Janicki 2001), and process algebras such as CCS (Milner 1989) and CSP (Hoare 1985, Roscoe 1997). Many of these notations have supporting tools. These include tools for checking properties of the specification; theorem provers to assist in proving safety, security; and other important system properties, and code generation tools to generate correct source code directly from the formal specifications.
The use of these notations and tools requires specialized training. Application domain experts cannot be expected to learn these notations and tools; this is the province of the software engineer. Unfortunately, this can create a disconnect between the domain experts who understand how the system must perform, and the software professionals who understand how to construct provably correct software systems.

Sequence-based software specification techniques (Prowell 2003) provide a connection between the initial requirements obtained from the domain experts and the formal software derivations which can be in any appropriate notation. The core techniques, sequence enumeration and sequence abstraction, can be used to transform the initial requirements into an appropriate formal notation for rigorous verification and semi-automated implementation (Broadfoot 2003). Sequence enumeration involves the deliberate, systematic consideration of each possible scenario of use and the assignment of the appropriate software response. Each decision is traced to the initial requirements. This process is made practical by the introduction of sequence abstractions, which allow practitioners to work at higher levels of abstraction without losing completeness and consistency.

This paper provides an overview and example of the sequence-based specification process and the transformation of the result into an external, implementation-free description of software behavior.

2. SEQUENCE SPECIFICATION

Sequence enumeration is a systematic, rigorous method of iteratively discovering a complete, consistent, and traceably correct system specification from initial requirements. Initial system requirements are typically written in a combination of natural language and mathematical notation. These requirements are seldom written in a formal notation suitable for automated reasoning, because such a notation may actually obscure the very details the domain experts find most interesting.

Only a short overview of sequence-based specification is presented here; for more details, see (Prowell 2003). For a larger case study which follows the process through coding and the development of a test plan, see (Prowell 1999).

2.1. Example

To illustrate sequence-based specification, this paper will present highlights of the method as it would be applied to a hypothetical PWR Thermal Margin Calculator (TMC). Table 1 shows a simplified list of requirements for the TMC, as they might be expressed at the beginning of system development, regardless of the design method used.
<table>
<thead>
<tr>
<th>Tag</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The TMC shall compute an accurate, but conservative, static estimate of DNBR every 100 ms. Sensor inputs for the DNBR calculation are pressurizer pressure ( p ), neutron flux at selected locations ( \phi_i ), primary coolant mass flow rate ( F ), cold leg temperature ( T_c ), and hot leg temperature ( T_h ).</td>
</tr>
<tr>
<td>2</td>
<td>The DNBR, denoted ( D_d ), is compared to a trip setpoint, ( D_t ), each time it is computed. If ( D_d ) is less than the setpoint, then the trip contact output is set.</td>
</tr>
<tr>
<td>3</td>
<td>Sensor inputs ( p ), ( \phi_i ), ( F ), ( T_c ), and ( T_h ) are checked for validity when sampled. If a sensor value is invalid, the channel failure indicator is set.</td>
</tr>
</tbody>
</table>
| 4   | The operator interface has the following features:  
  - Numeric displays for the point ID and value of sensor inputs, calculated values, calibration constants, and setpoints.  
  - An alphanumeric display of the current point ID.  
  - A numeric keypad and associated function keys for entry of point ID selections and new values for calibration constants and setpoints.  
  - Indicator lights for trip status and channel failure status.  
  - A keyswitch to turn power on and off. |
| 5   | If a non-existent or invalid point ID is entered, the point ID and value displays revert to their previous values. |
| 6   | If an invalid calibration or setpoint value is entered, the value display reverts to the original value. |
| 7   | A watchdog timer must be reset at least every 100 ms. Otherwise the channel failure indication is set and a reboot command is issued. |
| 8   | When a TMC is powered up or rebooted, all sensor inputs are read and used to initialize the DNBR calculation, the watchdog timer is reset, and the trip contact output is set. |
| 9   | If a TMC is powered down or rebooted the trip contact output must be set. |

These requirements imply a software system involving a real time executive with preemptive task scheduling. Figure 1 shows a conceptual partitioning of the system into the executive and a few modules dedicated to computations or handling I/O interfaces. This partitioning reflects the expected system architecture and helps simplify the design task.
2.2. System Boundary, Stimuli, and Outputs

The first step in sequence-based specification is to construct the list of external interfaces to the system being specified. The system may interface with other hardware and software components, or with the end user. This list of interfaces is called the system boundary, and it delimits the problem to be solved. All components which interface directly with the system being specified comprise the system’s environment. For the TMC the executive is to be developed. Thus the system boundary is as shown in Figure 1, with the interfaces indicated, and with all other components in the executive’s environment.

Next, each interface is considered and stimuli are identified. A stimulus is some event in the system's environment which is observed by the executing software, and a response is any system behavior which is observable from the environment. A sequence of stimuli therefore represents the history of use of the system. All information required by the software to generate responses must either be stored in the software itself, or it must come from stimuli. Thus it is possible to write down, for each possible stimulus sequence, the appropriate next response. For a hand calculator with stimulus history “ON 1 2 + 2 =,” the appropriate next response is to display 14.
The system stimuli and outputs are given in Tables 2 and 3. The stimuli for displaying point IDs and modifying calibration constants and setpoints are actually abstract stimuli. An *abstract stimulus* is a combination of a prior history condition and an atomic event, and is used to change the view of the stimulus sequence during enumeration. See Prowell (2003) for a formal definition. One or more outputs are combined to form a single response. As with stimuli the two outputs related to the operator interface are abstractions of the specific values to be displayed.

### 2.3. Sequence Enumeration

Sequence enumeration is performed by writing down each stimulus sequence, in order by length, and then noting the appropriate response for the most recent stimulus, based on the previous history. There are two special cases to deal with: sequences for which no external event is observed and sequences which, given the operational definitions of the system and stimuli, are not possible.

There are often sequences for which the appropriate response is to do nothing. For example, monitoring software may receive temperature reports but do nothing unless the temperature reports are outside a control range. For such sequences the “response” is said to be the *null* response.
Consider the following sequence for a simple calculator: “1 2 + 2 = ON.” This sequence cannot occur; it is impossible since the first event in any sequence for the calculator must be ON. Further, no sequence which starts with this sequence can occur. All such sequences are said to be illegal, and this is noted in place of the response.

As each sequence is written down, or “enumerated,” the appropriate next response (possibly null) is written. If the history is illegal, then “illegal” is written in place of the response. Then traces to the initial requirements are defined which justify the choice of response; every line of the sequence enumeration must have such justification.

For each sequence there are three possibilities: (1) The requirements uniquely specify a response. The specified response is written down, with a trace to the requirement(s) which justify it. (2) The requirements do not specify a response; they are incomplete. A response must be chosen, and the reason for the decision written down as a derived requirement. (3) The requirements seemingly specify two distinct responses for the sequence. A single response must be chosen for each sequence, and the reason for the decision written down as a derived requirement.

Consider again a simple calculator and the sequences “ON 1 2 Clear 1 + 1 =” and “ON 1 + 1 =.” Both sequences have the same appropriate next response: display 2. Further, if these sequences are extended identically they will always have the same response. When two sequences are intended to agree for all extensions, it makes no sense to enumerate both. In sequence enumeration the second sequence is noted as equivalent or “reduced” to the first, and need not be extended.

The sequence enumeration process works by extending each legal, unreduced sequence with every stimulus. These new sequences are then considered and a response, trace, and possibly a prior equivalent sequence noted. When all sequences of a particular length are either illegal or reduced to prior sequences, the sequence enumeration is complete and denotes a complete, consistent, and traceably-correct external specification.

The sequence enumeration for the TMC executive includes 204 sequences with a maximum length of eight stimuli. Table 4 is an excerpt from the full enumeration, showing sequences of length two and three. In this enumeration lambda (λ) is used to denote the empty sequence, which represents the initial conditions of the system.

<table>
<thead>
<tr>
<th>Prior Sequence</th>
<th>Stimulus</th>
<th>Response</th>
<th>Equivalent Sequence</th>
<th>Trace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pi</td>
<td></td>
<td>Illegal</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Pe</td>
<td>Ts</td>
<td>λ</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>T1</td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Pr</td>
<td>Ud</td>
<td>Pi</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Pc</td>
<td>Ud</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Pn</td>
<td>Null</td>
<td>Pi</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Cs</td>
<td></td>
<td>Illegal</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Ci</td>
<td></td>
<td>Illegal</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
## During the enumeration process for the TMC, an omission in the original requirements was discovered. D1 denotes the derived requirement, “If a second 100 ms clock interrupt is received before task completion, declare a channel failure and initiate a reboot.” The discovery and documentation of such derived requirements is a natural and desirable part of the sequence enumeration process.
2.4. Canonical Sequence Analysis

A sequence in a complete enumeration which is legal and not reduced to a prior sequence is a \textit{canonical sequence}. The empty sequence $\lambda$ is a canonical sequence which stands for the initial conditions, and every other sequence in the enumeration is a combination of a canonical prefix and a single stimulus. Further, every sequence (even those not explicitly enumerated) is either illegal, or equivalent to exactly one canonical sequence. The set of canonical sequences thus represents the set of system states, and the canonical sequence analysis process encodes these sequences using state variables and values. The canonical sequences (except $\lambda$) of the TMC enumeration are encoded using four Boolean variables in Table 5.

**Table 5**

<table>
<thead>
<tr>
<th>Table 5</th>
<th>TMC Canonical Sequence Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence</td>
<td>TaskPending</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>False</td>
</tr>
<tr>
<td>$\Pi C1$</td>
<td>True</td>
</tr>
<tr>
<td>$\Pi Pc$</td>
<td>False</td>
</tr>
<tr>
<td>$\Pi R$</td>
<td>False</td>
</tr>
<tr>
<td>$\Pi C1$</td>
<td>True</td>
</tr>
<tr>
<td>$\Pi Pc$</td>
<td>False</td>
</tr>
</tbody>
</table>

The enumeration of Table 4 can easily be converted into a state machine specification using Table 5 as follows. Every prefix is a canonical sequence which can be replaced with its state conditions. This gives previous state and current stimulus mapped to response. Each legal sequence is either equivalent to a prior canonical sequence or is itself canonical. Thus the next state is given by the state conditions for the equivalent canonical sequence. Requirements traces can be copied forward. An excerpt of the state machine specification for the stimulus $Tc$ is given in Table 6.

**Table 6**

<table>
<thead>
<tr>
<th>Table 6</th>
<th>State Machine Specification for Stimulus $Tc$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial State</td>
<td>Next State</td>
</tr>
<tr>
<td>TaskPending = False</td>
<td>TaskPending = False, Fail = False, Trip = False</td>
</tr>
<tr>
<td>TaskPending = True</td>
<td>TaskPending = False, Fail = False, Trip = False</td>
</tr>
</tbody>
</table>

It is a simple matter to encode the state machine description in a variety of formal notations. It is also possible to transform the sequence enumeration directly into other notations without performing a sequence analysis. For example one can simply assign numbers to the canonical sequences. One could then construct two functions: one which maps a sequence prefix to the appropriate number (call this $f$), and one which maps a
number and stimulus to the appropriate response (call this \texttt{resp}). ACL2 (Kaufmann 2000) implementations of these functions can be automatically generated from the enumeration for simulation of the system, and automated reasoning about the specification. The following is a fragment of such an implementation of \texttt{resp} in ACL2.

\begin{verbatim}
; Definition of resp.
(defun resp (s)
  (if (endp s) 'null
      (let ((C (f (cdr s))))
        (case (car s)
          ; Other stimuli omitted here...
          (Tc (case C
            (1 'illegal)
            (2 '(Wr Fd Td))
            (3 'illegal)
            (4 'illegal)
            (5 '(Wr Fd Td))
            (6 'illegal)
            (7 '(Wr Fd Td))
            (8 'illegal)
            (9 'illegal)
            (10 '(Wr Fd Td))
            (11 '(Wr Fd Td))
            (12 '(Wr Fd Td))
            (13 'illegal)
            (14 'illegal)
            (15 '(Wr Fd Td))
            (16 '(Wr Fd Td)))))
      )))
\end{verbatim}

This ACL2 implementation can be automatically generated from the enumeration. It provides an executable version of the specification for simulation and for use as an oracle during testing. Further, one can immediately use the mechanical theorem proving capabilities of ACL2 to prove important system properties.

5. CONCLUSIONS

Sequence-based specification provides a systematic way to generate a specification from initial requirements. The resulting specification can be written as a state machine, or encoded in a variety of formal notations for use with automated theorem provers, model checkers, and such. Thus one can use sequence-based specification to move from the realm of the application domain experts who understand the external control requirements for the system, to the realm of the trained software engineer, who understands how best to use notations and tools to create reliable, efficient software.

NOMENCLATURE

\begin{itemize}
\item $p$ pressure
\item $F$ mass flow rate
\item $T_c$ cold leg temperature
\item $T_h$ hot leg temperature
\item $\phi_i$ neutron flux
\item $D_d$ computed DNBR
\end{itemize}
\( D_t \)  DNBR setpoint

REFERENCES


