Development tools for bus controller software

by M. I. THOMAS
TECSI-SOFTWARE
Paris, France

ABSTRACT

This paper addresses the problem of software specification and generation for bus controller software. This software is representative of a class of software modules for which the tools may be used. Two tools are described, a simple language in which to specify the module and a program generator that produces code directly from the specifications. The language uses finite state diagram ideas as do many other specification languages, but is constrained so that generation of high-quality code is feasible. A brief outline of the structure of the code generated is given, followed by some indications of the performance of the tools and the experience gained from their use.
INTRODUCTION

This paper describes two program-development tools that have been constructed for use in an avionics software project. The tools are a language for specifications of a data-bus system and a program generator used to convert the specifications into code. They were developed for their utility in the specific problem domain rather than with more general objectives. Nevertheless they illustrate, in practical running software, the advantages to be gained by formal system specification and its automatic conversion into code.

The data-bus standard of MIL-STD-1553B defines a configuration consisting of a bus controller that supervises data traffic on a bus to which a number of peripherals may be attached. The controller uses internal data structures to decide which source and destination devices are appropriate for a data transfer request. In effect, these data structures model the current state of the peripherals attached to the bus. Certain signals, for example a notification of device failure, change the values in the data structures and will consequently affect the treatment of subsequent transfer requests.

The classical paradigm for problems of this nature is the state diagram. It is difficult to apply in this particular case because of the large number (up to 30) of peripherals, each of which may assume many different states. An average of \( n \) states for each peripheral gives a state space of \( n^3 \) configurations. Because the states of the peripherals are largely independent, the size of the state space is not greatly reduced by constraints on interrelations). Any reasonably complete state diagram or tabular representation is clearly impractical in these circumstances.

The consequent difficulties facing the project were those of

1. Specifying the controller software responsible for maintaining a correct model of the current bus state.
2. Specifying how bus signals were to be processed taking account of the information in the system model.
3. Verifying the coherence of these specifications.
4. Producing a program embodying these specifications in such a way that small changes in the specification (which may be frequent in a development environment) could be reliably and quickly incorporated into the program.

These difficulties were resolved to a greater or lesser extent by two complementary tools that have been developed. The first tool is a simple language that can be used to specify the bus controller software. It has facilities to describe the data structures used by the bus controller to model the system state. The language may also be used to describe how the controller must react to bus signals in the light of the current state of its model. The second tool is a program that accepts a specification written in the language and generates variable declarations and code to meet the specification.

The next section of this paper describes the language that has been developed and its relation to other specification languages and models. The third section outlines some features of the program-generator tool. This is followed by a brief description of the experience gained by the use of these tools.

THE SPECIFICATION LANGUAGE

Overview

The language was designed to allow the specification of software belonging to a relatively small subset of systems. It does not attempt to describe hardware components or the interactions and synchronizations between multiple software components, nor does it contain facilities to reflect timing constraints. It is clearly not intended to serve as a general system-specification language.

It does permit the description of an individual software module and the behavior required from it in response to the various inputs that it may receive. These inputs may come from one or several software modules. It is assumed that the processing of each input to the specified module is completed before processing of the next input starts. This means that the problems of deducing the access protection necessary for the module’s common data structures are avoided, since concurrent accesses are precluded.

These restrictions of the language were adopted for the pragmatic reason that the resultant language would adequately specify the bus controller software under development, and to facilitate the generation of code directly from specifications written in the language. There are some more general specification languages that permit the generation of code skeletons for the specified system either automatically or by a straight-forward manual process, but a considerable amount of information needs to be added to these code skeletons.

A Simple Example of Part of a System Specification

It is important to note that any example, such as the one given here, that is small enough to be understood easily can also be represented efficiently in a tabular structure. As the size of the example grows, the tabular structure becomes more unmanageable.
The example incompletely specifies the behavior required from a software module controlling part of the electrical system of a car. When the car’s ignition switch is on, the direction indicator switch activates the flashing direction indicators. When the ignition switch is turned off and the direction indicator switch is in the left or right position, the car’s electrical system illuminates the left or right parking light (unless the parking light switch is on, in which case both lights are illuminated).

**COMPONENTS**
- IGNITION = (ON, OFF);
- INDICATOR-SWITCH = (LEFT, NEUTRAL, RIGHT);
- PARKING-SWITCH = (ON, OFF)

**SIGNALS**
- ACTIVATE-INDICATOR-LEFT,
- ACTIVATE-INDICATOR-RIGHT,
- DEACTIVATE-INDICATOR,
- IGNITION-SWITCHED-ON,
- IGNITION-SWITCHED-OFF,
- PARKING-SWITCHED-ON,
- PARKING-SWITCHED-OFF

**RULES**

- (IGNITION = ON) & ACTIVATE-INDICATOR-LEFT
  - [START LEFT INDICATOR FLASHING]
  - (INDICATOR-SWITCH = LEFT);

- (IGNITION = OFF,
  PARKING-SWITCH = OFF,
  INDICATOR-SWITCH = LEFT) &
  DEACTIVATE-INDICATOR
  - [TURN OFF LEFT PARKING LIGHT]
  - (INDICATOR-SWITCH = NEUTRAL);

- (PARKING-SWITCH = OFF,
  INDICATOR-SWITCH = LEFT) &
  IGNITION-SWITCHED-ON
  - [TURN ON ALL IGNITION SYSTEMS]
  - [TURN OFF LEFT PARKING LIGHT]
  - [START LEFT INDICATOR FLASHING]
  - (IGNITION = ON);

- etc.

A specification contains three major subdivisions, namely the COMPONENTS, SIGNALS, and RULES sections.

The COMPONENTS section serves to define the data structures manipulated by the module being specified and used by the module to determine its responses to the input signal it receives. In the case of the bus controller these data structures may represent the state of peripherals attached to the data bus. It is also possible to define data structures that do not reflect the state of physical devices; for example a data structure that indicates the current flight phase could take the values TAKE-OFF, LANDING, CRUISE, and COMBAT.

**FLIGHT-PHASE =**

(TAKE-OFF, LANDING, CRUISE, COMBAT)

Two types of data structure, simple and complex, may be defined in the COMPONENTS section. Simple data structures are analogous to PASCAL’s enumeration types. The data structures IGNITION, INDICATOR-SWITCH, and PARKING-SWITCH in the example are all simple. Complex data structures are a way of grouping a set of related simple and complex data structures in order to refer to them by name.

Where a module’s data structures represent the states of entities outside the module they need only reflect those classes of states that can influence the actions of the module. That is, the internal states of the entity can be partitioned and each class of the partition be considered as one state from the point of view of the module. This approach is common in specification languages.

The SIGNALS section contains a list of input signals that may be received by the module. The input signals are not parameterized in any way, so an input signal whose parameters cause radically different actions to be performed should be characterized by different input-signal names in this section. In fact, the list may also contain signals which are generated internally within the module. These signals can be used to direct the sequencing of the processing of external inputs to the module.

The RULES section associates states of the module’s data structures and an input signal with the processing that needs to be executed and the new state of the module’s data structures. The left-hand side of each rule (preceding $\Rightarrow$) defines a subset of the state space of data structures, called the source set, and also contains the name of an input signal. The right-hand side of each rule contains an optional list of the names of actions to be taken and the new values to be assigned to the data structures, called the destination state. Only those data structures that actually change value need to be mentioned.

The semantics of a rule are clearly related to the state diagram paradigm:

- Whenever the signal is to be processed by the module and the data structure values belong to the source set than the rule is applicable. The named actions should be carried out and then the data structures should assume the new values given.

Each rule therefore defines a class of transitions on the state diagram whose states reflect all the possible configurations of the data structures.

As indicated in the example, the source set need not indicate values for all of the module’s data structures. It is also possible to specify that a data-structure value should be one of a subset of its possible values or that it should not take a certain value or a subset of values. For example,

- (FLIGHT-PHASE $\neq$ COMBAT)
- (FLIGHT-PHASE = (TAKE-OFF OR LANDING)).

Neither of these forms is permitted in the description of the
destination state since it makes no sense to assign any one of a set of values to a data structure.

There are situations in which the actions in a rule may change the new value to be assumed by the module’s data structures. For example, in a printing module there may be a data structure

\[ \text{END-OF-PAGE} = (\text{TRUE, FALSE}). \]

The rule describing the printing of a line may be

\[(\text{END-OF-PAGE} = \text{FALSE}) \& \text{LINE-TO-PRINT} \Rightarrow \]
\[\begin{align*}
&\text{[PRINT LINE]} \\
&\text{[DECREMENT NUMBER OF LINES REMAINING]} \\
&\text{(END-OF-PAGE} = \text{FALSE});
\end{align*}\]

The action of decrementing the number of lines remaining may yield the result zero, in which case END-OF-PAGE should become TRUE. It is possible to indicate that the value to be assigned to a data structure may be changed by an action so that the code-generation algorithm may take appropriate action.

The language also provides facilities for the definition of data structure types in a manner analogous to type definition in PASCAL or ADA. These definitions appear in an optional TYPES section.

Comparisons with Other Specification Languages

The term specification language as it is currently used covers two broad classes of language. The first is requirement-specification languages. The system to be specified is described from an external point of view, though reference may be made to subsystems in order to clarify the description. The second class is system-specification languages, where the system whose design is to be specified is decomposed into subsystems. Each of these subsystems is defined together with its relationship to other subsystems. This paper describes a specification language for use at a later stage in the development process, where the behavior of individual software components that are regarded as non-composable must be specified. Formalization of the interaction between such components, though of primary importance in system-specification languages, is not addressed here. The description of system components using the concept of internal state is common in specification languages.

However, the decomposition of these states into independent data structures whose combinations define the state space of the component is not present in these systems. The notion of a system stimulus is also used in specification languages. The interpretation in the language specified here is identical to its usage in these other languages.

An older technique for specification of software modules is the decision table. Although decision-table conditions can be regarded as specifying the subset of states in which some actions are to be performed, the idea of a new state to be entered after the actions is not present.

Verification of the Specification

One of the advantages of a rigorous formal specification is that verification techniques may be applicable, though the current system does not contain a verification tool. The only check that is currently carried out ensures that at most one transition rule is applicable from any system state in response to any input signal. In manually written programs this situation is often disguised by dependencies on the order in which the various tests of the data structures are made. This section will indicate some verification techniques that are readily applicable to specifications formulated in the language.

It must be assumed that the data structures of a module are independent, that is, that manipulation of one does not necessarily change the value of others. A bus-controller module where the data structures model independent peripherals attached to the bus is an example of such a case. Such systems can be concisely modeled using Petri Nets. Several existing languages for system description use Petri Nets for validation purposes.

Each state of each data structure may be represented by a place in a Place/Transition Net. The transitions of the net represent the transition rules of the specification. The marking graph of these places defines the state space of possible configurations of the data structures. Clearly the Place/Transition net so far described is an alternative representation of the state diagram of the system and needs to be completed with a description of the signals and the order in which they may arrive for processing by the module. The new representation does however permit the application of analyses of structure that would be more difficult with state diagram representations, while the fact that the net represents a state diagram reduces the complexity of these analyses.

The use of such verification techniques implies an extension of the language to cover descriptions of possible input-signal orderings. Riddle has developed a formalism to describe such orderings. More advanced verification may include the division of the state space of data-structure configurations into classes of a spectrum ranging from Impossible through Acceptable Malfunction to Complete Malfunction using the concepts of deontic logic.

THE PROGRAM GENERATOR

This software tool was developed to convert specifications into high-quality code. It also generates declarations for the data structures defined in the specifications. PASCAL or CORAL66 code may be generated. Since the generator uses an intermediate representation of the program, other target languages can be easily added.

The generated program has a structure analogous to PASCAL's case statement, where the case branch selection is made on the input signal that is to be processed. This does not represent a loss of generality, since the input signals can be represented as the values of a data structure in the COMPONENTS section and the SIGNALS section can be redefined to contain only a single input signal that is the notification that an input event has occurred.
When the transition rules of a specification have been partitioned into classes according to the input signals under which they may apply, code is generated for each class. The order in which the tests of data structure values are made greatly affects the efficiency of the generated code. An exhaustive search of possible orderings even within a subset of rules is not possible for combinatorial reasons, so heuristic methods are used.

A discussion of the problems of code generation and the algorithms used can be found in another of the author's articles.13

EXPERIENCE WITH THE TOOLS

The avionics software project for which the tools were developed is divided into phases. At the end of the first phase a hand-coded version of a simple bus controller had been prepared. This hand-coded version used a tabular, packed data representation of the states and transitions of the bus controller. The tools were also completed towards the end of the first phase. In order to gain experience, the simple controller was specified in the defined specification language. This took approximately two weeks. However, no comparison with the time taken to specify the hand-coded version is possible since no formal specification of it existed.

The program and data of the tool-generated version occupied approximately the same amount of memory as the original, but the execution time for a sequence of test inputs was reduced by approximately 30%. It is difficult to evaluate this result, since the automatically generated version does not use packed data.

The time taken for the tool to process a set of rules obviously depends on their characteristics, but sets of one hundred rules are typically processed in approximately three minutes of CPU time on a DEC Vax computer.

REFERENCES