International Journal of Scientific Research in Computer Science, Engineering and Information Technology

© 2017 IJSRCSEIT | Volume 2 | Issue 6 | ISSN : 2456-3307

An Empirical Principle for the Data Flow Analysis in State Transition Systems

A. Guerrouat

University of Bahrain, Zallaq, Bahrain

ABSTRACT

This paper focuses on the analysis of the data flow in formal specifications of embedded systems founded on state transition models. There are several properties of formal specifications that result from considering the data flow, however, these are irrelevant if only the control flow is assumed. The typical examples of formal models supporting data flow are extended finite state machines (EFSMs) whereas finite state machines (FSMs) specify only the control flow. It is crucial to predict issues that could be caused by the data flow because their occurrence in later stages of the system development or during operation would be very costly and critical. As it is practically hard to state the presence/absence of such issues analytically, this will be done empirically.

Keywords : Formal Analysis And Testing, Data Flow Analysis, Extended Finite State Machines, Formal Methods

I. INTRODUCTION

Because of the growing complexity of embedded systems the software development process becomes a costly and error-prone activity. The cost factor plays a central role in today's industrial competition, for instance between car manufacturers. The development of competitive and efficient products is imposing more and more constraints to the design of embedded systems. One of the means to reach this goal are formal methods to support the different phases of system development, i.e. specification, analysis, synthesis and testing.

Formal methods have proven their effectiveness in the analysis of complex requirements like those for communicating systems [1] [2]. Furthermore, they provide a solid mean for unambiguous specification and rigorous analysis. They are based on formal methods such (E)FSMs ('extended' finite state machines) and differ from conventional programming languages by providing not only a formal syntax but also a formal semantic [4]. Moreover, the application of formal specification increases the confidence in the software and the system. Especially in the area of safety-critical systems, the use of formal techniques is highly recommended [5].

Statecharts as a semi-formal model is actually the mostly used formalism to specify requirements for embedded systems [4]. Although Statecharts provide graphical facilities, they might lack formal and unambiguous semantics. Therefore, detecting bugs, incompleteness and inconsistencies becomes a difficult task. Furthermore, they are only used to describe behavioral requirements. To alleviate these lacks many authors try to combine formal notations like Z with state-transition models [5]. Z is based on set theory and first order predicate logic and used for data structuring and abstracting. However, approaches developed around this model do not clearly address test data generation methods for e.g. analysis and testing purposes.

The finite state machine model is very popular in the control flow specification of state/transition-based systems and many related analysis methods have been developed [6] [7]. These support a formal test derivation which can be used for validation and testing purposes. However, finite state machines lack to deal with the data flow. This shortcoming can be alleviated by using the extended finite state machine model (EFSM). However, the test generation and specification analysis task cannot be easily applied in this case due to the data flow.

In this paper, we discuss the data flow related problems for analysis and testing of embedded system specifications and implementations. Assuming an extended state machine model, these problems will be first identified, and then an empirical solution based on analysis and testing expert knowledge is proposed. Because an analytical solution would be very hard to achieve due to the huge amount of data to be considered.

The paper is organized as follows. Section 2 reviews the conventional finite state machine and the extended finite state machine models regarding data flow specification. In Section 3, we define the issues related to the data flow and propose the basic idea for localization. Finally, Section 4 concludes the paper.

II. Basic Concepts

In this section, we first review the formal definition of the used data flow model (EFSM) and compare it to the simple finite state machine model (FSM). These definitions are needed for subsequent section. Further, we functionally explain the meaning of such model for concrete systems such as embedded systems.

2.1 Definitions

2.1.1 Finite State Machines

A *finite state machine* (FSM) is a 5-tuple $<S, I, O, T, s_0>$, where S is a non-empty finite set of states, I a non-empty set of inputs, O a non-empty finite set of outputs, $T \subseteq S \times I \times O \times S$ the set of transition relations, and $s_0 \in S$ the initial state of the FSM.

A *transition* $t \in T$ of an FSM is a 4-tuple $<s, i, o, s'>$, where $s \in S$ is a current state (the edge), $i \in I$ an input, $o \in O$ an output related to s and I, and $s' \in S$ the next state (a tail state) related to s and i.

The FSM model are well appropriate for specifying only the control flow, but not for the data flow that could be associated with the control flow. Thus, simple FSMs have been extended to support the data flow through additional state variables, interaction parameters and guard conditions associated with transitions that could be also temporal conditions.

2.1.2 Extended Finite State Machines

An *extended finite state machine* (EFSM) is a 7-tuple $<S, C, I, O, T, s_0, c_0>$ where S is a non-empty set of main states, $C = dom(v_1) \times \ldots \times dom(v_n)$ a non empty countable set of contexts with $v_i \in V$, V the non-empty finite set of variables and $dom(v_i)$ a non-empty countable set referred to as the domain of v_i, I a non-empty finite set of inputs, O a non-empty set of outputs, $T \subseteq S \times C \times I \times O \times S \times C$ the set of transition relations, $s_0 \in S$ the initial main state, and $c_0 \in C$ the initial context of the EFSM.

A main state may consist of sub-states. A context is a specific assignment of values to the variables. A transition $t \in T$ of an EFSM is a 6-tuple $<s, c, i, o, s', c'>$ where $s \in S$ is a current main state, $c \in C$ a current context, $i \in I$ an input, $o \in O$ an output, $s' \in S$ a next main state, and $c' \in C$ a next context.

A transition may be characterized, in addition to its current and next state and input and output interactions and context, by a so-called *guard condition* or *enabling predicate*. This represents a condition on a state transition and the related output to be carried out, once the predicate 'fires'. All usual logical and comparative operators *and, or,* =, > etc. are allowed in a predicate. Thus, a transition takes place only if its *enabling predicate* fires. It depends on the current FSM state with additional variables and the concrete variables values (context) of the input. Therefore, we introduce a so-called p-EFSM in which the enabling predicates on transitions are explicitly specified.

A guarded or predicated EFSM is resulted from the above defined EFSM and presented as follows:

A guarded or predicated extended finite state machine (p-EFSM) is an 8-tuple $<S, C, I, P, O, T, s_0, c_0>$ where S is a non-empty set of main states, $C = dom(v_1) \times \ldots \times dom(v_n)$ a non-empty countable set of contexts with $v_i \in V$, V a non-empty finite set of variables, and $dom(v_i)$ a non-empty countable set referred to as the domain of v_i, P a countable set of predicates (possibly empty), I a non-empty finite set of inputs, O a non-empty finite set of outputs, $T \subseteq S \times C \times I \times P \times O \times S \times C$ a set of transition relations, $s_0 \in S$ the initial main state, and $c_0 \in C$ the initial context of the p-EFSM.

A transition $t \in T$ of a p-EFSM is a 7-tuple $<s, c, I, p, o, s', c'>$ where $s \in S$ is a current main state, $c \in C$ a current context, $i \in I$ an input, $p \in P$ a enabling predicate which depends on the context c, $o \in O$ an output, $s' \in S$ a next main state, and $c' \in C$ a next context.

Note that EFSMs can functionally describe system components that may be blocks or modules depending on the used formal description technique such as SDL [4].

2.2 Specification of Embedded Systems

The basic structure of an embedded system environment comprises an *external process*, *sensors*, *actuators*, and a *controller*:

- *External process* is a process that can be of physical, mechanical, or electrical nature.
- *Sensors* provide information about the current state of the *external process* by means of so-called *monitoring events*. They are communicated to the *controller*. For the *controller*, they represent input events. They are considered as stimuli for the *controller*.
- *Controller* must react to each received event, i.e. input event. Events originate usually from *sensors*. Depending on the received events from sensors, corresponding states of the *external process* will be determined.
- *Actuators* receive the results determined by the *controller* which are communicated to the *external process* by means of so-called *controlling events*.

The embedded system specification consists of the specification of its environment and its controller. We assume that the embedded system is state-transition based. Thus, the behavior of the above components will be considered as a collection of p-EFSM models interacting with each other via broadcasting events according to a given interaction relationship.

In the simplest case, we can model the behavior of each component as a single p-EFSM as indicated below. The composition of EFSMs is out of the scope of this paper.

- Sensors: $p\text{-}EFSM_s = <S_s, C_s, I_s, P_s, O_s, T_s, s_{0s}, c_{0s}>$
- Controller: $p\text{-}EFSM_c = <S_c, C_c, I_c, P_c, O_c, T_c, s_{0c}, c_{0c}>$
- Actuators: $p\text{-}EFSM_a = <S_a, C_a, I_a, P_a, O_a, T_a, s_{0a}, c_{0a}>$

III. Fixing Analysis and Testing Issues

3.1 Defining Analysis Issues

Properties that usually aimed by analysis and testing when the data flow is considered are summarized as follows:

- The non-existence of non-executable actions: The system comprises no actions that cannot be executable under normal conditions.
- Liveliness: Each state of the system is reachable from the initial state.
- Deadlock-freeness: The system reaches no state that does not allow to interact with the environment and never leaves it.
- Livelock-freeness: The system comprises no non-productive cycles.
- Error tolerance and resynchronization: The system reaches a normal state within a limited time period after an error leading to an abnormal state within a limited time period after an error leading to an abnormal state has been occurred.
- Safety: The system comprises no unspecified inputs or outputs.
- Partial correctness: The system provides a special service when it terminates.
- Termination: The system reaches each time the final state(s), or the initial state for cyclic systems.

To allow the analysis of embedded systems precise specifications are essential. The use of formal methods enables the automation of most aspects of these activities [6] [7] [8]. We are particularly interested in the following analysis and testing issues:

- The non-executability of parts of the system
- Deadlock situations
- Inconsistencies of the system, i.e. whether the system contains non-deterministic behaviors
- Prohibited types of communication
- Incompleteness
- Checking of erroneous behaviors

3.2 Basic Idea of the Empirical Analysis

Analytically, the statement of the presence or absence of the above problems in specification or implementations is generally very hard or impossible due to the specified data flow in term of state variables (related to state explosion problem), interaction parameters, predicates or guard conditions specified for transitions, data types and domain variables and parameters etc. Furthermore, the derivation of appropriate test cases is also an important issue for embedded systems testing and for software testing in general. In the context of embedded system testing, the question of deciding whether a given part of the specification is executable is a difficult issue. For this reason, the most work on verification and test development for embedded systems assumes that the system is specified in a simple state transition model without considering the data flow.

Taking into consideration the difficulties explained above, we suggest to classify the variables and parameters specified in the data flow according to the extent of their definition domains:

- Behavior parts with only control flow, and without any data flow can be theoretically analytically analyzed.
- Behavior parts including exclusively variables and parameters with a finite number of values can be theoretically analytically analyzed:
  - if they are in a reasonable amount
  - otherwise empirically
- Behavior parts including at least one variable or parameter with an infinite number of values:
  - empirically

Note that the empirical approach is based on the knowledge and experiences of the analysis and test expert. Based on his expert knowledge, a tester is able to focus analysis by selecting successive sub-domains of a variable with a reasonable amount that could be very critical for the well-functioning of the system. After performing the analysis for all successive sub-domains, a heuristic value about the absence of the given analysis issue among those defined above can be stated.

3.3 Formulation of the Analysis Issues

Below is an example about the formulation of the problem of detecting one of the analysis issues mentioned above based on the modified EFSM model. Similarly, we formally define the other issues to which, then, the above empirical analysis principle can be applied.

Detecting of non-executable parts

The problem of detection of non-executable parts is the problem for deciding whether a given p-EFSM modeling a function for a given component of the embedded system contains non-executable transitions and detecting them if they exist.

The detection of non-executable branches allows to deduce a specification whose all transitions are executable. The detected specification should be more simplified and obtained by eliminating all non-executable transitions and their descendants.

We can find all non-executable branches in a p-EFSM as follows. A branch $s \to s'$ in the p-EFSM is a non-executable if the two following conditions are fulfilled:

- $\exists\, x_1, \ldots, x_k\, [\delta_s(x_1,\ldots,x_k)]$
- $\neg(\exists\, x_1, \ldots, x_k\, [\delta_{s'}(x_1,\ldots,x_k)])$

where

- s and s' are two states of the p-EFSM.
- $\delta_s(x_1,\ldots,x_k)$ represents the conjunction of all predicates for the context $x_1, \ldots, x_k$ from the initial state s_0 until the state s.

Statement

The problem for deciding, whether a given branch of the p-EFSM is non-executable, is resolvable under certain limiting assumptions.

Statement

The problem for deciding, whether a given branch of the p-EFSM is non-executable, is resolvable under certain limiting assumptions.

IV. CONCLUSION

In this paper, we have proposed an analysis principle of data flow for embedded systems. This assumes that the specification or implementation is modeled as EFSMs that allows to specify the data flow in addition to control flow. We have identified the issues related to data flow specification and explained the difficulties of their statement analytically. As alternative, we have proposed an empirical analysis principle that repetitively performs analysis on sub-domains of the variables specified in the data flow and the end provides a heuristic statement.

In a future work we plan to refine the empirical analysis approach proposed here and to apply it on a real-life embedded system.

V. REFERENCES

[1]. A. Sinha et al. On generating EFSM models from use cases. In Proc. of sixth international workshop on scenarios and state machines (SCESM'07), pp 1–8, 2007.

[2]. H. Katagiriy et al. Hardware, implementation of communication protocols modeled by concurrent EFSMs with multiway synchronization. In Proc. of the 37th conference on design automation (DAC'00), Los Angeles, 2000.

[3]. V. S. Alagar. Specification of Software Systems. In Springer, 2014. ISBN: 1475729219.

[4]. Specification and Description Language SDL '92. ITU-T Recommendation Z.100, 1992.

[5]. R. Buessow, R. Geisler, and M. Klar. Specifying safety-critical embedded systems with statecharts and Z: A case study. In Proceedings of Fundamental Approaches to Software Engineering (FASE'98), Lisbon, 1998.

[6]. M. Mendler, G. Luettgen. Statecharts: From Visual Syntax to Model-Theoretic Semantics. In K. Bauknecht, W. Brauer, and Th. Mück (editors), Workshop on Integrating Diagrammatic and Formal Specification Techniques (IDFST 2001), pages 615-621, Vienna, 2001.

[7]. B. Potter, J. Sinclair, and D. Till. Introduction to Formal Specification and Z (2nd Edition). Prentice Hall PTR; 1996.

[8]. A. V. Aho, A. T. Dahbura, D. Lee, and M.U. Uyar. An optimisation technique for protocol conformance test generation based on UIO sequences and Rural Chinese Postman Tours. In S. Aggarwal and K. Sabnani, editors, Protocol Specification, Testing, and Verification, New Jersey, 1988.

[9]. S. Fujiwara, G.v. Bochmann, F. Khndek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. IEEE transaction on Software Engineering 17(6): 591-603, 1991.