EDT: A Specification Notation for Reactive Systems

R Venkatesh, Ulka Shrotri, G Murali Krishna, Supriya Agrawal
Tata Consultancy Services Ltd.
Email: {r.venky, ulka.s, g.muralikrishna, supriya.agrawal1}@tcs.com

Abstract—**Requirements of reactive systems express the relationship between sensors and actuators and are usually described in a natural language and a mix of state-based and stream-based paradigms. Translating these into a formal language is an important pre-requisite to automate the verification of requirements. The analysis effort required for the translation is a prime hurdle to formalization gaining acceptance among software engineers and testers. We present Expressive Decision Tables (EDT), a novel formal notation designed to reduce the translation efforts from both state-based and stream-based informal requirements. We have also built a tool, EDTTool, to generate test data and expected output from EDT specifications.**

In a case study consisting of more than 200 informal requirements of a real-life automotive application, translation of the informal requirements into EDT needed 43% lesser time than their translation into Statecharts. Further, we tested the Statecharts using test data generated by EDTTool from the corresponding EDT specifications. This testing detected one bug in a mature feature and exposed several missing requirements in another. The paper presents the EDT notation, comparison to other similar notations and the details of the case study.

I. Introduction

Reactive systems interact with their environment via sensors (inputs) and actuators (outputs). These interactions are usually described informally using a combination of natural language, state-based paradigms such as state transition tables and stream-based paradigms such as timing diagrams. Translating these informal requirements to a formal notation is necessary for automated analysis, simulation and test case generation. Current popular formal notations are primarily state-based and hence the transition from non state-based informal requirements to these need a lot of analysis and is effort intensive. This often deters software engineers and testers from readily adopting formal notations. To address this issue, we propose an easy to use executable notation, Expressive Decision Tables (EDT). The notation is regular expression based and combines both the paradigms in a novel manner leading to compact specifications that are easy to manage. We adopted a tabular format for EDT, because tables are more readable, easier to edit and less error prone when compared to textual, graphical or logical notations [1]. We have also built a tool, EDTTool, to generate test data and expected output from EDT specifications.

The compactness of specifications in EDT compared to other notations is illustrated through three requirements of the wiper of a vehicle shown in Figure 1. The requirements are taken verbatim from a real world automotive application.

1) *If the ignition and wiper switch are* `on` *and there is no fault in the wiper, then send a* `wipe` *message to the wiper.*
2) *If wiper is vibrating between* `park` *and* `notpark` *position, that is, if the wiper is stuck, send a* `dontwipe` *message to the wiper. Note: Wiper is considered to be stuck if the wiper switches between* `park` *and* `notpark` *position thrice within a second.*
3) *To reset the wiper error, while ignition is* `on`*, the wiper switch needs to be switched* `on` *and* `off` *within half a second.*

Fig. 1. Wiper Example

The first requirement is based on the state of ignition and wiper switch (state-based), the second is based on the pattern of input values read by the wiper position sensor (stream-based) and the third combines the state of ignition and the input pattern of the wiper switch (state-based and stream-based).

Such requirements are typically formalized using a state-based notation like Statecharts [2] or Software Cost Reduction (SCR) [3]. Users of these notations need to distinguish between events and states. They also have to introduce intermediate states and auxiliary variables to

TABLE I. EDT FOR WIPER EXAMPLE

sno	in ignition	in wiperswitch	in parksensor	in error	out wipercmd	out error
1.	on	on		false	wipe	
2.			(park;notpark){= 3}{≤1 s}		dontwipe	true
3.	on			true		
		on{≤0.5 s};off				false

capture event patterns of stream-based requirements such as the second requirement above. This leads to lengthy specifications that are difficult to create and manage. These limitations are partially addressed by Stream-based I/O tables [4] that supports both stream-based and state-based requirements. However, the notation is not powerful enough to specify the second and third requirement of the Wiper Example.

We have shown the EDT specification of the Wiper Example, in Table I. It illustrates the power of EDT as it requires fewer rows than the corresponding specification in comparable notations like SCR (Table II) or Stream-based I/O tables (Table III). Details of this comparison can be found in Section IV.

To evaluate the power of EDT and its ease of use, we conducted a case study where novice testers successfully specified seven features of a real life automotive application. It took lesser time for them to create EDT than the time required to create Statemate [5] Statecharts for the same features. When testers tested the Statecharts specifications using test cases generated by EDTTool they detected a bug in one of the already tested features.

EDT is introduced informally in Section II and a formal semantics of the notation is given in Section III. Comparison of EDT with other notations is shown in Section IV. Case study details are presented in Section V. Section VI provides conclusion and future work.

II. EXPRESSIVE DECISION TABLES (EDT)

Table I shows an EDT specification of the Wiper Example described in the Figure 1. The column headers specify four input ports - `ignition`, `wiperswitch`, `parksensor` and `error` and two output ports `wipercmd` and `error`. `error` is an input and output (I/O) port. The table has three row-sequences with the first two row-sequences having only one row each and the third having two rows, depicted by the common sequence number 3, for rows three and four. Each requirement stated above is specified in the table as a separate row-sequence and should be interpreted as follows:

TABLE II. SCR TABLES FOR WIPER EXAMPLE

Mode Transition Function for Wipe		
Source Mode	**Events**	**Destination Mode**
NoWipe_S	@T(ignition = on) WHEN (wiperswitch == on) AND NOT error	Wipe_S
NoWipe_S	@T(wiperswitch = on) WHEN (ignition == on) AND NOT error	Wipe_S
NoWipe_S	@F(error) WHEN (ignition == on) AND (wiperswitch == on)	Wipe_S
Wipe_S	@T(error)	NoWipe_S

Event Function for wipercmd		
	Events	
	@T(Wipe = Wipe_S)	@T(Wipe = NoWipe_S)
wipercmd′ =	wipe	dontwipe

Event Function for cnt	
	Events
	@T(parksensor = notpark) WHEN cnt <3
cnt′ =	cnt + 1

Resetting cnt to zero is non trivial (not shown for brevity)

Event Function for starttime	
	Events
	@T(parksensor = park) WHEN (cnt = 0)
starttime′ =	time

Event Function for wiperontime	
	Events
	@T(wiperswitch = on)
wiperontime′ =	time

Event Function for error		
	Events	
	@T(cnt=3) WHEN (time - starttime < 1 sec)	@T(wiperswitch = off) WHEN (time - wiperontime < 0.5 sec)
error′ =	TRUE	FALSE

TABLE III. STREAM-BASED I/O TABLE FOR WIPER EXAMPLE

Iwiper-switch	Iignition	Ipark-sensor	Ierror	Owiper-cmd	Annota-tion
onϵ*	onϵ*	.	false	wipe	Req1
onϵ*	on	.	false ϵ*	wipe	Req1
on	onϵ*	.	false ϵ*	wipe	Req1

Requirements 2 and 3 cannot be specified using Stream-based I/O tables

1) If the values at ports `ignition` and `wiperswitch` are both `on` and the value at `error` is `false` then output `wipe` at `wipercmd` port as soon as all inputs arrive.
2) If `park` followed by `notpark` repeats thrice within one second at the `parksensor` port,

output `dontwipe` at the `wipercmd` port and `true` at the `error` port.

3) If the last value for `ignition` is `on`, and `error` is `true`, and then `wiperswitch` has value `on` followed by `off` within 0.5 seconds, then output `false` at the `error` port.

An EDT specification consists of one or more tables where the column headers specify the input and output ports and the rows specify the relationship between input and output values. Each cell in a row consists of a regular expression that is used to match input streams at that port. Input values arrive as a stream at input ports at discrete time units and output values are generated as a stream at output ports at discrete time units. The rules for regular expression pattern are explained by the following examples:

- The pattern `on` matches if the last value seen is `on`.
- `on`$\{\leq$`0.5s`$\}$ matches if the last value seen is `on` and not more than 0.5 seconds have passed since `on` was seen. (same is applicable to $<$, $=$, $\geq$ and $>$)
- `on`$\{\leq$0.5s$\}$;`off` matches if the last two values are `on` followed by `off` and `off` occurs within 0.5 seconds of `on`.
- (`park`;`notpark`)$\{$=3$\}\{\leq$1s$\}$ matches if the pattern `park` followed by `notpark` repeats thrice within 1 second.
- An empty cell matches any value if all corresponding cells before it in the row sequence are also empty, else it matches nothing. In the third row-sequence of the Wiper Example, both cells of the `parksensor` match any value whereas in the second row, the cell for `ignition` will match only if there is no value.

The first row of any row-sequence matches if each input cell of that row matches. Subsequent rows match when all rows before it match and all its input cells match. Once a row matches the system outputs values as specified by the patterns of that row's output cells. Once a row has matched, if no further inputs arrive at those ports, the row will continue to match but no new output will be generated.

The matching rules are illustrated for the wiper example below. For the EDT specification in Table I:

- Consider the set of input strings at the end of four time units as $wiperswitch$ = $[on\ \epsilon\ \epsilon\ \epsilon]$, $ignition$ = $[\epsilon\ on\ \epsilon\ \epsilon]$ and $error$ = $[false\ \epsilon\ \epsilon\ \epsilon]$, where ϵ represents the absence of any value at that time. At time 2, row-sequence one matches and the value $wipe$ is output to $wipercmd$ at time 3 and hence its output stream will be $[\epsilon\ \epsilon\ wipe\ \epsilon]$. Note that although this row-sequence continues to match at time 3, there is no output at time 4 because this is just an extension of the previous match with no further inputs.
- Consider the set of input strings at the end of three time units as $ignition$ = $[on\ \epsilon\ \epsilon]$, $error$ = $[true\ \epsilon\ \epsilon]$ and $wiperswitch$ = $[\epsilon\ on\ off]$. At time 1, the first row of row-sequence three matches and at time 3, the second row matches (assuming each time unit corresponds to 100ms). This results in $false$ being output to $error$ and its string becomes $[true\ \epsilon\ \epsilon\ false]$

It is evident from the example that both the state-based and stream-based requirements map directly to row-sequences in EDT. The third row-sequence illustrates how complex time ordering dependencies between inputs and outputs can be expressed succinctly.

The following section presents the formal syntax and semantics of EDT.

III. Formal Syntax and Semantics

An EDT specification consists of a set of tables. These tables are merged into a single table by taking a union of columns and rows adding empty cells wherever required. Due to space constraints only a few key elements of the formal syntax and semantics are given below.

A. Syntax

Each cell in a table is either empty or consists of a pattern expression. The syntax of a pattern expression is: $e := v|e;e|e\{op\ t\ s\}|e\{\ op\ n\}|(e)$ where-

- v is a value
- ; denotes sequence
- op $\in \{<, \leq, =, \geq, >\}$
- t is a floating point constant for time
- s indicates seconds
- n is an integer constant for multiplicity

B. Semantics

EDT employs a discrete unit of time and hence at any given time t, a string is present at each port, consisting of values from the port's domain or ϵ. ϵ represents the absence of any value at that time. The semantics of EDT define which rows of a given table match the input strings at a given time t. The row matching semantics assume cell matching predicates which are standard regular expression pattern-matching predicates. Each value v in a pattern is translated to $v \cdot \epsilon*$ and a time expression is translated to an appropriate ϵ^n. The first row of a row-sequence is matched if all the cells of the row match. Subsequent rows are matched if all previous rows match and current row also matches. Formally, matching of a row at time t is defined as:

$$m_r(r_i^j, t) \equiv \begin{cases} m_r^1(r_i^j, t) & \text{if } j = 1 \\ m_r^*(r_i^j, t) & \text{otherwise} \end{cases}$$

$$m_r^1(r_i^j, t) \;\equiv \forall\, c \;\cdot m^x(s_c^t, e_c^{i,j}) \;\vee \phi(c)$$

$$\begin{aligned} & m_r^*(r_i^j, t) \equiv \\ & \quad \exists t_1 < t \;\cdot m_r(r_i^{j-1}, t_1) \quad \wedge \\ & \quad\quad \forall c\cdot\ \phi^-(c) \quad \vee \\ & \quad\quad\quad \exists\ t_2 \cdot t_1 < t_2 \le t \ \wedge \\ & \quad\quad\quad\quad m(s_c^{t_2 \to t}, e_c^{i,j}) \ \wedge \\ & \quad\quad\quad\quad s_c^{t_1 \to t_2 - 1} = \epsilon* \ \wedge \\ & \quad\quad\quad\quad \exists c_1 \cdot t_2 = t_1 + 1 \end{aligned}$$

where -

$m_r(r, t)$	is a predicate that is true if row r matches at time t.
r_i^j	is the j^{th} row of row-sequence i
c, c_1	are columns
$e_c^{i,j}$	is the pattern expression of row-sequence i, row j and column c.
$m^x(s, e)$	is a predicate that checks if a suffix of string s matches the regular expression e.
$m(s, e)$	is a predicate that checks if the complete string s matches the regular expression e.
s_c^t	is the string corresponding to column c at time t
$s_c^{t_1 \to t_2}$	is the sub-string from t_1 to t_2 for column c
$\phi(c)$	checks for emptiness of a cell
$\phi^-(c)$	checks for emptiness of that cell and all corresponding cells in previous rows of the same row-sequence

Output of a row is triggered every time a row matches afresh, that is either the row did not match at the previous time unit or there is some non-ϵ value which caused the row to match. This is formalized by the predicate $o(r, t)$ which is true if the row r triggers an output at time t.

$$\begin{aligned} o(r, t) \equiv\, & m_r(r, t) \ \wedge \\ & (\neg m_r(r, t-1) \vee \exists c \cdot \neg\phi(c) \wedge v(c, t) \neq \epsilon) \end{aligned}$$

where $v(c, t)$ is the value present at time t in the input string for column c.

In case multiple rows are output enabled at the same time, outputs of each such row are triggered, and in case of a conflict in the outputs, it is considered as an error in the specification.

IV. Comparison with other notations

Several notations have been proposed to specify reactive systems. These can be classified either as textual, tabular or graphical, based on their concrete syntax, or based on their underlying formalism, as mathematical logic-based, state-based or stream-based. Tabular notations are the easiest to use and less error prone than the other two [1]. Interestingly, as per Leveson [1], even for requirements for which viewing the requirements as state-based is beneficial, using tables to specify the state-based requirements is better than using state-based graphical notations. Additionally, graphical notations need specialized editors.

Notations such as Z [6] require strong mathematical background which average software engineers and testers do not often possess. State-based notations such as graphical Statecharts [2], SCR [3], RSML [7], state-based synchronous notations like Esterel [8] and others [9] require the user to analyze and re-structure non state-based requirements to a state-based format. Stream-based notations such as timing diagrams [10] have the converse problem of users having to analyze and re-structure requirements described in state-based format to a stream-based format. Stream-based notations such as live sequence charts [11] are mainly used to specify interactions across processes, whereas EDT is used to specify interactions within a single process.

We compare EDT with SCR [3] and Stream-based I/O tables [4] using the Wiper Example given in Figure 1. EDT specification for this example is given in Table I.

In the SCR specification shown in Table II, the first table defines mode transitions and the other tables are event functions. To capture the first requirement, SCR requires three rows in the mode transition table. To capture the second and third requirements which are stream-based and have time ordering dependencies between inputs and outputs, the specifications are spread across many event tables in SCR. Additionally, an SCR user has to introduce auxiliary variables such as `cnt` and `starttime` to keep track of the count of `park` and `notpark` events and the time at which the first `park` event occurred. Introducing these variables requires analysis and re-structuring of the requirements. The Statecharts specification too needs auxiliary variables and states. We are not presenting the Statecharts specification here due to lack of space.

Stream-based I/O tables is a stream-based notation that has support for state-based requirements. A specification of the Wiper Example in this notation is given in Table III. Similar to SCR specifications, this too needs three rows to specify the first requirement and the pattern language of the notation is not powerful enough to specify the second and third requirements.

Unlike existing notations EDT has a simple yet powerful pattern language eliminating the need for complex concepts in semantics. For example, there is no need to introduce auxiliary variables and states for the second and third requirement.

V. Case Study

To evaluate the power of EDT and its ease of use we conducted a case study in which requirements of seven features from a real life automotive application were specified in EDT. We also compared the time required to specify in EDT with the time required to create Statemate Statecharts for the same features. The selected seven features had more than 200 requirements and they covered a wide variety of complex requirements of the body functionality of a vehicle. Each feature was split into multiple logic blocks and requirements documents informally describing the functionality of the logic blocks were available. Mature Statemate Statecharts models for all these features were available. Note that these models were created to formalize and test the requirements and not for code generation and hence, contained the same level of detail as the corresponding EDT specifications. We therefore used these models for comparison in our case study.

A. *Details of the Case Study*

We chose seven features from a real life automotive application, for the case study. We selected five testers who were undergraduates with automotive domain knowledge but without a strong mathematical background and unfamiliar with the selected features. Initially we trained three testers for two weeks on the EDT language and they in turn trained two others with a similar background for two weeks. From the requirements documents, the testers created an EDT specification corresponding to each logic block of each feature. Testers trained by us converted the requirements of sixteen logic blocks of four features to EDT specification in two days. The newly trained testers converted the remaining three features in one and half days. Using EDTTool the testers were able to generate test cases and expected output at the feature level from the EDT specifications of that feature's logic blocks. These test cases were executed on Statecharts models which were created by a separate team with expertise in both domain knowledge as well as creating Statecharts. The outputs were compared to check that the Statecharts specifications and EDT specifications were consistent. They also recorded and compared the time required to create EDTs and Statecharts.

Table IV presents the details and findings of our case study, including the number of logic blocks for each feature, number of input and output ports of each logic block and the time taken to specify a logic block. The table also gives time needed to convert the requirements of these features to Statecharts models.

In most cases, each independent requirement of a logic block mapped to one row-sequence in EDT. Complexity of a feature can be judged by the number of row-sequences needed to specify the feature. Since testers specified EDTs only at the logic block level, no single EDT was too big. For example, Rear Wiper feature was split into 6 logic blocks specifying the functionalities including Normal Wiping, Multi-Speed Wiping, Washer, Rain Sensing and Reverse Wiping. Although the feature had 57 requirements in all, no logic block level EDT had more than 15 rows. Capturing the requirements of rear wiper feature in EDT needed a total of 14 hours, whereas Statecharts modeling had taken 21.5 hours. Similarly, Statecharts modeling of Rear Defogger feature took 7.75 hours more than that of EDT specifications.

B. *Findings*

Our case study clearly showed that testers without a strong mathematical background were able to capture requirements with two weeks of training. The training time required was comparable to that required for Statecharts training and the time needed to create EDTs was 43% less. Further, when EDTTool generated test data was run on the Statecharts models, the testers detected a bug in the Rear Defogger feature which was already tested before this case study. Because of the success of the case study the team started using EDT for specifying other features as well and found several missing requirements in the document describing enhancements to the turn indicator feature.

TABLE IV. CASE STUDY DETAILS AND FINDINGS

Feature	Logic Block	Inputs/Outputs	Row-sequences	EDT Time Hrs	Statecharts Time Hrs	Savings %
Rear Wiper	LB1	4/1	13	2.75	4	31.25
	LB2	3/1	10	2.5	4	37.5
	LB3	4/2	8	2.75	4	31.25
	LB4	7/2	15	3	4	25
	LB5	4/1	5	1.5	2.75	45.45
	LB6	5/2	6	1.5	2.75	45.45
Rear Defogger	LB1	6/3	17	3.5	5	30
	LB2	4/3	20	3.5	6.5	53.84
	LB3	2/2	4	1.5	4.75	68.42
Auto Backdoor	LB1	1/1	4	1.5	5.5	72.72
	LB2	2/2	14	2.5	7	64.28
	LB3	2/1	4	1	4.75	78.94
Headlamp Washer	LB1	6/2	11	3	5	40
	LB2	6/2	26	4	6	33.33
	LB3	2/1	5	1	3.5	71.43
	LB4	2/1	3	1	3.5	71.43
Auto HiLo-Beam	LB1	6/2	15	2.5	4.75	47.37
Warning Buzzer	LB1	3/2	5	1.5	1.5	0
	LB2	3/1	5	0.5	1.5	66.67
	LB3	3/1	6	0.75	1.5	50
	LB4	3/1	5	0.5	1.5	66.67
	LB5	4/2	10	4	3	-33.33
	LB6	5/3	12	4	3	-33.33
	LB7	8/1	9	0.5	1.5	66.67
Shorting Pin	LB1	4/4	12	3	3	0

VI. CONCLUSION

In this paper, we presented a simple, user-friendly notation to specify requirements for reactive systems. Our case study showed that testers who did not have strong mathematical background were able to formally specify several requirements of various features with just two weeks of training. Time taken to create the EDTs was nearly half of that needed for creating Statecharts. Further, when EDT generated test data was run on the Statecharts models, the testers detected missing requirements in a feature and also found a bug in another feature that was already tested before. Going ahead we will be exploring scalable techniques to analyze EDTs for state reachability and also interesting coverage criteria that can help detect common implementation errors.

REFERENCES

[1] M. K. Zimmerman, K. Lundqvist, and N. Leveson, "Investigating the readability of state-based formal requirements specification languages," *International Conference on Software Engineering*, 2002.

[2] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, 1987.

[3] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj, "SCR*: A toolset for specifying and analyzing software requirements," *Computer Aided Verification*, pp. 526–531, 1998.

[4] J. Thyssen and B. Hummel, "Behavioral specification of reactive systems using stream-based I/O tables," *Software and Systems Modeling*, 2011.

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. SE-16, no. 4, pp. 403–414, Apr. 1990.

[6] J. Bowen, *Formal specification and documentation using Z: A case study approach*. International Thomson Computer Press, 1996, vol. 66.

[7] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese, "Requirements specification for process-control systems," *Software Engineering, IEEE Transactions on*, vol. 20, no. 9, pp. 684–707, 1994.

[8] G. Berry and G. Gonthier, "The esterel synchronous programming language: Design, semantics, implementation," *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.

[9] M. Herrmannsdörfer, S. Konrad, and B. Berenbach, "Tabular notations for state machine-based specifications," *Crosstalk*, vol. 21, no. 3, pp. 18–23, 2008.

[10] C. Antoine, B. L. Goff, and J.-E. Pin, "A graphic language based on timing diagrams," in *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK, UK: Springer-Verlag, 1993, pp. 306–316. [Online]. Available: http://dl.acm.org/citation.cfm?id=646831.707726

[11] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, pp. 45–80, 2001.