Stateflow to Tabular Expressions

Neeraj Kumar Singh
IRIT-ENSEEIHT
University of Toulouse
Toulouse, France
nsingh@enseeiht.fr

Mark Lawford
McMaster Centre for Software Certification
McMaster University
Hamilton, Ontario, Canada
lawford@mcmaster.ca

Thomas S. E. Maibaum
McMaster Centre for Software Certification
McMaster University
Hamilton, Ontario, Canada
tom@maibaum.org

Alan Wassyng
McMaster Centre for Software Certification
McMaster University
Hamilton, Ontario, Canada
wassyng@mcmaster.ca

ABSTRACT

Stateflow is a visual tool that is used extensively in industry for designing the reactive behaviour of embedded systems. Stateflow relies on techniques like simulation to aid the user in finding flaws in the model. However, simulation is inadequate as a means of detecting inconsistencies and incompleteness in the model. *Tabular Expressions* (function tables) have been used successfully in software development for more than thirty years. Tabular expressions are also visual representations of functions, but include the important properties of *completeness* and *disjointness*. In other words, a tabular expression is well-formed only when the input domain is covered completely (completeness), and when there is no ambiguity in the behaviour described by the tabular expression (disjointness). The goal of our work is to use the completeness and disjointness properties of well-formed tabular expressions to aid us in establishing those properties in Stateflow models. From the Stateflow models, we generate a new kind of tabular expression that includes extended output options. We use the informal Stateflow semantics from MathWorks documentation as the basis for generating our tabular expressions. The generated tabular expressions are then used to guarantee completeness and disjointness. We provide a transformation algorithm that we plan to implement in a tool to automatically generate tabular expressions from Stateflow models.

CCS Concepts

•**Software and its engineering** → **Consistency; Completeness; Formal software verification;** *Software notations and tools; Requirements analysis; Software design engineering;*

Keywords

Stateflow; Tabular Expressions; completeness; disjointness

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SoICT 2015, December 03-04, 2015, Hue City, Viet Nam

© 2015 ACM. ISBN 978-1-4503-3843-1/15/12. . . $15.00

DOI: http://dx.doi.org/10.1145/2833258.2833285

1. INTRODUCTION

To hide complexity and to minimize the introduction of errors, the software development process has changed dramatically over the last few years. Software engineers and developers prefer to use model-based development to design complex systems, where graphical blocks and symbols can represent high level abstraction of a system. Model-based development supports various tools based on graphical blocks and symbols, which are used by industries for design, simulation, and testing. Matlab is one of the tools that is used worldwide by the automotive, medical and avionic industries for developing their products.

Stateflow is a complex language with numerous features, but does not have formal semantics. Its documentation [18] describes informal semantics of the Stateflow execution based on the Matlab simulation environment using various examples. Moreover, Stateflow does not support a formal verification tool for checking properties like consistency, completeness, and disjointness. The existing problems in Stateflow offer an opportunity to investigate techniques that could provide such properties, as well as the capability to verify a model's consistency, taking into account Stateflow's informal semantics. In this paper, we propose the transformation of Stateflow models into tabular expressions to address these problems. Tabular expressions have precise semantics [17, 16], and verification capabilities [8], and have been used for many years in industry [11, 32, 31]. We use the Stateflow informal narrative semantics to capture precisely the order of execution of different components of the Stateflow models. The generated tabular expressions of a Stateflow model shows precise behavior of the system based on tabular semantics by preserving the properties of disjointness and completeness. This operational approach is able to express complicated behaviours of a system in tabular expressions that are likely to be lost in a state transition table. Moreover, the generated tabular expressions include extended conditions, and behaviours that are not explicitly clear in the original Stateflow models. These extended conditions and missing behaviours are identified during the transformation process. The tabular expressions satisfy required properties like consistency, completeness and disjointness and this can be verified by existing tools [8]. The generated tabular expressions contain detailed information that can be used further for an-

Partially supported by: The Ontario Research Fund, and the National Science and Engineering Research Council of Canada.

alyzing the specific properties of a particular component of a very large system. Moreover, the tabular expressions can be used by software engineers and developers to understand desired system behaviours more precisely.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 introduces the characteristics of Stateflow and tabular expressions. Section 4 shows the transformation process from Stateflow to tabular expressions. Experiments in Section 5 demonstrate the applicability and usefulness of our approach. Section 6 discusses the paper, and Section 7 concludes the paper along with potential future work.

2. RELATED WORK

Since the late 1950s, tables have been used for various purposes like analyzing computer code, and developing requirements documents. In the 1960s, tables first appeared in the literature to show their usability in software development [4, 20, 21]. All these tables can be distinguished by their names and forms such as decision tables, transition tables, etc.

Parnas and others introduced tabular expressions for developing the requirements document for the A-7E aircraft [13, 6, 12, 22] in work undertaken for the US Navy. Parnas was the most influential person to apply tabular expressions in documenting software [27]. Later, tables were used by many others, including at Bell Laboratories, and the US Air Force. Starting in the late 1980s tabular notations were applied by Ontario Hydro in developing the shutdown systems for the Darlington Nuclear Plant [32, 31, 2, 25]. Formal semantics of tabular expressions have been proposed by Parnas [23] and other researchers [14, 15, 17, 16]. A slightly outdated survey on tabular expressions is available in [16].

Stateflow is a graphical modelling language that shares many features with Statecharts [10]. Stateflow semantics are completely deterministic while Statecharts semantics can handle non-determinism. For instance, the execution order among parallel states in Stateflow is sequential. Moreover, Stateflow has its own modelling features such as defining that a condition action of a transition occurs before its source state becomes inactive. Therefore, Statecharts semantics are not applicable to Stateflow. However, several papers have reported work on formal verification of Stateflow models. Banphawatthanarak et al. [3] used the SMV model checker to verify Stateflow models, in which they had not considered modelling multiple hierarchy levels of states. Cavalcanti [5] proposed verification of Stateflow models using the Circus specification language. Scaife et al. [28] converted a subset of Stateflow into a synchronous language, Lustre, in which inter level transitions were not allowed. The operational semantics of Stateflow proposed by Rushby et al. [9] have been used as a foundation for developing a prototype tool for formal analysis of Stateflow designs.

A transition table is a tabular presentation of a Stateflow model, that shows only the conditions and actions of transitions, and state information. This table does not contain other information like entry, exit and during actions. Due to this lack of information in the transition table, it cannot be used for analyzing the consistency, completeness and disjointness properties of a Stateflow model. In order to analyze these properties, our approach is to transform Stateflow models into tabular expressions or function tables using the narrative informal semantics of Stateflow that contain detailed information about the actions and transitions. In this way, entry, exit, during and condition actions also appear in their proper order in the produced tabular expression. The tabular expression allows a careful inspection of the Stateflow model through identifying missing conditions and desired behaviours by checking completeness and disjointness.

3. PRELIMINARIES

3.1 Stateflow

In this section, we describe how the Stateflow language can be used for designing complex behaviours of a system. Simulink is a block diagram environment for modelling, simulating and analyzing a system, whereas Stateflow is an interactive tool that can be used for modelling the behaviours of reactive systems. Stateflow models can be included in a Simulink model by placing them in Stateflow blocks. The syntax of Stateflow is similar to Statecharts [10]. It supports the notions of hierarchy (states may contain other states), concurrency (executes more than one state at the same time), and communication (broadcast mechanism for communicating between concurrent components). It also includes complicated features like inter level transitions, complex transitions through junctions, and event broadcasting. These features allow us to design complex systems effectively and concisely.

Fig. 1 depicts a simple Stateflow diagram, which contains all the basic components of Stateflow. A Stateflow model consists of a set of states connected by arcs called transitions. Each state has a name and can be decomposed to model a hierarchical state diagram. States can have different types of actions that can be executed in a sequential order. These action types are *entry*, *exit*, *during*, and *on event_name*.

There are two types of decomposition for a state: 1) OR-states and 2) AND-states. In the Stateflow diagram, the OR-states are indicated by a solid border, while AND-states are indicated by a dashed border. Each hierarchy level of the Stateflow presents either OR-states or AND-states decomposition (see Fig. 1). It should be noted that, AND-states do not allow pure concurrency because the Stateflow model runs on a single thread, therefore only one AND-state executes at a time. Each AND-state is executed sequentially according to the execution order, which can depend on the states' geometric positions or manually assigned priorities. A Stateflow model can have both data and event input/output ports that can be defined as local as well as external.

A transition is an arc that connects two states, where one state is source (state) and another is destination (state). A transition can be characterized by a label that can consist of event triggers, conditions, condition actions, and transition actions. The '?' character is the default empty label for transitions. A transition label can have the following general format:

event [condition] condition_action/transition_action

Each part of the transition label is optional. The event specifies an event that causes the transition to be taken, provided the condition, if defined, is true. The absence of an event indicates that the transition is taken upon the occurrence of any event. Multiple events can be specified using the OR logical operator (see Fig. 1). A condition is a boolean expression that indicates that the transition can be taken if the condition expression is true. A condition action is enclosed in curly braces({}) and executes as soon as the condition (guard) becomes true before the transition destination has been determined to be valid. Absence of a condition expression is implied by true, and the condition action is executed only if the transition is true. The transition action is always executed after the transition destination has been determined upon validation of the provided condition. Each transition also has a priority of execution, which can be determined based on the geometric position and hierarchy level.

Stateflow uses two different types of junctions named as *connective* and *history* junctions (see Fig. 1). The connective junctions

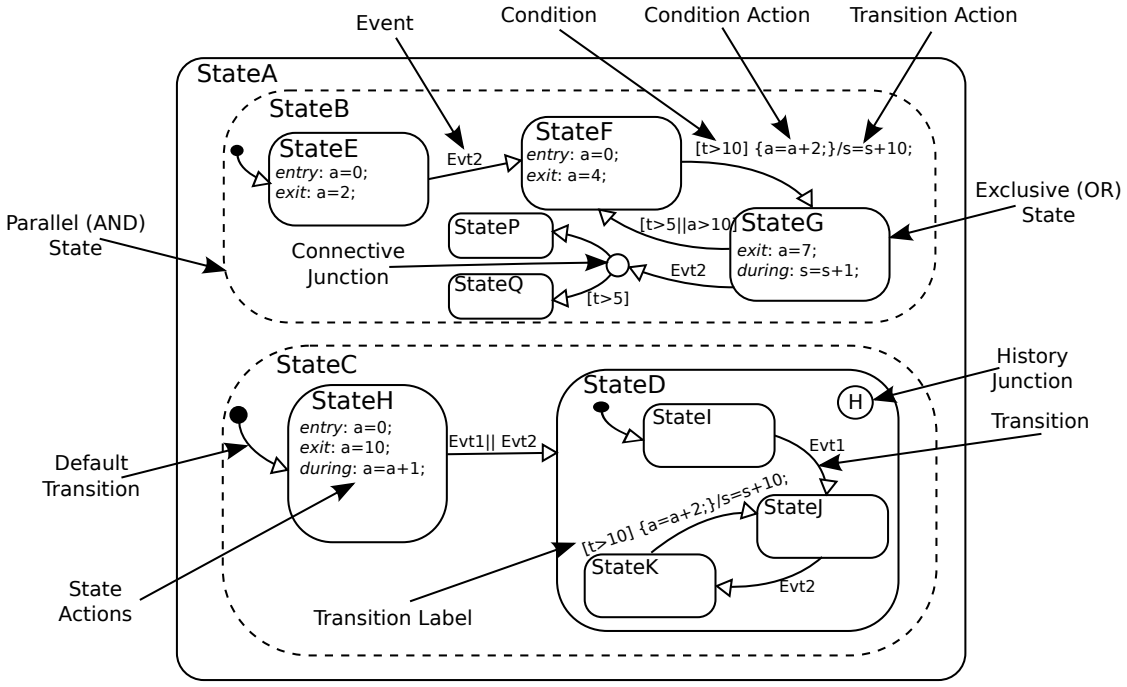

Figure 1: Stateflow Example.

provide alternative transitions paths for a single transition or desired system behaviour. They are often used to model an *if-then-else* structure, a *case* structure or a *for loop*. The history junctions record historical activity information of states or superstates. A history junction of a superstate stores the state or substate which was active when the superstate was exited.

3.2 Tabular Expressions

In late 1970s, Parnas et al. [24, 25, 13] used tables to specify the software system requirements for expressing complex behaviours through organizing the relation between input and output variables. These tables were used simply to describe the system requirements unambiguously. Parnas formally defined ten different types of tables for different purposes using functions, relations, and predicates [23]. Parnas also called these tables tabular expressions because the tables use mathematical expressions and recursive definitions. Some foundational works reported formal semantics [17], table transformation, and composition of tables [30]. The formal semantics of tables specify the precise meaning that helps to maintain the same level of understanding when tables are used by various stakeholders. Similarly, table transformation can be used to get a desired system behaviour under various system situations, and the composition of tables can be used to integrate different tables to obtain the final complex behaviour. These tables have been used by several international projects like Ontario Hydro in the Darlington Nuclear Plant [7], and the US Naval Research Laboratory [11], etc.

Tabular expressions [26, 27] are not only effective visually, and an effective and simple approach to documenting the system requirements by describing conditions and relations between input and output variables, they also facilitate preserving essential properties like completeness and disjointness. In our work, for transforming Stateflow models into tabular expressions, we use *horizontal condition tables (HCT)* shown in Fig. 2. The HCT table contains a group of columns for input conditions and a group of columns for output results. However, the input column may be sub-divided to specify multiple sub-conditions. The tabular structure highlights the structure of predicates, and adjoining cells are considered to be ANDed (see Fig. 2) that can be interpreted in the tabular structure as a list of "*if-then-else*" predicates.

Condition		f_name
Condition 1	Sub Condition 1	Result 1
	Sub Condition 2	Result 2
....		....
Condition n		Result n

Figure 2: Horizontal Condition Table or Function Table.

4. RULES FOR TRANSFORMING STATEFLOW INTO TABLES

Stateflow models can be designed using graphical components (states and transitions) as well as state transition tables. Both the graphical representation and transition table of a Stateflow model often contains inconsistencies and may be incomplete. In this section, we describe the general rules for transforming Stateflow models into tabular expressions according to the narrative executional semantics of Stateflow, in order to analyze complex behaviours, and to guarantee essential properties like consistency, completeness and disjointness through discovering missing information. This missing information is often considered as not to be part of the system design. In this paper, we consider simple Stateflow models that can allow only exclusive (OR) states without hierarchy. But the simple

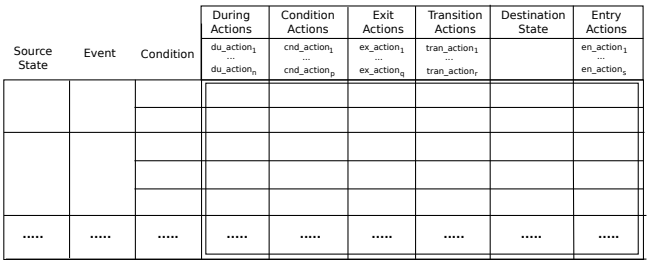

Figure 3: Table Architecture for Stateflow.

Stateflow models do allow *entry*, *during*, *exit*, *condition* and *transition actions*. In the following sections, we describe an architecture of the table and a set of rules for transforming Stateflow models into tabular expressions.

4.1 Tabular Expression Architecture for Stateflow

To preserve the narrative semantics of Stateflow, we use output columns of HCT in a specific order. Fig. 3 presents an architecture of the tabular expressions, which contains the elements of Stateflow model. In the Fig. 3, first three columns contain *Source State*, *Event*, and *Condition* of a transition. The cells of these columns can be further split into two or more cells as per the requirement during the transformation process to cover the possible scenarios for complex properties. For example, the conditions of a transition do not contain negation of the given conditions that must be identified during the generation of tabular expressions. The rest of the columns of Fig. 3 contain ordered *During Actions*, *Condition Actions*, *Exit Actions*, *Transition Actions*, *Destination State* and *Entry Actions*. Let n be the number of user defined program variables appearing in a Stateflow block. All of the Actions columns may be split into many sub-columns to record the possible actions on these n variables. In the Stateflow, a variable can be used many places. For instance, a variable can be in both *Exit Action* and *Transition Action*, and these actions matter on the execution order. To handle the multiple times appearance of the same variable, we apply *specialization* to distinguish all the Stateflow variables. For example, we use some extra tags on each variable name as per the type of action, such as, for *Exit Action* variable name can be *ex_var*, where *exit* belongs to the *Exit Action* and *var* is a variable name. We also use some fixed keywords like NONE, NC, and CALL that can be used to fill the cells of the table. The meanings of these keywords are as follows: NONE when an action is not given, NC for No Change (a value is not modified after execution), and CALL for function or events calling in the actions.

It should be noted that we have considered *During Action* before the rest of the action columns and destination state, because the during action(s) of the source state is only allowed when all the transitions are invalid from the source state to all the possible destination states. In this case, we need to fill only during action(s) and the destination state with the same value as the source state, and the rest of the columns do not change.

4.2 Transformation Rules

This section provides a set of rules for transforming the Stateflow models into tabular expressions. We have identified these rules through our understanding of the Stateflow executional semantics, and the rules are then applied to transform the Stateflow models. Our current rules deal with only simple Stateflow models in which parallel and hierarchy levels of modelling are not included. However, this is our first step in applying tabular expressions to analyze Stateflow models through transformation. In the future, we will provide transformation rules for other complex Stateflow models, including parallel and hierarchical models. These transformation rules will extend the simple Stateflow rules to cover all the required functional behaviors for checking the properties of disjointness and completeness. In fact, the simple Stateflow covers all the essential information that can be required by the parallel and hierarchy Stateflow models. A set of rules can be applied iteratively to cover all the states and transitions for transforming the simple Stateflow models into the specific table format we have defined (see Fig. 3). The current rules are given in a narrative style as follows:

1. **Default / Normal Transition:** Identify a *default transition* or *normal transition*.
2. **Source State:** The next step is to identify a *source state* of the identified transition and add it into the source state column of the table (in case of default transitions set source state to START).
3. **Event:** In this step, we analyze the label of an identified *outgoing transition* from the source state. If this transition contains one or more *events* then we add them into the event column of the table, otherwise we place *TRUE* into the event column of the table. (The *event* element is optional so it can be absent sometimes).
4. **Condition:** Identify the given *condition(s)* of the transition and add it into the condition column of the table (if *condition(s)* are absent then set the condition to *TRUE*).
5. **Condition Action:** The next step is to identify the given *condition action(s)* of the identified transition, and add it into the condition action column of the table (if the *condition action* is absent then set the condition action to NC). If more than one condition action is identified, then with the variable's name use *specialization* and all these new specialized variables will be added into multiple sub-columns of the action column of the table. If the given condition action is an *event* or predefined function then use *CALL function* or *event*.
6. **Exit Action:** The next step is to identify the *exit action(s)* of the source state, add it into the column of exit action of the table. If more than one *exit actions* is identified then with the variable's name use *specialization* with exit action(s) to fill other sub-columns of the exit action column. If the given exit action is an *event* or predefined function then use *CALL function* or *event*.
7. **Transition Action:** This step is used to identify the *transition action(s)* and add it into the transition action column of the table. If more than one *transition action* is identified, then with the variable's name use *specialization* with transition action(s) to fill other sub-columns of the transition action. If the given transition action is an *event* or predefined function then use *CALL function* or *event*.
8. **During Action:** The next step is to identify the *during action(s)* of the source state, and add it into the during action column of the table. During actions are included only when there are no valid transitions available, causing the state to remain active (i.e. source state = destination state). If more than one *during action* is identified, then with the variable's name use *specialization* with during action(s) to fill other sub-columns of the during action. If the given during action

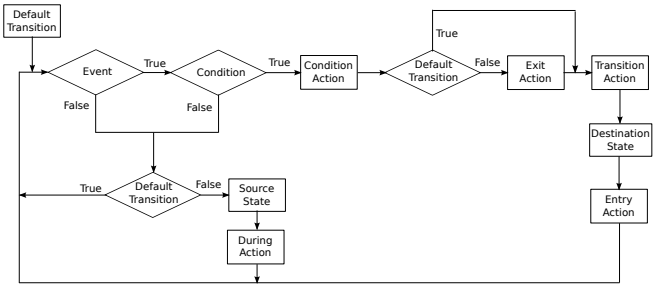

Figure 4: Building Tables from Stateflow : Abstract Flowchart.

is an *event* or predefined function then use *CALL function* or *event*.

9. **Destination State:** In this step, we identify the *destination* or *target state* and add it into the destination state column of the table. For any during action(s), the destination state must be the same as the source state.

10. **Entry Action:** Finally, we identify the *entry action(s)* in the *target* or *destination state* and add it into the entry action column of the table. If more than one *entry action* is identified, then with the variable's name use *specialization* to fill other sub-columns of the entry action. If the given entry action is an *event* or predefined function then use *CALL function* or *event*.

11. **Completeness of Events and Conditions:** Corresponding to the split cells in the columns of event and condition, make respective entries in the entire row through splitting the rest of the cells in each column. For example, for input domain completeness related to an event, add the negation of the given *event(s)* in the same column by splitting the cell into two or more cells (if an event does not exist then this step is not required); and for completeness of the input domain related to conditions, add the negation of the *condition(s)* in the same column by splitting the cell into two or more cells. It is important that the identified *condition(s)* and its negation be added into the table for the identified *event* and its negation as well. It should be noted that in the case of a *during action* all the cells will use *NC* as a cell value except a destination state cell entry, which will be the same as the source state.

12. Goto Step 1.

The above rules apply iteratively to transform the Stateflow models into tabular expressions. The textual description of the rules may appear complex, but the transformation process is actually quite straightforward. To better understand this process, we have included an abstract view of the transformation process using a flowchart (see Fig. 4).

5. CASE STUDY

In this section, we apply the transformation rules for generating the tabular expressions from Stateflow models. We have applied this transformation process to several case studies and industrial examples, in which a system is developed without using hierarchical and parallel components of Stateflow. However, we would like to share our experience with a real-time case study related to a robotic system, which is taken from [1]. It is a small case study that is developed by researchers for the *Field Robot Event 2007*. The given stateflow model is used for designing the behaviour of a robotic system. Fig. 5 presents a basic exclusive (OR) Stateflow chart of the robotic system. We applied our transformation process to obtain the tabular expressions from the selected Stateflow models of the robotic system that captures all the possible activities including different types of actions, transitions and state information. Fig. 6 shows the generated tabular expression. In Fig. 5, a *default transition* is identified that is used initially to start the process for generating the tabular expressions (see Fig. 6). In Fig. 6, the first row of the column *Source State* contains $State = START$ to present the default transition of the robotic stateflow. In the case of a normal transition the column *Source State* contains the name of the source state in place of START. For example, the next row of the same column contains $State == ManualDrive$. The next column (*Event*) of the table contains $TRUE$ that shows there is no explicit event associated with the default transition. The default transition of the robotic stateflow has three transitions connected by a junction. These three transitions have transition conditions that are given in the column *Condition* of the table in three separate rows. For example, $stateReq == 1$ is the condition for the first transition, $stateReq == 2$ OR $stateReq == 3$ OR $stateReq == 5$ OR $stateReq == 12$ is the condition for the second transition, and $stateReq == 4$ is the condition for the third transition. The default transition does not have a source state, therefore the *During Actions* column of the table contains NC. Similarly, the transitions do not have condition actions and exit actions so that the next two columns, *Condition Actions* and *Exit Actions* are set to NC. The next column, *Transition Action*, contains a list of actions that are placed in three separate rows, one for each transition. For example, $drivingDirOut = 1; turnCounter = 1;$ is a list of sequential actions for the first transition, *drivingDirOut = 1; turnCounter = 1; state = stateReq;* is a set of sequential actions for the second transition and $drivingDirOut = 1; turnCounter = 1; state = 4;$ is a list of sequential actions for the third transition. To distinguish these variables *drivingDirOut*, *turnCounter* and *state* from other actions (e.g. exit, during), we can use *specialization*, such as *tran_drivingDirOut*, *tran_turnCounter* and *tran_state*, in which *tran'* indicates for transition action (see Fig. 3). The next column, *Destination State*, is used to set a destination state corresponding to the selected transition. In the example, three destination states *ManualDrive*, *drivingRow* and *Turning* are set in the *Destination State* column of the table for each transition. Finally, the last column, *Entry Actions*, is used to place a list of entry actions of the destination state in the table. In our running example, there is no entry action in any of the three destination states $ManualDrive$, $drivingRow$ and $Turning$, therefore we enter NC in each row of the transitions. In order to satisfy completeness properties in the table, we need to add an extra row with the negation of the given transitions conditions. In this case, we make the *Destination State* the same as the source state, the *During Actions* column contains a list of during actions, and the rest of the columns of the table are set to NC. In our example, we use negation of all the conditions of the given transitions as $((stateReq == 1)$ OR $(stateReq == 2$ OR $stateReq == 3$ OR $stateReq == 5$ OR $stateReq == 12)$ OR $(stateReq == 4))$ and the rest of the columns are set to NC because there is no during action in the source state. The *Destination State* column must also be the same as the source state. In a similar way, we applied all the transformation rules to cover all remaining states and transitions to produce the tabular expression in (see Fig. 6). The generated table contains much more significant information than the transition table. In the generated table, the condition column has some highlighted boldface conditions. These conditions do not exist in the Stateflow model. These conditions

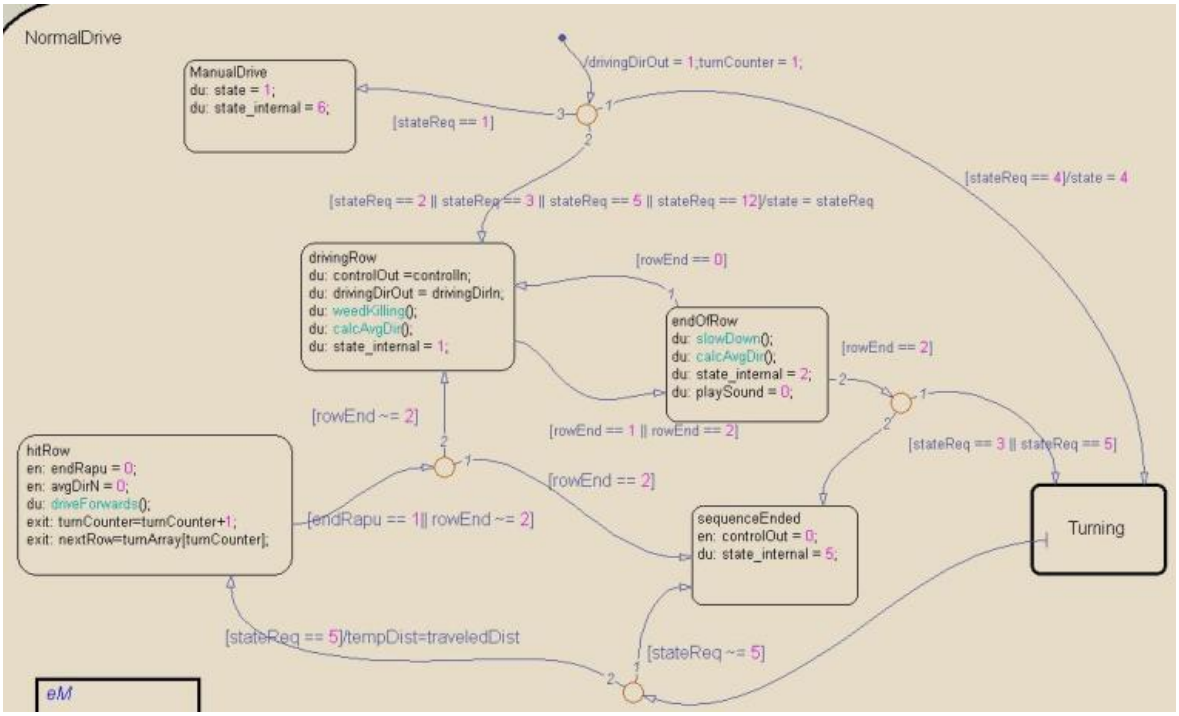

Figure 5: Stateflow models of a Robotic System [1].

are discovered during the transformation process used to generate the tabular expression, since we have to ensure that the tabular expression is disjoint and complete. It should be noted that the associated row for each condition is produced in the table according to 'discovered' conditions. We have used this transformation process on several other Stateflow models to generate tabular expressions. We have observed through this experiment that most of the time a Stateflow model does not satisfy disjointness and completeness properties, which are essential for developing critical systems. Our experiment results have provided enough evidence to assert that often a Stateflow model does not satisfy essential properties like completeness and disjointness, and this is not safe for developing a safe and dependable system. Checking properties like completeness, disjointness and consistency are not easy without tool support. Therefore, we have used our previously developed tool [8] to check completeness, disjointness and consistency.

6. DISCUSSION

Simulink is a visual notation that is used by many industries, including automotive, avionics and medical, to design complex systems. Stateflow is a graphical component of Simulink to model the reactive behaviour of a system. Simulation techniques are a common approach for checking the validity of, and tracing errors in Stateflow models. Simulink Design Verifier [19] is an extension of the MathWorks Matlab / Simulink tool set that uses formal methods to identify design errors in models. These errors include dead logic, integer overflow, division by zero, and violations of design properties and assertions. Moreover, Simulink Design Verifier is also used to analyze Simulink models to check the correctness of given properties. These properties are specified directly in Simulink and Stateflow in the form of assertions. Simulink Design Verifier uses a prover plug-in to prove the given properties by searching for possible values for Simulink and Stateflow functions. It cannot be used to check unsupported elements that may cause incomplete analysis or that will be stubbed out during analysis. According to [9], Simulink does not support an interpretation of concurrency in terms of a nondeterministic interleaving of concurrent events. Hence, the Simulink Design Verifier does not check safety properties related to concurrency, like race conditions. Moreoever, Simulink Design Verifier does not support checks for *disjointness* and *completeness* in the system requirements. In fact, simulation, like testing, cannot guarantee the safe and reliable behaviour of a system that is developed with the help of Stateflow. It is obviously useful to be able to complement these approaches by mathematical analysis. Unfortunately, we do not have formal semantics for Stateflow. It should be noted that the operational semantics presented in [9] are not sufficient to check disjointness and completeness properties. We do have some narrative semantics provided by MathWorks [18]. The narrative semantics are not easy to comprehend and apply, and this is why practitioners have relied on simulation to understand the meaning of Stateflow models. As we have demonstrated, Stateflow models often lack of disjointness and completeness properties, and it is easy to miss subtle behaviours in the graphical presentation of a complex and large Stateflow model, because the developer has to keep in mind the complex semantics governing the ordering of actions. Regulators and certification bodies are striving for reliable techniques [29] to guarantee the safe behaviour of systems that are developed using Stateflow. In this context, we have proposed the idea of generating a tabular expression from a Stateflow model to address the described issues. In this paper, we have presented a list of translation rules to generate a table that contains more significant information than the Stateflow model from which it was derived. This approach has three primary benefits: firstly, we have spent the time and effort necessary to understand the informal semantics provided by MathWorks [18], and will generate a consistent interpretation of Stateflow models, compared with what happens now when different groups have slightly different inter-

Source State	Event	Condition	During Actions	Condition Actions	Exit Actions	Transition Actions	Destination State	Entry Actions
State == START	TRUE	stateReq == 1	NC	NC	NC	drivingDirOut=1; turnCounter=1;	ManualDrive	NC
		stateReq == 2 OR stateReq == 3 OR stateReq == 5 OR stateReq == 12	NC	NC	NC	drivingDirOut=1; turnCounter=1; state=stateReq;	drivingRow	NC
		stateReq == 4	NC	NC	NC	drivingDirOut=1; turnCounter=1; state=4;	Turning	NC
		¬ ((stateReq == 1) OR (stateReq == 2 OR stateReq == 3 OR stateReq == 5 OR stateReq == 12) OR (stateReq == 4))	NC	**NC**	**NC**	**NC**	State	NC
State = ManualDrive	NONE	NONE	state=1; state_internal=6;	NC	NC	NC	State	NC
State == drivingRow	TRUE	rowEnd == 1 OR rowEnd == 2	NC	NC	NC	NC	endOfRow	NC
		¬(rowEnd == 1 OR rowEnd == 2)	controlOut=controlIn; drivingDirOut=drivingDirIn; CALL weedKilling(); CALL calcAvgDir(); state_internal=1;	NC	NC	NC	State	NC
State == endOfRow	TRUE	rowEnd == 0	NC	NC	NC	NC	drivingRow	NC
		rowEnd == 2 (stateReq == 3 OR stateReq == 5)	NC	NC	NC	NC	Turning	NC
		rowEnd == 2 ¬(stateReq == 3 OR stateReq == 5)	NC	NC	NC	NC	sequenceEnded	controlOut=0;
		¬ ((rowEnd == 0) OR (rowEnd == 2))	CALL slowDown(); CALL calcAvgDir(); state_internal=2; playSound=0;	NC	NC	NC	State	NC
State == sequencEnded	NONE	NONE	controlOut=0; state_internal=5;	NC	NC	NC	State	NC
State == Turning	TRUE	¬(stateReq == 5)	NC	NC	NC	NC	sequenceEnded	controlOut=0;
		(stateReq == 5)	NC	NC	NC	tempDist= traveledDist	hitRow	endRapu=0; avgDirN=0;
State == hitRow	TRUE	endRapu == 1 (rowEnd == 2)	NC	NC	turnCounter= turnCounter+1; nextRow= turnArray[turnCounter];	NC	sequenceEnded	controlOut=0;
		endRapu == 1 ¬(rowEnd == 2)	NC	NC	turnCounter= turnCounter+1; nextRow= turnArray[turnCounter];	NC	drivingRow	NC
		¬(endRapu == 1)	CALL driveForwards();	NC	NC	NC	State	NC

Figure 6: Generated tabular expressions of the Robotic System.

pretations of the informal semantics; secondly, the generated table is very easy to comprehend and it contains self explanatory details for system behaviour according to those informal semantics; and thirdly, the generated table satisfies disjointness and completeness properties. Thus, the generated tabular expression of a complex and large Stateflow model shows system behaviour equivalent to the Stateflow narrative semantics in a way that makes the complete behaviour more readily understandable, and also satisfies disjointness and completeness properties on the input predicates. This approach can assist in the construction, clarification, and validation of Stateflow models. It should be noted that during the generation of tabular expression from the robotic Stateflow model, we have found some unwanted or incomplete transition conditions, which may allow some undesired behaviours. In fact, these unwanted or incomplete conditions prevent the generated table from being complete and disjoint. We have rewritten the transition conditions without changing the original behaviour to achieve disjointness and completeness properties in our generated table. However, we are still investigating such types of unwanted or incomplete design patterns in other Stateflow models that can be removed with the help of our proposed solution. Moreover, this approach has the potential to help regulators and certification bodies asses the quality of systems that are developed using Stateflow models.

7. CONCLUSION AND FUTURE WORK

The use of model based development is growing extremely rapidly, and some of the associated tools hide important system complexities. These tools may provide a very rich set of graphical block diagrams or symbols for developing complex behaviours. Matlab/Simulink is one of the tools that has been adopted by many industries for developing complex products. Stateflow is a component of Simulink, and its narrative executional semantics cannot guarantee essential properties like completeness, disjointness and consistency, and often leads to the introduction of design flaws during system modelling.

In this paper, we proposed the idea of transforming Stateflow models into tabular expressions, in order to analyze the completeness, disjointness, and consistency of Stateflow models. We are interested primarily in guaranteeing completeness and disjointness properties of Stateflow. Our proposed approach for transforming Stateflow models into tabular expressions can identify missing behaviour to make it complete, and to identifying overlapping behaviours, so that we can make it disjoint. Moreover, we are still developing semantics of the table to cope with the action columns

(not covered by tabular expression semantics). At the present time, we have transformed the Stateflow models into tabular expressions manually, but we have used a formal tool for checking consistency, completeness and disjointness properties.The given transformation rules in this paper are applicable for generating tabular expressions from simple Stateflow models in a restrictive way. Our current transformation rules do not cope with hierarchy and parallel state models, including connective and history junctions. However, the transformation rules do preserve the simulation semantics of the Stateflow behaviour in the defined architecture of the table, and are an indication of what we should be able to achieve once we have built a more complete set of rules. The proposed transformation rules are well suited to automatic processing.

In future work, we plan to investigate rules for transforming hierarchical level Stateflow models and parallel Stateflow models into tabular expressions, and then develop a tool to automate the transformation process. Another possible direction to extend this work is to consider the timing behaviour in Stateflow models.

8. REFERENCES

[1] http://autsys.aalto.fi/en/fieldrobot2007.

[2] G. Archinoff, R. Hohendorf, A. Wassyng, B. Quigley, and M. Borsch. Verification of the shutdown system software at the darlington nuclear generating station. In *International Conference on Control and Instrumentation in Nuclear Installations*, Glasgow, UK, 1990.

[3] C. Banphawatthanarak, B. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 581–586, 1999.

[4] H. N. Cantrell, J. King, and F. E. H. King. Logic-structure tables. *Commun. ACM*, 4(6):272–275, June 1961.

[5] A. Cavalcanti. Stateflow diagrams in circus. *Electron. Notes Theor. Comput. Sci.*, 240:23–41, July 2009.

[6] P. Clements. *Function Specifications for the A-7E Function Driver Module*. NRL Memorandum Report. Defense Technical Information Center, 1981.

[7] D. Craigen, S. Gerhart, and T. Ralston. Case study: Darlington nuclear generating station. *IEEE Softw.*, 11(1):30–39, 28, Jan. 1994.

[8] C. Eles and M. Lawford. A tabular expression toolbox for matlab/simulink. In *NASA Formal Methods*, pages 494–499, 2011.

[9] G. Hamon and J. Rushby. An operational semantics for stateflow. *Int. J. Softw. Tools Technol. Transf.*, 9(5):447–456, Oct. 2007.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[11] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. Scr: A toolset for specifying and analyzing software requirements. In A. Hu and M. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 526–531. Springer Berlin Heidelberg, 1998.

[12] K. Heninger. Specifying software requirements for complex systems: New techniques and their application. *Software Engineering, IEEE Transactions on*, SE-6(1):2–13, 1980.

[13] K. Heninger, J. Kallander, and S. J. E. Parnas D. L. *Software Requirements for the A-7E Aircraft*. NRL Memorandum Report 3876. Naval Research Laboratory, 1978.

[14] R. Janicki. Towards a formal semantics of parnas tables. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 231–240, New York, NY, USA, 1995. ACM.

[15] R. Janicki, D. Parnas, and J. Zucker. Tabular representations in relational documents. In C. Brink, W. Kahl, and G. Schmidt, editors, *Relational Methods in Computer Science*, Advances in Computing Sciences, pages 184–196. Springer Vienna, 1997.

[16] R. Janicki and A. Wassyng. Tabular expressions and their relational semantics. *Fundam. Inform.*, 67(4):343–370, 2005.

[17] Y. Jin and D. L. Parnas. Defining the meaning of tabular mathematical expressions. *Science of Computer Programming*, 75(11):980 – 1000, 2010. Special Section on the Programming Languages Track at the 23rd {ACM} Symposium on Applied Computing 08.

[18] Mathworks. *Stateflow and Stateflow Coder, User's Guide*, 2003.

[19] Mathworks. *Simulink Design Verifier, User's Guide*, 2011.

[20] M. Montalbano. Tables, flow charts, and program logic. *IBM Syst. J.*, 1(1):51–63, Sept. 1962.

[21] R. C. Nickerson. An engineering application of logic-structure tables. *Commun. ACM*, 4(11):516–520, Nov. 1961.

[22] D. L. Parnas. A generalized control structure and its formal definition. *Commun. ACM*, 26(8):572–581, Aug. 1983.

[23] D. L. Parnas. Tabular representation of relations. Technical report, McMaster University, 1992.

[24] D. L. Parnas. Inspection of safety-critical software using program-function tables. In *IFIP Congress (3)*, pages 270–277, 1994.

[25] D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of Safety-Critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, June 1991.

[26] D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. Comput. Program.*, 25(1):41–61, Oct. 1995.

[27] D. L. Parnas, J. Madey, and M. Iglewski. Precise documentation of well-structured programs. *IEEE Trans. Softw. Eng.*, 20(12):948–976, Dec. 1994.

[28] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of simulink/stateflow into lustre. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 259–268, New York, NY, USA, 2004. ACM.

[29] N. K. Singh. *Using Event-B for Critical Device Software Systems*. Springer-Verlag GmbH, 2013.

[30] M. von Mohrenschildt. Algebraic composition of function tables. *Formal Aspects of Computing*, 12(1):41–51, 2000.

[31] A. Wassyng and M. Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. In *FME*, pages 133–153, 2003.

[32] A. Wassyng, M. Lawford, and T. S. E. Maibaum. Software certification experience in the canadian nuclear industry: lessons for the future. In *EMSOFT*, pages 219–226, 2011.