Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Formal Modeling and Validation of Stateflow Diagrams

Chunqing Chen[1]⋆, Jun Sun[2], Yang Liu[3], Jin Song Dong[4], Manchun Zheng[4]

[1] Hewlett-Packard Laboratories Singapore
e-mail: `chunqing.chen@hp.com`
[2] Singapore University of Technology and Design
e-mail: `sunjun@sutd.edu.sg`
[3] Temasek Laboratories, National University of Singapore
e-mail: `tslliuya@nus.edu.sg`
[4] School of Computing, National University of Singapore
e-mail: {`dongjs, zmanchun`}`@comp.nus.edu.sg`

The date of receipt and acceptance will be inserted by the editor

Abstract. Stateflow is an industrial tool for modeling and simulating control systems in model-based development. In this paper, we present our latest work on automatic verification of Stateflow using model checking techniques. We propose an approach to systematically translate Stateflow diagrams to a formal modeling language called CSP# by precisely following Stateflow's execution semantics which is described by examples. A translator is developed inside the PAT model checker to automate this process with the support of various Stateflow advanced modeling features. Formal analysis can be conducted on the transformed CSP# with PAT's simulation and model checking power. Using our approach, we can not only detect bugs in Stateflow diagrams, but also discover subtle semantics flaws in Stateflow user's guide and demo cases.

Key words: Model-Based Development – Transformation – Validation – Model Checking – Stateflow

1 Introduction

Stateflow, a product of the MathWorks Company, has been used widely to specify and simulate embedded control systems in various industry areas like aerospace [18] and transportation [1,6]. Stateflow's rich set of graphical modeling constructs allow users to quickly describe complex systems, and its simulation capability further helps users visualize system behavior under particular circumstances. On the other hand, the increasing criticality of embedded systems, for example, systems deployed in huge quantities in automobile manufacturers, raises the issue of providing high-level assurance at early development stages. Unfortunately, Stateflow fails to support this, due to the following two factors: (1) its semantics is specified in a narrative and sometimes partial manner over its 1358 pages long user's guide [17]; (2) simulations may be infeasible to analyze system behavior over a large and possibly infinite number of situations.

The above gap between the requirements and the current state of Stateflow creates an opportunity for formal methods, which are mathematically based analysis techniques for software engineering [11,26], since they provide unambiguous semantics (e.g., CSP [12] in process algebras) and rigorous verification capabilities (e.g., model checking [5]). In this paper, we develop formal analysis support to complement Stateflow based on a generic model checking system called Process Analysis Toolkit (PAT) [23,15]. PAT is a self-contained framework to support composing, simulating, and analyzing dynamic systems. PAT's specification language CSP# [22] offers great modeling flexibility by integrating high level operators from CSP with low level operators from common programs. Also PAT implements various model checking techniques catering for different properties such as reachability and linear temporal properties [14] . These strengths of PAT are useful for achieving our goal of formal modeling and reasoning about Stateflow diagrams.

We first identify the execution semantics of Stateflow, namely, how a Stateflow diagram executes its components and updates variables when it is simulated. Second, we construct a translator in C# as part of PAT for automatic translations. The inputs of the translator are MDL files which are textual representation of Stateflow diagrams in nested blocks of *keywords* and *parameter-value pairs*. However, there is no document available explaining the syntax or grammar of the format of MDL files. We hence acquire the details from experiments. Last, we specify important requirements based on translated CSP# models which represent the whole diagrams

⋆ The work was done when the author was at the School of Computing, National University of Singapore.

or some components, and they are verified fully automatically by applying PAT's model checking power.

So far we have covered a wide range of modeling features of Stateflow, including history junctions, inter-level transitions, implicit events, etc. Our approach has been applied to all semantic examples in the user's guide and non-trivial examples (including demo cases used by the MathWorks Company). We remark that we validate the correctness of the execution semantics we model by means of simulations, because the execution semantics of Stateflow is described in informal operational terms supported by examples. Thus, it is not possible to prove the equivalence between these two execution semantics. We carefully identify the execution semantics from the user's guide, and rigorously compare the execution order and variable values of our CSP# models (that are executable) and those counterparts of Stateflow diagrams step by step during simulations. This raises the confidence in our CSP# models, and also leads to discovery of subtle bugs and incomplete semantics in Stateflow user's guide [17]; these flaws have been acknowledged and fixed by the MathWorks Company in its latest user's guide (released in September, 2010).

The remainder of this paper is organized as follows. Related work is reviewed in the next subsection. Section 2 introduces the characteristics of Stateflow and PAT. Section 3 shows the translation from Stateflow diagrams into CSP# models. Experiments in Section 4 demonstrate the applicability and usefulness of our approach. Section 5 concludes the paper.

Related Work Although Stateflow appears to share many graphical notations with Statecharts [9], (e.g., arrow lines depicting transitions between states), its semantics is clearly different from Statecharts in its handling of non-determinism. The Stateflow semantics is complete deterministic, as the execution order among parallel states or outgoing transitions in a diagram is always sequential and fixed. Furthermore, Stateflow has its own modeling features; a *condition* action of a transition occurs before its source status becomes inactive, for instance. Therefore, existing work [10,13] on supporting Statecharts is inapplicable to Stateflow.

There exist several approaches to apply various formal verification tools to Stateflow. Banphawatthanarak et al. [2] translated Stateflow diagrams into the language of the SMV model checker. However, their work excluded multi-level hierarchical states and events, and only Boolean-valued variables were allowed. In 2001, Sims et al. [21] manually constructed Simulink/Stateflow models[1] in an invariant checker, although the construction was informally presented. Transformation to communication pushdown automata was reported by Tiwari [24], although the work did not consider the *sequential* execution order among parallel states in Stateflow. Scaife et al. [20] converted a subset of Stateflow constructs into Lustre, a synchronous programming language. Restrictions on the use of recursion in Lustre constrained the type of Stateflow diagrams supported; for instance, inter-level transitions were disallowed. Toyn and Galloway [25] proposed to model Stateflow diagrams using Z notation where they interpreted Stateflow as Statecharts. Recently, Cavalcanti [3] discussed the use of Circus [19] to specify Stateflow diagrams, though the way of handling event broadcasting in Stateflow was inappropriate: the evaluation was not done in a top-down manner. Denotational and operational semantics of Stateflow were proposed by Hamon [7] and Hamon and Rushby [8], respectively. Their definitions were at an abstract level and a number of Stateflow modeling features were left out, such as the way to capture the effect of actions was not specified and states having *during actions* were missing. Simulink Design Verifier [16] is a new product from the MathWorks Company to check Simulink/Stateflow models against assertions that are specified by logic expressions over variable values. The analysis is performed to all assignments of variables during a simulation period, and lack support of properties beyond assertions, for example, temporal properties specified in temporal logic.

Compared with those existing approaches, our work can provide a more comprehensive analysis of Stateflow diagrams. We follow its intrinsic execution semantics which is deterministic and sequential (among parallel states). A wide range of Stateflow modeling features is covered such as inter-level transitions and the *during* actions. We also carefully investigate complex behavior like event broadcasting in multi-level hierarchical diagrams. We also develop a translator automate the translation from Stateflow diagrams into PAT models.

In addition, this paper distinguishes itself from our preliminary work [4] by extensive refinement and enhancement. Previously, we discussed the possibility of modeling Stateflow diagrams in PAT, and a few basic Stateflow constructs were considered. In this paper, we extend the coverage of Stateflow by considering more complicated constructs such as implicit events, and refine previous method to capture the execution semantics more precisely; in particular, we add auxiliary variables for transitions to cope with event broadcasting in Stateflow. We also develop the translator and apply our approach to more and larger systems, for example, a fault-tolerant fuel system that is a demo case used by the MathWorks Company.

2 Background

In this section, we give a brief introduction to Stateflow and PAT, by covering their necessary features used in this paper. Complete description are available in their respective documents [17,22,15].

[1] Simulink is a commercial product specifying and simulating continuous dynamics, and usually integrates with Stateflow to model hybrid systems.

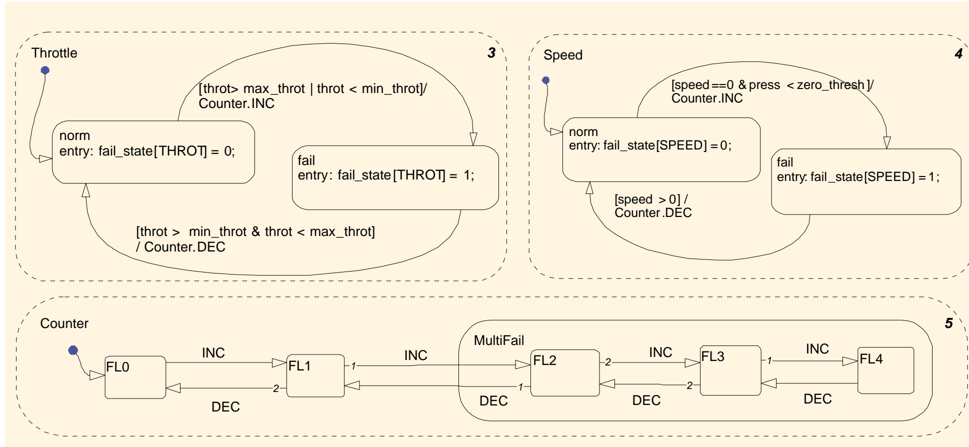


Fig. 1. Part of a Stateflow diagram of a fuel control system

2.1 Stateflow

A Stateflow diagram is basically formed by *states*, *transitions*, and *junctions* to specify dynamics of an event-driven system which changes its mode according to *events* and *conditions*. In the following, we will introduce important features of *states*, *transitions*, *events*, and *junctions*. *Actions* will be explained with states and transitions. We will also cover the behavior of a Stateflow diagram during simulation and the general structure of its textual representation in MDL files.

2.1.1 States

A state is either *active* or *inactive* during a simulation; it changes in response to events and conditions. The activity status of a state determines the state's behavior. Specifically, *entry actions* occur when a state becomes active, *during actions* take place when a state is active, and *exit actions* are executed when a state becomes inactive. We here use Figure 1, part of Stateflow diagrams modeling a fuel control system[2] which will be illustrated in Section 4, as a running example. For instance, when substate `norm` within state `Throttle` becomes active, an assignment `fail_state[THROT] = 0` occurs, where `fail_state` is an array and `THROT` is a constant.

To support hierarchical structure in Stateflow, a superstate can contain substates. During simulation, a superstate is activated (and executed) followed by its substates, and it is exited after all of its substates become inactive. For instance, state `Counter` contains a substate `MultiFail` that is a superstate of states `FL2`, `FL3` and `FL4` in Figure 1. A superstate is decomposed either 1) *exclusively* where its substates are called *OR* states and there is at most one *OR* state being active at a time, or 2) *in parallel* where its substates are called *AND* states and all *AND* states are active at the same time. For example, states `Counter`, `Speed`, and `Throttle` in Figure 1 are parallel state as indicated by the dashed border. Note that the numbers at the top right corners of these states show the activation and execution order among them; `Speed` becomes active after the activation of `Throttle` but before `Counter`, for instance. The order can be assigned manually by users or determined automatically by the Stateflow simulator.

2.1.2 Transitions

A transition is a directed edge that links one graphical object, either a state or a junction (discussed later), to another. A transition between two states represents a mode change from the source state to the destination state. When the source state is the destination state, it is a *self-loop* transition that causes the source state to become inactive and immediately thereafter become active. When the source state and the destination state are at different structure layers, namely, contained in different superstates (or diagrams), the transition is an inter-level transition; as shown in Figure 1 when the transition from state `FL1` to state `FL2` is executed, state `MultiFail` then becomes active implicitly. When the destination state is a substate of the source state, the transition is an *inner* transition. A default transition is a transition with no source state and it specifies which exclusive state to enter initially, for example, state `FL0` is entered when state `Counter` becomes active.

A transition is characterized by its label which consists of events, conditions, condition actions, and transition actions in the following format.

$$events[conditions]\{cond_actions\}/trans_actions$$

A transition with a label is enabled when its source state is active, events (the absence of them means any event) occur, and conditions (the absence of them means value *true*) are true. When a transition is enabled and before it is taken, condition actions take place. In contrast, transition actions occur only after the source state of a

[2] Also available at `http://www.mathworks.com/products/stateflow/demos.html?file=/products/demos/stateflow/sldemo_fuelsys/sldemo_fuelsys.html`

valid transition becomes inactive, i.e., the transition is taken. For instance, transition from `FL1` to `FL2` can occur when `FL1` is active and event `INC` occurs. Note that there are two outgoing transitions from `FL1` and both are ornamented with numbers indicating the execution order. Similar to the execution order of states, Stateflow can determine the order among multiple transitions from the same state so as to avoid nondeterminism.

2.1.3 Events

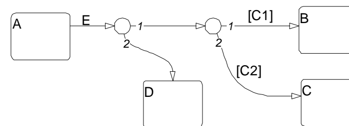
An event is used to trigger actions of a transition or a Stateflow diagram. An explicit event is defined by users, and it can be an input from Simulink, an output to Simulink, or local within a diagram. An implicit event is a built-in event that broadcasts automatically during diagram execution. Three commonly used implicit events are *tick*, *enter(state_name)*, and *exit(state_name)*: *tick* indicates the moment when a Stateflow diagram awakens, and the other two occur when the specified state of *state_name* is entered or exited, respectively.

Event broadcasting is a common communication technique in Stateflow. When an event is globally broadcast, the evaluation of the event starts from a Stateflow diagram that is the root of all its components and follows the hierarchy of states in a top-down manner. An event can also be *directly* broadcast from one state to another to synchronize parallel states, and the evaluation of the event is within the destination state. Direct event broadcasting is specified in a qualified format *state_name.event_name*. For example, transition from state `fail` to state `norm` of state `Speed` in Figure 1 broadcasts directly event `DEC` to its parallel state `Counter`.

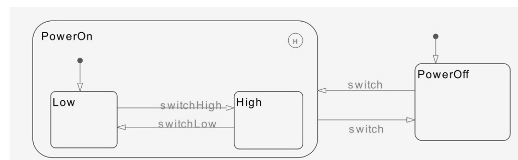
2.1.4 Junctions

There are two types of junctions in Stateflow. First, connective junctions enable representations of different possible transition paths for a single transition. They are often used to model certain types of constructs in a flow graph, such as an *if-then-else* decision and a *for* loop. For instance, in Figure 2(a), the transition has three possible paths to different states by a nested if-then-else construct. To be specific, if condition `C1` is true, state `B` is entered; else if condition `C2` is true, state `C` is entered; when both `C1` and `C2` are false, a *backtracking* occurs and hence state `D` is entered.

Second, history junctions record historical activity information of states. A superstate containing a history junction chooses to activate the substate which was active last time when the superstate was exited. Taking state `PowerOn` in Figure 2(b) as an example, it possesses a history junction (located at the top right corner), and hence after the initial phase, when `PowerOn` becomes active, it will activate substates `Low` or `High` based on the historical state activation information rather the default



(a) Two connective junctions



(b) A history junction

Fig. 2. Examples of junctions in Stateflow

transition. In other words, after the initial phase the activation of substates is determined by history junctions not default transitions.

2.1.5 Diagram Execution

A Stateflow diagram is executed in discretely steps during simulation. At each sample (discrete) time step, execution proceeds top-down through the diagram hierarchy. Initially, default activities are performed; for example, a substate attached to a default transition becomes active. After the initial phase, the diagram updates its components in response to events and conditions. Actions that take place based on an event are *atomic* to that event, and hence these actions must finish before evaluating another event; for instance, state `norm` of state `Speed` in Figure 1 can be re-entered only after the evaluation of the direct event broadcast `Counter.DEC` to state `Counter` completes.

2.1.6 MDL Structure

Stateflow diagrams are saved as text files with MDL extension. The MDL files are organized in a *nested block* structure with *parameter-value pairs*. Each Stateflow object such as *states* is specified by a block that comprises a pair of brackets following a keyword. The contents within each block depict the properties of the corresponding object by a sequence of pairs where each pair consists of a Stateflow parameter and particular value. As mentioned in the introduction, there is no document available for the syntax of MDL files. We have learned the structure of MDL files by careful inspection of a number of case studies. Figure 3 presents part of the MDL file that represents the fuel control system in Figure 1, specifically, states `Speed` and `fail` and the transition from `fail` to state `norm` in `Speed`.

As shown in Figure 3, each object is assigned to a unique integer as its identification (`ID`) in MDL files.

```
state {    id          11
  labelString          "Speed"
  treeNode             [2 14 10 16]
  type             AND_STATE
  executionOrder       4      }
state {    id          15
  labelString          "fail\nentry: fail_state[SPEED] = 1;"
  treeNode             [11 0 14 0]
  type             OR_STATE  }
transition { id        49
  labelString          "[speed > 0] /\nCounter.DEC"
  src {      id            15    }
  dst {      id            14    }
  linkNode             [11 47 0]    }
```

Fig. 3. Contents of a MDL file for the fuel control system

User-defined properties such as state names, actions, and transition conditions are stored as *strings* of parameter `labelString`. The hierarchical structure of states is preserved by parameter `treeNode` where the first element indicates the superstate and the second one the first substate. For example, `treeNode [11 0 14 0]` within the block of `fail` whose ID value is 15 indicates that `Speed` (ID value 11) is its superstate; value 0 means no Stateflow object, i.e., `fail` has no substate. In addition, execution order of a parallel state is explicitly stated; for instance, `executionOrder 4` of `Speed` which is a parallel state, denoted by `type AND_STATE`. Important transition properties like source state (by `src`), destination state (`dst`), and scope (`linkNode [11 47 0]` specifies that the transition is within state `Speed`) are also captured in the transition block specification.

The MDL files are the input of our translator as these files capture all characteristics denoted by Stateflow diagrams. The complicated behaviors of Stateflow diagrams require our analyzing system to be very expressive and have powerful reasoning capabilities; the process analysis toolkit which is introduced in the next subsection satisfies these requirements.

2.2 Process Analysis Toolkit (PAT)

PAT is an extensible model checking framework for system modeling, simulation and verification. PAT consists of an editor, a simulator and various verifiers. The editor provides an environment to develop system models and includes syntax checking. The simulator enables users to interactively execute and observe system behavior using facilities such as random simulation, user-guided step-by-step simulation, trace playback, counterexample visualization, etc. The verifiers apply state-of-the-art model checking techniques (e.g., depth-first-algorithm for safety properties and strongly connected components based algorithm for liveness properties) to analyze systems. Using PAT, we can rigorously and automatically verify important properties including deadlock-freeness and linear temporal logic properties.

PAT adopts a layered design to support the analysis of different models such as concurrent and probabilistic models. For each supported model, a dedicated module is created which defines the (specialized) language syntax, well-formedness rules and formal (operational) semantics. In our work, we use the CSP module due to its support of a rich modeling language name CSP#. CSP# combines high-level operators like nondeterministic choice from the classic process algebra CSP (short for Communicating Sequential Programs) [12], with programmer-favored low-level constructs like variables and if-statement. In addition, built-in types in PAT cover integers, Boolean, and array of integers. User-defined data types can be constructed by creating a C# class which inherits the C# *Value* interface and importing that class to CSP# models.

A constant is declared with keyword `#define`. For example, `#define on 1` defines a global constant `on` with value 1. Note that constant values can only be integer and Boolean in CSP#.

A variable is indicated with keyworkd `var` and is either a scalar or an array. Range and initial value(s) can be specified explicitly. For instance, `var status: = on` assigns constant value `on` as the initial value of variable `status`, and `var fail_state[4]:{0..1}` defines an array `fail_state` with 4 elements and the range (0 or 1) of each element.

A CSP# process can be constructed as follows.

- The process `Skip` terminates and does nothing.
- *Data operation prefixing*: extends the conventional event prefixing processes in CSP by attaching a statement block of a sequential program to an event. The sequential program that usually updates variables is executed atomically with the occurrence of the event. When no event is given, it is interpreted as the invisible event τ. For instance, `{x = x + 1;} -> Skip` increases variable `x` and then terminates.
- *Sequential composition*: processes `P` and `Q` composed by operator `;` perform in a sequential manner: `Q` starts only when `P` has finished.
- *General choice*: in process `P [] Q`, either process `P` or process `Q` may execute.
- *Conditional choice*: classic *if-then-else* construct is supported in PAT by the form `if(c){P} else{Q}`, where `c` is a Boolean formula. If `c` is true, process `P` executes, otherwise process `Q` executes. Note that the else-part is optional.
- *Atomic sequence*: keyword `atomic` defines a process which always makes maximum progress. If a sequence of statements is enclosed in parentheses with `atomic` as its prefix, the sequence executes in one super-step without interference from other processes.

An assertion is a query about system behaviors or properties, indicated by keyword `#assert`.

- *Deadlock*: `#assert P deadlockfree` checks if process `P` does not deadlock.
- *Reachability*: `#assert P reaches goal` tests whether process `P` can reach a state at which a given Boolean-valued condition `goal` holds.

– *Linear Temporal Logic* (LTL): PAT supports the *full* set of LTL syntax, such as □ (*always*) and ◇ (*eventually*) temporal modal operators. In general, `#assert P |= F` examines if `P` satisfies an LTL formula `F`.

The above expressive power of CSP# and the automatic reasoning ability of PAT allow us to model the execution semantics of Stateflow and verify dynamic systems represented as Stateflow diagrams against important properties.

3 Translating Stateflow Diagrams to CSP#

Our translation takes MDL files that are textual representations of Stateflow diagrams; states, transitions and junctions are denoted by block entities that contain user-specified parameter values. The translation first converts states (Section 3.1) followed by transitions (Section 3.2) into PAT models. During the translation of states and transitions, we invoke processes of transforming actions (Section 3.3) and (implicit) events when needed. After translating all states and transitions, we construct the PAT models for the diagrams (Section 3.4.1). Our translation also covers advanced Stateflow modeling features like junctions (Section 3.4.2). At the end of this section, we present our discoveries and discussion.

3.1 States

We translate each state into four CSP# processes, where three of them model types of behavior, and the fourth process captures transition between those three types of behaviors. A state is represented by a block of contents headed by keyword `state`, and essential parameters that capture its structure and functionality are listed below, where *N* and *string* denote a natural number and a string value, respectively.

```
state{
   id               N
   labelString      " string "
   treeNode         [ N, N, N, N ]
   type             AND_STATE or OR_STATE or FUNC_STATE
   decomposition    SET_STATE or CLUSTER_STATE
   executionOrder   N
}
```

The value of the `labelString` parameter stores user-defined information like state name, and actions in response to different state status; for example, actions following string *exit* are executed when the state exits its active status. We demonstrate below how these parameters with their values guide our translation.

3.1.1 Modeling Entry Behavior

The name of the process modeling a state entry behavior is suffixed by `_EnAct()`. In addition, we use the process

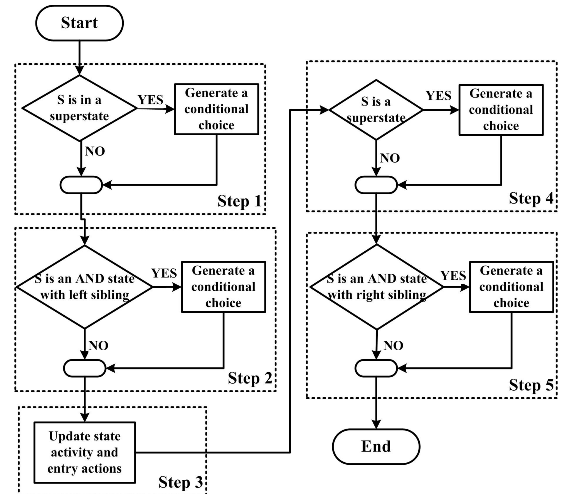


Fig. 4. Workflow for modeling entry behavior of a state *S*

name to indicate the hierarchy structure of the state, by appending the names of all states (using symbol "_") along the diagram's hierarchy. For example, a process name `A_B` indicates that a state named *B* is contained in state *A*. This hierarchy information can be derived from the `treeNode` parameter.

The procedure of constructing a process body to model the entry behavior of a state is sketched in Figure 4; its five constituent steps are demonstrated below.

1. If the state is contained in a superstate (i.e., the *first* element of parameter `treeNode` is greater than zero), a CSP# conditional choice is generated followed by a CSP# sequential composition operator. Otherwise, the construction procedure moves to the next step. In the generated conditional choice, the condition returns true when the superstate is inactive, and the *then*-branch invokes the process name which represents the superstate so as to activate the superstate.
2. If the state is a parallel state (i.e., the value of parameter `type` is `AND_STATE`) with a left sibling state (i.e., the *third* element of parameter `treeNode` is greater than zero), a CSP# conditional choice is generated; otherwise the construction procedure continues. In the generated conditional choice, the condition returns true when the left sibling state is inactive. In other words, we need to activate the parallel state (in this case, the left sibling state) which has higher execution priority. The *then*-branch invokes the process name which represents the left sibling state, and the *else*-branch contains all CSP# specifications that will be produced in the following steps.
3. The state activity status is updated and entry actions are executed (the way to handle actions will be elaborated in Section 3.3). In our CSP# model, each status is indicated by a variable whose name comprises the process name of the corresponding state and the suffix "`_Status`". The value of such a vari-

able is either `active` (defined as a CSP# constant with value 1) or `inactive` (a constant with value 0).

4. If the state is a superstate (i.e., the *second* element of parameter `treeNode` is greater than zero), a CSP# conditional choice without the *else*-branch is created; otherwise, the procedure moves to Step 5.
   In the generated conditional choice, the condition returns true if this superstate is *not* activated by any of its substates. During our translation, an auxiliary variable is defined for every superstate to indicate the activation direction. The name of such a variable appends the suffix "`_EnterBU`" to the process name of the superstate, and the value is Boolean; initially, the value is false denoting that no substate activates that superstate, and the value can be changed when a substate executes Step 1.
   The *then*-branch contains a process, which specifies the entry behavior of the default substate(s), followed by any CSP# specifications generated in Step 5. Based on the decomposition type of this state, There are two ways to identify the default substate(s).
   – When the substates are *AND* states, (i.e., the value of parameter `decomposition` is `SET_STATE`), the substate whose identification (`id`) is equal to the *second* element of parameter `treeNode` has the highest execution priority. Therefore the process name representing the substate as the default state is invoked to perform entry behavior. We remark that the property where the second element of `treeNode` points to a substate with the highest execution priority is learnt from our intensive experiments although this property is unavailable from Stateflow's documents (the property has also been confirmed by the MathWork).
   – When the substates are *OR* states, (i.e., the value of parameter `decomposition` is `CLUSTER_STATE`), we identify default substate(s) from default transition(s) that will be described in Section 3.2. Once default substate(s) are known, we can invoke corresponding process name(s) to model the entry behavior. Note that multiple default substates are possible (guarded by different conditions). Moreover, the *second* element of parameter `treeNode` in this case usually does not point to a default substate; this is different from the case when substates are parallel.
5. If the state is a parallel state (i.e., the value of parameter `type` is `AND_STATE`) with a right sibling state (i.e., the *fourth* element of parameter `treeNode` is greater than zero), a CSP# conditional choice without the *else*-branch is generated and the whole construction procedure completes. Otherwise, if the state is not a superstate (checked in the previous step), a `Skip` process is created and the construction procedure terminates; else the procedure terminates. In the generated conditional choice, the condition returns true if the right sibling state is inactive, and the *then*-branch invokes the process name which represents the right sibling state.

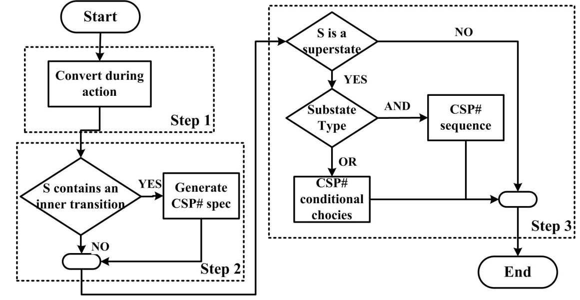


Fig. 5. Workflow for modeling during behavior of a state S

Taking state `Speed` in Figure 1 as an example, it is a parallel state and contains two exclusive substates. Process `Speed_EnAct()` is generated to specify the entry behavior: the *then*-branch at line 2 (created at Step 2) activates state `Throttle` which has higher execution priority than `Speed` if `Throttle` is inactive; after changing `Speed`'s status at line 3 (Step 3), the default substate `norm` may be activated if `Speed` is not activated by any of its substates (indicated by variable `Speed_EnterBU`) at line 4 (Step 4); the *then*-branch at line 5 will activate state `Counter` which has lower execution priority than `Speed` if `Counter` is inactive (Step 5).

```
1 Speed_EnAct()=
2   if(Throttle_Status == inactive){ Throttle() }
3   else{ {Speed_Status = active;} ->
4     if(Speed_EnterBU  ==  false){ Speed_norm();
5       if(Counter_Status ==  inactive){ Counter()}}};
```

The above CSP# specification complies with the informative description about entry behavior in Stateflow user's guide. First, when a superstate is activated by one of its substate, the superstate does not trigger its substates and not check its right sibling state. Second, all parallel states are activated at the same time according to their execution priority. In addition, our construction prevents repeated execution of entry actions of parallel states; details are elaborated in Section 3.5.

3.1.2 Modeling During Behavior

We construct a process whose name is append suffix "`_DurAct()` to the process name denoting the state. Figure 5 shows the conceptual workflow of a generation procedure of the process body which depicts the *during* behavior of a state. Three steps are explained below.

1. Transform user-defined *during* actions into CSP# specifications (Section 3.3 elaborates how to handle actions in details).
2. Generate CSP# specifications to represent an *inner* transition (its definition is given in Section 3.2) if there exists any.

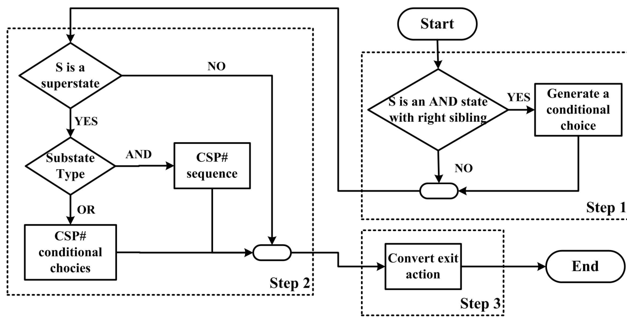


Fig. 6. Workflow for modeling exit behavior of a state S

3. When the state is a superstate (i.e., the *second* element of parameter `treeNode` has positive value), there are two cases based on its decomposition type.
   – If the decomposition is parallel, all substates are executed in parallel. However, their execution is still in a *sequential* order which is determined by their execution priority; higher priority, executed earlier. We declare a CSP# *sequence* composing process names of all AND substates and the sequence order is the same to their execution order.
   – If the decomposition is exclusive, there is at most one active substate. We declare nested CSP# conditional choices to identify any active substate. Specifically, in a conditional choice, the condition returns true when a substate is active and its *then*-branch invokes the process name denoting the substate; otherwise, the *else*-branch contains a conditional choice for another substate. Note that the order of checking is not important unlike the case when the decomposition is parallel.

We reuse state `Speed` in Figure 1 here to further show the resulting CSP# process specifying its during behavior. In process `Speed_DurAct()`, the during action of `Speed` is null (denoted by `Skip`), and the conditional choices at lines 2 to 4 check the activity status of substates `norm` and `fail`.

```
1 Speed_DurAct()= Skip;
2   if(Speed_norm_Status == active){ Speed_norm() }
3   else{ if(Speed_fail_Status == active){
4              Speed_fail()}};
```

3.1.3 Modeling Exit Behavior

A CSP# process is generated to model the exit behavior of a state, where the process name is suffixed with "`_ExAct()`". The generation procedure displayed in Figure 6 consists of the following steps.

1. If the state is a parallel state (i.e., the value of parameter `type` is `AND_STATE`) *and* has a right sibling state (i.e., the *fourth* element of parameter `treeNode` is greater than zero), a conditional choice without the *else*-branch is created. Otherwise, the procedure moves to the next step.
   The created conditional choice captures the behavior that any active parallel state with lower execution priority should be deactivated before the current parallel state. Specially, the condition returns true if the right sibling state is active, and the *then*-branch references a process name which models the exit behavior of the right sibling state.
2. If the state is a superstate (i.e., the *second* element of parameter `treeNode` has positive value), any active substates should be deactivated before exiting the state; otherwise, the procedure moves to Step 3. There are two cases based on its decomposition type.
   – If the decomposition is parallel, all active AND substates should become inactive in a *sequential* order which is opposite to their execution priorities; namely, a substate having lower priority is deactivated earlier. Therefore, we declare a *sequence* of conditional choices without *else*-branches, where the sequence order is reverse to the substates' execution priorities and each conditional choice deactivates a substate if it is active.
   – If the decomposition is exclusive, all active OR substates should become inactive in *any* order. Thus, we declare *nested* conditional choices. Each conditional choice checks the activity status of one substate, and invokes the process name which models the exit behavior of the substate in the *then*-branch; the *else*-branch contains another conditional choice which checks another substate.
3. Convert user-defined *exit* actions into CSP# specifications (details are in Section 3.3) followed by changing the activity status of the state to *inactive*.

Below we use state `Speed` in Figure 1 to demonstrate our way of capturing exit behavior. `Speed` is a parallel state where state `Counter` is its right sibling state, and `Speed` also contains two substates `norm` and `fail`. In our translated process `Speed_ExAct()`, the conditional choice at line 2 checks the activity status of `Counter`, the nested conditional choices between lines 3 and 5 deactivate any active substates, and the update of `Speed`'s activity status is captured at line 6.

```
1 Speed_ExAct()=
2   if(Counter_Status == active){ Counter_ExAct()};
3   if(Speed_norm_Status == active){ Speed_norm_ExAct()}
4   else{ if(Speed_fail_Status == active){
5              Speed_fail_ExAct()}};
6   {Speed_Status = inactive;} -> Skip;
```

3.1.4 Modeling Overall Behavior

The previous three subsections define processes to model the *entry*, *during*, and *exit* behavior of a state. In this subsection, we construct a process to model the overall behavior of a state, to be specific, the transitions from entry, through during, to exit behaviors. The process name is formed by appending the state name to other

state names along the hierarchy in a particular Stateflow diagram. The contents of the process is a conditional choice, where the condition returns true when the state is *inactive*, and the *then*-branch invokes the process name (generated in Section 3.1.1) which represents the entry behavior. The *else*-branch copes with two cases according to the existence of an outgoing transition (its detection will be explained in Section 3.2).

Case 1: When a state has no outgoing transition, the state will stay active, once activated. Thus, the *else*-branch contains the process name which represents the *during* behavior.

Case 2: When a state has an outgoing transition, the state will become inactive if the transition occurs; an auxiliary Boolean variable suffixed with "`_OUTGOING`" is defined to indicate the occurrence. Furthermore, when there are more than one transition, only one transition can take place, and checking of their occurrences follows their individual execution priority (indicated by the `executionOrder` parameter in a transition block textual representation); in other words, a transition with higher execution priority is checked earlier. This behavior is modeled by a *sequence* of conditional choices, where the sequence order is the same as the transition execution priority. Each conditional choice corresponds to an outgoing transition: it contains a predicate that requires the auxiliary variable to be false, and updates the variable to be true in its *then*-branch that specifies the behavior when a transition occurs. Moreover, except the last conditional choice, the others are created without their *else*-branches; the *else*-branch of the last one invokes the process name which represents the *during* behavior of the state. This modeling avoids that transitions with lower execution priorities to be retaken when a transition with higher priority is possible.

In addition, as a transition can cross multiple layers between states, i.e., an inter-level transition, it is crucial to invoke the correct process names which represent the *exit* behavior of a source state and the *entry* behavior of a destination state in the *then*-branch of an above generated conditional choice. Algorithm 1 is designed to derive the outermost source state of an outgoing transition: it starts checking if the parent object of the source state ($Parentof(SrcID)$) is equal to the container of a transition, and updates the parent object iteratively until the equivalence is valid.

Algorithm 1 Find the outermost source state

Require: $SrcID > 0$

1: $ContainerID \Leftarrow$ *first element of linkNode*
2: **while** Parentof($SrcID$) $\neq$ $ContainerID$ **do**
3: $\quad SrcID \Leftarrow$ Parentof($SrcID$)
4: **end while**
5: **return** $SrcID$

The above algorithm returns a state whose parent is the container of the transition, and thus we reference the process which specifies the *exit* behavior of that state. In contrast, it is unnecessary to derive an outermost destination state, because our approach for modeling *entry* behavior of a state requires a state to activate its superstate when needed (Section 3.1.1). Namely, we can reference the process which depicts the *entry* behavior of the destination state of the transition.

We below exemplify state `FL2` in Figure 1 which has two outgoing transitions, one of them an inter-level transition. In the following simplified CSP# model of `FL2` (process names are in an abstract form), the outermost *then*-branch (at line 1) triggers the *entry* behavior when process `FL2()` is invoked and `FL2` is inactive. The two sequentially composed conditional choices (lines 3-7) captures the execution priority between the two outgoing transitions. In addition, the outermost source states of these transitions are different; the source state in the first conditional choice is `MultiFail` which is a superstate of `FL2`.

```
1 FL2()= if(FL2_Status == inactive){ FL2_EnAct()}
2   else{
3     if(... && (FL2_OUTGOING == 0)){
4       MultiFail_ExAct();{FL2_OUTGOING = 1;}...FL1()};
5     if(... && (FL2_OUTGOING == 0)){
6       FL2_ExAct();{FL2_OUTGOING = 1;}... FL3()}
7     else{ if(FL2_OUTGOING == 0){FL2_DurAct()}};
8     {FL2_OUTGOING = 0;}};
```

In this section, we covered the transformation of a state by constructing four processes to respectively model three types of behaviors and the changes among them. The translation retains the complex and dynamic behavior of states; for example, the entry behavior in our approach takes into account if a state is a substate, a superstate, or a parallel state.

3.2 Transitions

In a Stateflow diagram, a transition represents a change of system mode and associated actions. We start with the general structure of a transition in the textual format, followed by details of different types of transitions. Last but not least, an algorithm converting transitions into CSP# models is presented.

A transition is described by a block starting with the keyword `transition` in a MDL file. As shown below, the block includes information such as the unique id of a transition, ids of the source object (denoted by block `src`) and the destination object (by block `dst`), and the id of the container, the object containing the transition (by the *first* element of parameter `linkNode`). In addition, user-defined specifications such as guarded events and actions are stored in a string attached to parameter `labelString`. When there are multiple transitions from the same source object, parameter *executionOrder* indicates the execution priority of a transition.

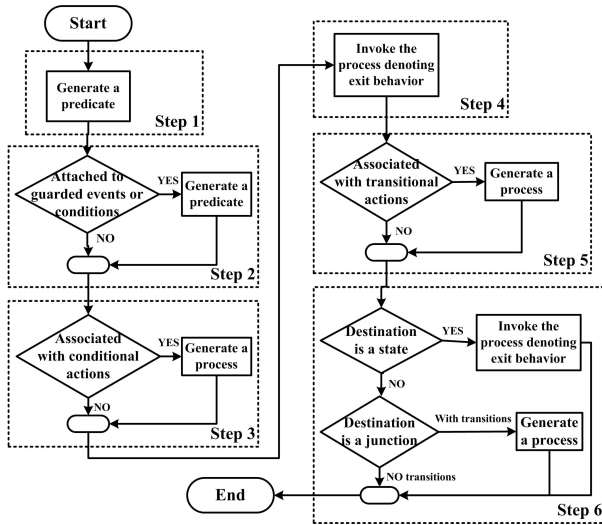


Fig. 7. Workflow for modeling a transition

```
transition{ id              N
            labelString     " string "
            src { id        N }
            dst { id        N }
            linkNode        [ N,  N,  N]
            executionOrder  N }
```

Transitions can be classified into four types according to their parameter values.

– Default transitions: the `src` block has no parameter-value pair. In other words, the corresponding transition has no source object, and the destination object will execute initially when the container becomes active. For instance, if a default transition points to a state, the state is a default state of its superstate (used in Section 3.1.1).
– Inner transitions: the source object is the container (in Section 3.1.2). Namely, their IDs are equal.
– Regular transitions: the container is the parent object of the source object, and is also the parent object of the destination object.
– Inter-level transitions: the container is neither the source object nor the parent object of the source object, or the container is not the parent object of the destination object (used in Section 3.1.4).

The algorithm that converts a non-default transition consists of six steps as shown in Figure 7.

1. A conditional choice without an *else*-branch is generated. The condition contains a predicate that returns true when the source state is still active (as mentioned in Section 3.1.4 using auxiliary variables), and other predicates can be created in the next step. The *then*-branch includes a sequence of CSP# specifications that are produced in Steps 3 to 6.
2. If the transition has guarded events or conditions, they are translated into predicates which return true when events occur and conditions are true. Note that an event is represented by a Boolean-valued variable in our CSP# models. This allows us to capture the *deterministic* behavior for multiple transitions from the same source object. Reusing state `FL2` in Figure 1 as an example, the sequential execution order between its outgoing transitions is difficult to capture by a predefined event in CSP# which cannot specify the occurrence order of events, although such behavior is supported by the encoding we use (the CSP# model is shown in Section 3.1.4).
3. If the transition is associated with conditional actions, those actions are converted into CSP# model actions (details are in Section 3.3).
4. Invoke the process name which represents the *exit* behavior of the source state; the way to deal with case when the transition crosses different layers (an inter-level transition) has been covered in Section 3.1.4. Nevertheless, for an inner transition, the source state remains active.
5. If a transition action is specified by users, it is converted into a CSP# process (details are in Section 3.3).
6. If the destination object is a state, invoke the process which represents the *entry* behavior of the state. Otherwise, if the destination object is a connective junction without outgoing transitions, no translation is needed; and the situation that the object is a connective junction with outgoing transitions is covered later in Section 3.4.2.

The above procedure can be easily adapted to convert default transitions by skipping Steps 1 and 4. Moreover, this procedure can serve as a foundation to cope with transitions consisting of in-between connective junctions (an advanced features of Stateflow) in Section 3.4.2.

3.3 Actions

In Stateflow, actions are attached to either states (e.g., entry actions) or transitions (e.g., conditional actions). We consider two common types of actions. One is assignments, and the other is event broadcasting.

Assignments manipulate variable values, and are expressed in conventional mathematical format. These assignments are naturally mapped to their CSP# counterparts in a data operation prefixing process. Currently *all* unary and binary mathematical operators over *integers* in Stateflow are supported in CSP#. We remark that CSP# lacks direct support for floating points and structured datatypes. Nonetheless, PAT supports user-defined data structures written in programming languages such as C#, C, and Java, and hence we can construct C# (C or Java) floating point fields (or structured datatypes) and use them in CSP# model.

There are two kinds of event broadcasting, *global* and *directed*. Directed event broadcasting is an efficient means of synchronization among parallel states. When an event is broadcast to a specific state, that state receives the event and evaluates its impact. A directed

event broadcasting is translated into a *sequence* of CSP# processes within the process which denotes the source state of a transition attached to the action. The sequence first enables the event, followed by invoking the process name which represents the target state, and lastly disables the broadcast event.

For example, the transition in Figure 1 from substate `norm` to substate `fail` in state `Speed` contains a transition action that directly broadcasts an event `INC` to a parallel state `Counter`. This transition is modeled by the *else*-branch of the following process `Speed_norm`: the data operation at line 3 updates the event (denoted by `INC`), and the invocation of process `Counter` at line 4 evaluates the effect of this action; the event becomes disabled at line 5. In the process, `occurred` and `notoccurred` are declared as globally CSP# constants.

```
1 Speed_norm()=if(Speed_norm_Status == inactive){...}
2   else{if(((speed == 0)&&(press < zero_thresh))&&...){
3     ... {INC = occurred;}->
4     Counter();
5     {INC = notoccurred;}...}...};
```

When an event is broadcast globally, it impacts the whole Stateflow diagram. That is to say, the evaluation starts from the diagram level and moves down to states according to their execution order. Moreover, an auxiliary global variable named `broadcast` is defined in our CSP# model to count the occurrence number of global event broadcasting. Similarly to the way of dealing with directed event broadcasting, a *sequence* of CSP# processes is created within the source state of the transition which triggers this action for modeling global event broadcasting. The sequence first increases `broadcast` by value 1, and then invokes the process named `Chart` which denotes the whole diagram, followed by decreasing `broadcast` by value 1. More details about `Chart` and the usage of `broadcast` are in the next section. Note that we here assume that the number of occurred event broadcasting is *finite*; nonetheless, we can set a large threshold to avoid potentially infinite event broadcasting.

3.4 Advanced Features

The previous three subsections have illustrated our translation for states, transitions and actions which are essential constituents of Stateflow diagrams. This subsection describes our translation for advanced modeling features: diagrams, junctions, and implicit events.

3.4.1 Modeling Diagrams Behavior

As initially mentioned, we focus on the execution semantics of Stateflow diagrams, namely, the behavior at each sample time step during simulation. We generate a CSP# process named `Chart` to represent a Stateflow diagram. The body of `Chart` is a conditional choice, where the *then*-branch specifies the initialization of the diagram, and the *else*-branch models the *during* behavior.

Similar to states, a diagram is decomposed by either exclusive states (denoted by value `CLUSTER_CHART` of parameter `decomposition`) or parallel states (by `SET_CHART` of `decomposition`). The decomposition type determines how a chart activates and executes its states. For example, the *during* behavior of a chart with exclusive states is represented by nested CSP# conditional choices to execute any active states. Note that this is the same way as states are handled (mentioned in Section 3.1.2).

A CSP# event `click` is defined to imitate a sample time step in `Chart`. In addition, to capture the instantaneous behavior of actions at a sample time step, we use the CSP# atomic construct to invoke processes denoting states. Thus, when an event is broadcast globally, the effect consumes zero time; and no `click` occurs.

The above `Chart` process models the behavior of a Stateflow diagram at one sample time step. When the diagram interacts with environment, it requires input data and events. We create a process named `Initialization` to simulate the valuation of environmental variables, i.e., finite data and events at each sample time step. This process captures all possible assignments by a *sequence* of processes, where each constituent process comprises of a set of CSP# *general choices* to describe a possible assignment of an environment input. For instance, process `([]x: {l..h} @{ed = x;} -> Skip` randomly assigns an environment input `ed` a value which is from a lower bound `l` to a higher bound `h`.

Based on processes `Chart` and `Initialization`, a process named `Stateflow` is constructed to represent the *periodic* behavior of a Stateflow diagram. When there is an environmental variable, the contents of `Stateflow` is `Chart(); Initialization(); Stateflow();` that invoke itself recursively. Otherwise, we have `Stateflow() = Chart(); Stateflow();`.

3.4.2 Modeling Junctions

Junctions in Stateflow are divided into two groups: connective junctions (by value `CONNECTIVE_JUNCTION` of parameter `type` in a textual format shown below) and history junctions (by value `HISTORY_JUNCTION` of parameter `type`). They serve different purposes, and thus the ways of translating them varies; illustrative examples of their use and our translation will appear in Section 4.

```
junction{  id              N
  linkNode         [ N,  N,  N]
  type  CONNECTIVE_JUNCTION  or HISTORY_JUNCTION }
```

Connective junctions within a single transition denote different transition paths. During simulation, the evaluation of such a transition ends at either a state or an ending connective junction (that has no outgoing transitions). These connective junctions are used to guide our translation algorithm to generate nested C-

SP# conditional choices to capture *all* transition paths. To be specific, the condition at line 3 checks the type of the destination object ($tr.dst$), and the condition at line 5 detects that the destination object contains at least an outgoing transition ($OutTS$). The *for* loop (lines 6 to 10) selects every outgoing transition ($temptr$) from the connective junction, updates $trActs$ by appending the transition action associated with the transition which ends at the junction, and invokes recursively the method $translateT$ with the latest transition and transition actions. When the destination object is an ending junction (at line 11) or a state (at line 15), the operation of $GenerateT$ generates processes that respectively model the entry behavior of the source object, transition actions (if any), and the exit behavior of the destination object (same as Steps 4 to 6 explained in Section 3.2).

Algorithm 2 Cover all transition paths

Require: src: the source object of the transition
tr: a transition; $trActs$: a list of transition actions.
1: $translateT(src, tr, trActs)\{$
2: $handleCondAct(tr.condActs)$
3: **if** $tr.dst$ is instance of Junction **then**
4: $OutTS \Leftarrow tr.dst.OutTS$
5: **if** $\#OutTS > 0$ **then**
6: **for** $i = 0$ to $OutTS.size()$ **do**
7: $temptr \Leftarrow OutTS(i)$
8: $temptrActs \Leftarrow append(trActs, tr.trActs)$
9: $translateT(src, temptr, temptrActs)$
10: **end for**
11: **else**
12: $GenerateT(src, tr, trActs)$
13: **end if**
14: **else**
15: $GenerateT(src, tr, trActs)$
16: **end if**
17: $\}$

In the above algorithm, conditional actions are handled at line 2 by function $handleCondAct$ (similar to Step 3 in Section 3.2). The reason for separating conditional actions from transition actions is that the former can be executed once the corresponding condition is true. We note that line 6 sorts the list of outgoing transitions according to their execution priorities as specified by parameter `executionOrder`; so that an outgoing transition with higher execution priority is translated before another with lower priority.

Another group of junctions are history junctions. When a state, called container (whose ID is the *first* element of parameter `linkNode`), containing a history junction, becomes active again, it activates a substate (usually is exclusive) which was exited last time; the container activates its default substate at the beginning of a simulation, i.e., time zero. In other words, the history junction records the latest active substate. We capture this modeling feature by the following three steps.

1. Declare a global CSP# variable to represent a history junction. The variable name is the process name denoting the state containing the history junction and the suffix "`_H`".
2. Update the history variable value in the process that represents the *exit* behavior of the container. The update takes place after the invocation of the process which denotes the *exit* behavior of a substate in the container (Step 2 in Section 3.1.3), and assigns the substate ID to the history variable.
3. Construct conditional choices to control the activation of a substate of the container in the process representing the container's *entry* behavior. When the history variable is equal to the default value 0, the container activates default substate(s) as demonstrated in Step 4 of Section 3.1.1. Otherwise, the substate to be activated is determined by the value of the history variable.

3.4.3 Modeling Implicit Events

Implicit events are built-in events in Stateflow, which are not described by event objects in MDL files. Instead, they are specified as strings associated with other Stateflow objects, usually as guarded events in transitions. In our translation, an implicit event is identified by its specific representation format and the location where it appears, and the event is captured by a CSP# specification specifying its effect in a particular diagram.

For example, an outgoing transition, with "`enter(A)`" as the value of its parameter `labelString`, is guarded by an implicit event that occurs when the specified state (`A`) is activated. This behavior is captured by CSP# specifications generated in the following steps. First, a Boolean-valued variable is defined to denote the implicit event; for the above example, `Enter_A` is declared in `var Enter_A:{0..1} = notoccurred`. Next, a sequence of processes is produced to model a directed event broadcasting in the process that specifies the *entry* behavior of the specified state, `A` in this case; the targeted object of the broadcasting is the source state of the transition. Last, a condition is created by checking the occurrence of the declared variable in the process that represents the overall behavior of the source state of the transition.

3.5 Discussions and Discoveries

We have up to now illustrated how to transform fundamental elements of Stateflow, such as states, transitions, actions and junctions, into CSP# models. We have also explained how to cope with advanced Stateflow modeling features such as implicit events and history junctions.

Our CSP# modeling of the Stateflow execution semantics is based on Stateflow user's guide [17] which provides concrete examples to illustrate their behavior step by step. We used our approach to automatically translate the majority of these examples into CSP# models.

We validated our modeling by comparing the simulation results from our CSP# models and those from Stateflow simulations. The comparison is conducted by checking the equivalence 1) between the execution order of diagram states and that of processes, and 2) between the variable values of diagrams and CSP# models at each step. When there is a difference, we revised carefully our models and discussed our findings with experts from the MathWorks, so as to ensure that our interpretations comply with the execution semantics adopted by Stateflow. We have also validated that fundamental properties of Stateflow are retained; for example, we have proved that the property where at most one OR substate is active holds in those CSP# models by model checking.

From our rigorous modeling and checking procedures, we have discovered two previously unknown flaws as demonstrated below. These flaws have been confirmed by experts from the MathWorks and corrected in the latest version of user's guide.

One flaw is the description of the entry behavior for parallel states. Originally, (1) before executing entry actions of a parallel state, say A, "*all entry steps*" of parallel states with higher execution priorities than A are performed if they are inactive, and (2) after executing the entry actions of A, "*all entry actions*" of parallel states with lower execution priories are performed if they are inactive. The above description may result in repeated execution of A's entry actions when it activates a parallel state, say B, with a higher execution priority: first all entry steps of B are performed due to original rule (1), and those steps include performing all entry actions of A because of original rule (2); after B completes its entry steps, A continues its own entry behavior including the execution of its entry actions. However, this redundant execution behavior conflicts with the actual execution behavior from the Stateflow simulator where the entry action of A is executed only once. We fix this original description in our construction procedure for modeling *entry* behavior of a state, particularly, at Step 2 in Section 3.1.1 by separating the invocation of the parallel state with higher execution priority (in the *then*-branch) from the other entry steps of the current parallel state (in the *else*-branch); A only performs its entry actions when it is invoked by B in the above example.

Another flaw concerns the behavior of substates that are destination objects of inner transitions. First, the original user's guide misses one circumstance where a substate is active and an inner transition whose destination is this substate is valid; from our experiments, we observed that the substate became inactive and *immediately* active again when the inner transition was taken. Second, we defined generic execution semantics of this type of substates: an active substate exits and enters itself when there is a valid inner transition, no matter the substate is the transition's destination. Lastly, based on our definition, we identified an incorrect statement in [17] which stated simplification of diagrams by adopting substates with inner transitions can retain the *same* behavior. However, that statement left out the special behavior as we discovered that shows the simplification may not behave correctly.

4 Experimental Studies

We have built up a translator in C# to automate the transformation procedure illustrated in the previous section. Besides applying the translator to examples from the user's guide [17], we have also used it on several applications, including two demo cases from the MathWorks company: an alarm monitor system for a car [20], and a stopwatch with lap time measurement [7,8]. These applications cover a wide range of Stateflow modeling features such as inter-level transitions, history junctions, and implicit events. In this section, we first show how to systematically convert and rigorously analyze a fault-tolerant fuel control system, and next summarize the experimental results.

4.1 System Description and Stateflow Diagrams

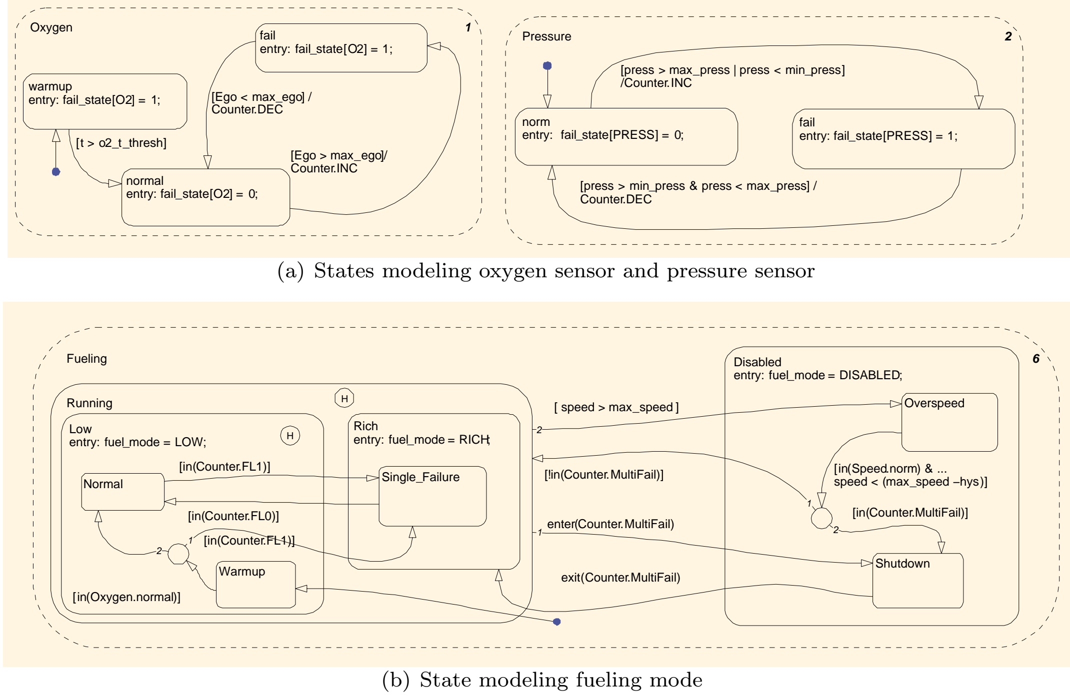
The fault-tolerant fuel control system is designed to be robust where individual sensor failures can be detected and the control system can be dynamically reconfigured. A Stateflow diagram, consisting of six parallel states, displayed in Figures 1 and 4.1, models the control logic. The four parallel states, at the top of Figure 1 and in Figure 8(a), indicate four individual sensors. The remaining two parallel states determine the overall system operation mode according to the status of four sensors.

Initially, all sensors are in their *normal* mode except the oxygen sensor; the `warmup` state in state `Oxygen` is activated in the beginning until a period (symbol `t` in Stateflow represents time) exceeds a predefined constant `o2_t_thresh`. When a sensor fails, the control systems broadcasts directly an event `INC` to state `Counter` (in Figure 1) which records the number of failed sensors.

The fueling mode of the engine is modeled by state `Fueling` in Figure 8(b). When a single sensor fails (denoted by an implicit event `in(Counter.FL1)`), the engine continues its operation and moves to a *rich* mode (in state `Rich`). When more than one sensor fails (by another implicit event `enter(Counter.MultiFail)`), the engine shuts down (at state `Shutdown`). In addition, when speed exceeds the maximum setting (`max_speed`), state `Overspeed` is activated. Note that the fueling mode after state `Running` is re-entered is decided by two history junctions in states `Running` and `Low`, respectively.

4.2 Transformation and Validation of Diagrams

The Stateflow representation of this fault-tolerant fuel control system applies several Stateflow modeling fea-



(a) States modeling oxygen sensor and pressure sensor

(b) State modeling fueling mode

Fig. 8. Fuel Control System in Stateflow

tures, such as directed event broadcasting among parallel states, inter-level transitions, history junctions and implicit events. We here show how to deal with these advanced features by transforming states `Running` and `Overspeed` in Figure 8(b); Section 3.1 demonstrated our approach to convert states in Figure 1.

State `Running` contains two substates, a history junction, and two outgoing inter-level transitions. The following process `Fueling_Running_EnAct` captures its *entry* behavior, where the first conditional choice (lines 2 to 4) checks the activity status of state `Fueling`, a superstate of `Running`. The data operation at line 5 updates the status of `Running`, followed by a conditional choice (from line 6 to line 11) depicts the behavior of its substates when `Running` is not activated by any of its substates, i.e., variable `Fueling_Running_EnterBU` being false.

```
1  Fueling_Running_EnAct()=
2    if(Fueling_Status == inactive){
3     {Fueling_EnterBU = true;}->Fueling();
4     {Fueling_EnterBU = false;}->Skip };
5    {Fueling_Running_Status = active;}->
6    if(Fueling_Running_EnterBU == false){
7       if(Fueling_Running_H == 0){ Skip}
8       else{if(Fueling_Running_H == 27){
9                Fueling_Running_Low()}
10          else{if(Fueling_Running_H == 28){
11                 Fueling_Running_Rich()}}}};
```

In the above process specification, the *then*-part of the conditional choice at line 6 captures the effect of a history junction, denoted by variable `Fueling_Running_H`, on the activation of the substates of `Running`. For instance, lines 10 and 11 depicts that state `Rich` becomes active when the variable value is equal to 28. On the other hand, the update of this history junction is specified by the following process `Fueling_Running_ExAct`. To be specific, before state `Running` becomes inactive, it assigns `Fueling_Running_H` the ID value of the substate which is inactivated. For example, lines 16 to 18 states that if state `Rich` whose ID value is 28 is inactivated, the value of `Fueling_Running_H` is updated to 28.

```
12 Fueling_Running_ExAct()=
13   if(Fueling_Running_Low_Status == active){
14      Fueling_Running_Low_ExAct();
15     {Fueling_Running_H = 27;}->Skip}
16   else{if(Fueling_Running_Rich_Status == active){
17           Fueling_Running_Rich_ExAct();
18          {Fueling_Running_H = 28;}->Skip}};
19  {Fueling_Running_Status = inactive;}->Skip;
```

Note that at the beginning of execution there is no substate to be activated in `Running` (at line 7). This is because state `Warmup`, a substate of state `Low`, is activated by default with an inter-level default transition as shown in Figure 8(b). This special behavior can be derived from the transition's textual representation below: 1) the empty content of the `src` block states that the transition is a default transition, 2) the first element, i.e., 23, of the `linkNode` parameter is the ID of state `Fueling`, and 3) the value 32 in the `dst` block indicates that the default state is `Warmup`.

```
1  transition{ id      70
2    src{               }
3    dst{ id          32}
4    linkNode      [23 69 0] }
```

The above implicit event associated with Running is captured in the following process Fueling_Running. In particular, the condition at line 4 checks if the implicit event, denoted by variable Enter_Counter_MultiFail, takes place. We note that the sequence of conditional choices from line 4 to line 12 captures the execution priorities between Running's two outgoing transitions.

```
1  Fueling_Running()=
2    if(Fueling_Running_Status == inactive){
3       Fueling_Running_EnAct()}
4    else{if(...&& (Enter_Counter_MultiFail==occurred)){
5            Fueling_Running_ExAct(); ...
6            Fueling_Disabled_Shutdown()};
7          if((speed > max_speed) && ...){
8            Fueling_Running_ExAct(); ...
9            Fueling_Disabled_Overspeed()}
10         else{if(Fueling_Running_OUTGOING == 0){
11           Fueling_Running_DurAct()}};
12          {Fueling_Running_OUTGOING = 0;}->Skip};
```

The outgoing transition from state Overspeed contains a connective junction that is the source of two outgoing transitions to different states.

```
junction{    id     35
  type  CONNECTIVE_JUNCTION }
transition{  id     60
  labelString    "[in(Speed.norm) &...]"
  src{  id      26}   dst{  id      35} }
transition{  id     61
  labelString    "[!in(Counter.MultiFail)]"
  src{  id      35}   dst{  id      25} }
transition{  id     66
  labelString    "[in(Counter.MultiFail)]"
  src{  id      35}   dst{  id      31} }
```

The above contents show these three transition segments and the junction in the MDL file, where ID 26 is Overspeed, ID 25 Running, and ID 31 Shutdown. Applying Algorithm 2 in Section 3.4.2, the translation starts at the transition with ID 60 that is the outgoing transition of Overspeed, and checks the object type of the transition's destination whose ID is 35 (at line 3 in Algorithm 2). In this case, the type is a connective junction which has two outgoing transitions with ID 61 and 66, respectively. Hence, the *for* loop between line 6 and line 10 of Algorithm 2 is executed. As shown in the following CSP# process specifying the overall behavior of Overspeed, transitions from the connective junction are converted into a sequence of conditional choices between line 4 and line 9, and the sequence order follows their execution order priorities.

```
1 Fueling_Disabled_Overspeed()=
2   if(Fueling_Disabled_Overspeed_Status == inactive)...
3   else{if(((Speed_norm_Status == active) &&...){
4     if(!((Counter_MultiFail_Status == active))&&...){
5        Fueling_Disabled_ExAct();...Fueling_Running()};
6     if((Counter_MultiFail_Status == active) &&...){
7        Fueling_Disabled_Overspeed_ExAct(); ...
8        Fueling_Disabled_Shutdown()}
9     else{...}} ...};
```

We note that "in(state_name)" is an implicit event that occurs when the specified state is active. Thus, in our above CSP# specification, we represent this type of implicit events by checking the activity status of their corresponding states.

The whole diagram is translated into a process named Chart that describes the diagram behavior at a sample time step denoted by event click (mentioned in Section 3.4.1). The process shown below executes six processes of six parallel states based on their execution priority order; initially, only state Oxygen is invoked (at line 3), because the process modeling the entry behavior of Oxygen will activate other parallel states with lower execution priority, see Step 5 in Section 3.1.1 Note that no click event occurs when variable broadcast is not equal to value 0 (at line 8). Moreover, the update of time symbol t used in Oxygen is associated with the click event (at lines 2 and 5).

```
1  Chart()= if(Chart_init == true){
2    click{t = (t  +  1);}->
3    atomic{{Chart_init= false;}->Oxygen()}}
4   else{if(broadcast == 0){
5      click{if(t < 5){t = (t  + 1)};}->
6      atomic{Oxygen(); Pressure(); Throttle(); Speed();
7             Counter(); Fueling()}}
8     else{ atomic{Oxygen(); Pressure(); Throttle();
9                  Speed(); Counter(); Fueling()}}};
```

This control system receives four inputs from its environment sensors. For the sake of simplicity, we restrict the input type to integer. The *random* valuation of these four inputs is captured by the process Initialization using general choices (discussed in Section 3.4.1). With process Initialization, the periodic behavior of the control system is modeled by process Stateflow below.

```
1  Initialization()= atomic{
2    ([]x:{0..2}@{Ego=x} -> Skip);
3    ([]y:{0..6}@{press = y} -> Skip);
4    ([]z:{0..4}@{throt = z} -> Skip);
5    ([]w:{0..5}@{speed = w} -> Skip) };
6  Stateflow()= Chart();Initialization();Stateflow();
```

Our transformed CSP# models preserve the hierarchical structure of corresponding Stateflow diagrams, and capture *all* information such as states, variables and events. For example, variable Counter_FL0_Status indicates the activity status of state FL0 within its superstate Counter, and INC denotes a broadcasting event *INC* in Figure 8(a). This one-to-one correspondence allows users to easily specify desired properties of the diagrams using the CSP# models. We present below four safety properties of the control system which have been verified by the model checking capabilities of PAT.

Case	States	Trans	Junc	Addition
1	30	32	2	inter-level transitions, directed event broadcast, history junctions, array datatype, implicit events, temporal constraints
2	9	14	0	directed event broadcast, temporal constraints
3	10	13	1	implicit events
4	6	15	4	inter-level transitions, inner transitions

Table 1. Summary of Diagram Features

```
#define R1 !(Pressure_fail_Status == active)
         ||(Counter_FL0_Status == inactive);
#assert Stateflow() |= [] R1;
#define R2 !(Pressure_fail_Status == active &&
            Throttle_fail_Status == active)
       ||(Counter_MultiFail_Status == active);
#assert Stateflow() |= [] R2;
#define R3 !(Counter_MultiFail_FL4_Status == active &&
            INC == occurred);
#assert Stateflow() |= []R3;
#define InterMulti Counter_MultiFail_Status == active;
#define EngineDown Fueling_Disabled_Status == active;
#assert Stateflow() |= [](InterMulti -> <> EngineDown);
```

- `R1` ensures that the failure of a sensor (e.g., state `fail` becomes active in state `Pressure`) is captured by state `Counter`, namely, state `FL0` is inactive.
- `R2` checks that when there are at least two failed sensors, the counter must be greater than value 1, i.e., state `MultiFail` in Figure 1 is active.
- `R3` indicates that when the counter reaches the maximum value 4 (denoted by state `FL4`), no `INC` event can be generated from any sensor.
- The last assertion requires that when there are more than one failed sensors (denoted by `MultiFail`), the fueling mode becomes disabled eventually. In other words, state `Disabled` is activated.

4.3 Summary of Experiments

Besides the above fuel control system, we have also transformed and validated other systems. Table 1 lists the modeling features of those systems' Stateflow diagrams; not only basic elements such as states, transitions, and connective junctions are used, additional features like event broadcasting and implicit events occur. Note that the one-to-one correspondence between our transformed CSP# models and Stateflow diagrams facilitates users to quickly relate problematic traces of CSP# models (denoting a counterexample) to Stateflow diagrams.

- Case 1 is the fault-tolerant fuel control system, our running example throughout this paper.
- Case 2 is another demo from the MathWorks Company[3]. It models gear selection in an automatic transmission. The temporal constraints modeled in this Stateflow diagram return true *after* the diagram wakes up *user-specified* times *since* activation of their associated states.
- Case 3 is an alarm monitor system for cars designed to fulfill two safety properties [20]. One property (`R1`) ensures that car doors are locked when the car exceeds a predefined speed, and the other (`R2`) triggers a belt alarm when the car exceeds a specific speed and the seat belts are not fastened. The PAT model checker analyzed the transformed CSP# models of the Stateflow diagram, and detected subtle bugs violating both properties, respectively [4]. One counterexample depicts that: the car can exceed the predefined speed before turning on its engine (e.g., moving on a slope), and thus after the engine is on, the default state where doors are unlocked is entered without checking the speed. The other counterexample revealed also the necessity of adding conditions to different default states.
- Case 4 is a stopwatch with lap time measurement [8]. Its Stateflow diagram adopts *inner* transitions to specify the counting of time from seconds to minutes and to hours. From our experiment, we identified a bug that the time for display may not be equal to the digital clock (`R1`) when several events occur between two digital clock clicks; the update of the time is a *during* action which cannot take place when the state of that action becomes active and then inactive at a pair of adjacent of clock clicks.

[3] `http://www.mathworks.com/products/stateflow/demos.html?file=/products/demos/shipping/simulink/sldemo_autotrans.html`

Case	Property	Verdict	Visited States	Time(Sec)
Fuel Control	[]R1	Yes	3,021,176	63
Fuel Control	[]R2	Yes	3,021,176	65
Fuel Control	[]R3	Yes	3,021,176	58
Fuel Control	[]R4	Yes	3,178,838	173
Alarm Monitor	[]R1	No	9303	0.19
Alarm Monitor	[]R2	No	3993	0.07
Stopwatch	[]R1	No	140	0.01

Table 2. Validation Result

Table 2 shows the validation results of those cases using PAT, where property `R4` of the fuel control system is `InterMulti -> <> EngineDown`. The result includes the number of visited states and time (in seconds). The computer running the experiment is equipped with Intel Core Duo CPU at 1.86GHz and 2GB memory.

5 Conclusion

We have demonstrated a systematic approach to transform and validate Stateflow diagrams based on a generic model checker PAT. The automatic translation from Stateflow models to CSP# models in PAT preserves the execution semantics of Stateflow, and it covers advanced Stateflow modeling features such as implicit events and history junctions. The transformed CSP# models are executable, and this enables us to validate our interpretation of Stateflow semantics by means of simulations, specifically, comparing execution sequences and variable

values step by step between CSP# models and Stateflow semantic examples from its user's guide. Moreover, users can verify their systems against important safety and liveness properties based on the model checking facilities of PAT. We have applied our approach to several examples including two demo cases from the MathWorks Company, where we discovered subtle defects in Stateflow user's guide and demo cases.

Our modeling language CSP# lacks direct support for floating points and structured datatypes. Nonetheless, PAT supports user-defined data structures written in programming languages such as C#, C, and Java. Thus, we can construct C# (C or Java) floating point fields or a structured datatypes and then invoke them in CSP# models. Note that using C# is straightforward in PAT since PAT is developed based on the .NET framework; other programming languages can be used in CSP# models by leveraging some bridging libraries (e.g., using JNBridge to call Java in .NET). One catch is that we must ensure that there are only finitely many different values for any of those datatypes. Separately, human effort is still needed to map execution trace of CSP# models to corresponding Stateflow diagrams when there is a violation of desired properties. Automating this mapping process is one future work.

Acknowledgements. The authors would like to thank all the reviewers for their constructive comments which help us improve the paper. The authors are also grateful to Melody Yung, John de Leon, and Wiriyanto Darsono from the MathWorks Company for their assistance in using Stateflow. This work is partially supported by the following projects: A*STAR SERC PSF 1121202016, MOE2009-T2-1-072, TRF Project "Research and Development in the Formal Verification of System Design and Implementation", and IDG31100105 / IDD11100102 from Singapore University of Technology and Design.

References

1. S. Bacherini, A. Fantechi, M. Tempestini, and N. Zingoni. A story about formal methods adoption by a railway signaling manufacturer. In *FM'06: Proceedings of the 14th International Symposium on Formal Methods*, pages 179–189. Springer, 2006.
2. C. Banphawatthanarak, B. H. Krogh, and K. Butts. Symbolic verification of executable control specifications. In *CACSD'99: Proceedings of the 10th International Symposium on Computer Aided Control System Design*, pages 581–586. IEEE, 1999.
3. A. Cavalcanti. Stateflow diagrams in circus. *Electronic Notes in Theoretical Computer Science*, 240:23–41, 2009.
4. C. Chen. Formal analysis for stateflow diagrams. In *SSIRI-C'10: Proceedings of the 4th International Conference on Secure Software Integration and Reliability Improvement Companion*, pages 102–109. IEEE Computer Society, 2010.
5. J. Edmund M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
6. A. Ferrari, A. Fantechi, S. Bacherini, and N. Zingoni. Formal development for railway signaling using commercial tools. In *FMICS'09: Proceedings of the 14th International Workshop on Formal Methods for Industrial Critical Systems*, pages 197–198. Springer, 2009.
7. G. Hamon. A denotational semantics for Stateflow. In *EMSOFT'05: Proceedings of the 5th ACM International Conference on Embedded Software*, pages 164–172. ACM, 2005.
8. G. Hamon and J. M. Rushby. An operational semantics for Stateflow. *International Journal on Software Tools for Technology Transfer*, 9(5-6):447–456, 2007.
9. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
10. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. B. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, 1990.
11. M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria. Software engineering and formal methods. *Communications of the ACM*, 51(9):54–59, 2008.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
13. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML Statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.
14. Y. Liu, J. Sun, and J. S. Dong. Analyzing hierarchical complex real-time systems. In *FSE'10: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 365–366. ACM, 2010.
15. Y. Liu, J. Sun, and J. S. Dong. PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In *ISSRE'11: Proceedings of the 22nd annual International Symposium on Software Reliability Engineering*, pages 190–199. IEEE, 2011.
16. The MathWorks. *Simulink® Design Verifier™ 1 - User's Guide*, September 2009.
17. The MathWorks. *Stateflow® and Stateflow® coder™ 7 - User's Guide*, March 2009.
18. L. Ng, P. Hubbard, and S. O'Young. Simulation of fully autonomous control of unmanned air vehicles for maritime surveillance. In *Proceedings of the 2010 Spring Simulation Multiconference*, SpringSim'10, pages 40:1–40:9. ACM, 2010.
19. M. Oliveira, A. Cavalcanti, and J. Woodcock. A utp semantics for circus. *Formal Aspects of Computing*, 21(1-2):3–32, 2009.
20. N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. In *EMSOFT'04: Proceedings of the 4th International Conference on Embedded Software*, pages 259–268. ACM, 2004.
21. S. Sims, R. Cleaveland, K. Butts, and S. Ranville. Automated validation of software models. In *ASE'01: Proceedings of the 16th IEEE International Conference*

on Automated Software Engineering, pages 91–96. IEEE Computer Society, 2001.
22. J. Sun, Y. Liu, J. S. Dong, and C. Chen. Integrating specification and programs for system modeling and verification. In *TASE'09: Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 127–135. IEEE Computer Society, 2009.
23. J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards flexible verification under fairness. In *CAV'09: Proceedings of the 21th International Conference on Computer Aided Verification*, pages 709–714. Springer, 2009.
24. A. Tiwari. Formal semantics and analysis methods for Simulink Stateflow models. Technical report, SRI International, 2002. See URL: `http://www.csl.sri.com/~tiwari/~stateflow.html`.
25. I. Toyn and A. Galloway. Proving properties of stateflow models using ISO standard Z and CADiZ. In *ZB'05: Proceedings of the 4th International Conference of B and Z Users*, pages 104–123. Springer, 2005.
26. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4), 2009.