PROGRAMMING LANGUAGES

VERIFICATION OF STATECHART MODELS

Neetha Reddy (IMT2018050)
Arpitha Malavalli (IMT2018504)
Tanishq Jaswani (IMT2018077)

Aim:

The aim of this project shall be to explore and understand various methodologies involved in the verification of statechart models, and it translates to verification of the modeled system.

Formal Verification:

Formal verification is the act of proving or disproving the **correctness** of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

We encounter infinitely many software and hardware systems around us each day. Each of them is controlled by some instructions/code. In some systems, ensuring that the controller is “CORRECT” is of supreme importance - In Aviation systems , Medical systems, space missions etc.

While designing such safety-critical systems, it is important to ensure that the design is reliable i.e it satisfies all the properties expected from it , 100% of the time.

Some ways to ensure this is to :

1. Reduce chances of error while designing by using best software development best practices
2. Testing: manually checking a large number of test cases to identify possible errors

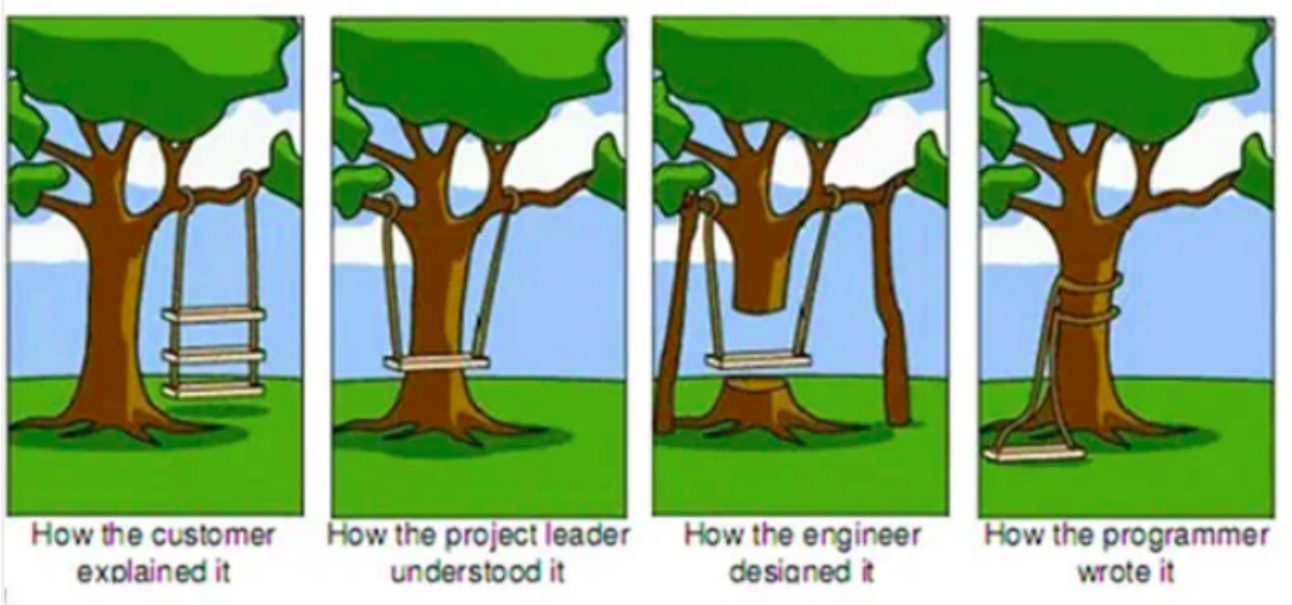
However, there is always a chance that some errors are not caught , and thus the system designed cannot be claimed as CORRECT. Further as the size and complexity of the underlying system increases, it becomes hard to generate test cases to cover all possible scenarios.

When errors are both difficult to detect using standard testing techniques and very expensive, Formal Verification techniques come to our aid. The correctness of the (software or hardware) system is mathematically proven. Hence it can be said to be always reliable!

Formal verification requires Formal Specification:

Verification is used to establish that the design or product under consideration possesses certain **properties**. The properties are mostly obtained from the system’s specification. This **specification** prescribes what the system has to do and thus this constitutes the basis for any verification activity. (eg: Property: a system should never be able to reach a situation in which no progress can be made (a deadlock scenario))

However, if the system specification and requirements is available only in the form of natural language, there is bound to be a large amount of ambiguity on the designer's end. Thus, doing a formal verification in this case, will not serve the required purpose of ensuring reliability.



Hence in order to carry out formal verification, 2 initial steps are essential:

1. Formal description of the system
2. Formal specification of the required properties

This specification prescribes what the system has to do and what not,
and thus constitutes the basis for any verification activity.
A **defect** is found once the system does not fulfill one of the specification's properties. The system is considered to be "**correct"** whenever it satisfies all properties obtained from its specification. **So correctness is always relative to a specification, and is not an absolute property of a system**.

Validation vs Verification:

Verification involves checking if the system (formally described model) indeed satisfies all the specification/ formally defined properties.
However, it may so happen that satisfaction of specified properties may not imply that the requirements of the systems is indeed met. Thus, there is a related problem of validation, which ensures that the properties considered indeed cover all the requirements.
Thus,
verification is "checking that we are building the thing right"
validation is "check that we are verifying the right thing".

Specification Models:

Complex systems in modern day essentially need specification models. The existing specification models need to evolve alongside the development of coding practices like incremental development and modularization.

A **statechart** is a visual formalism for the specification of complex systems. Statechart diagrams are one of the five UML diagrams used to model the dynamic nature of a system. They define different states of an object during its lifetime and these states are changed by events.

The Statechart Model allows for development practices like **incremental development and modularization** as required in modern day. However, clinical validation is often not adequate for guaranteeing the correctness and safety of cyber-physical systems, and formal verification is required.

Statecharts provide us the benefit of easier verification . The statechart models can be automatically transformed to a verifiable formal model, such as timed automata, so that existing formal verification tools can be used to verify required safety properties.

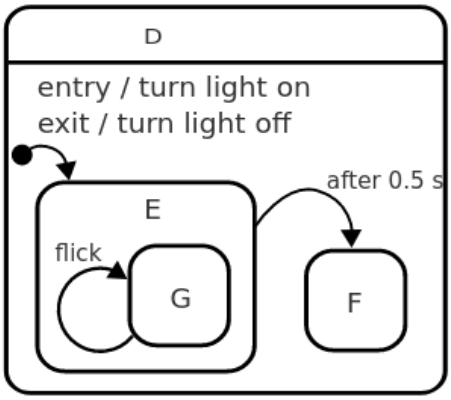
Statechart is widely accepted in the software engineering community, since the growing complexity of softwares has called for an increased level of engineering and consequentially, increased levels of specification and analysis.

The primary motivation behind this model was to **overcome the limitations inherent in conventional state machines** in describing complex systems – proliferation of states with system size and lack of means of structuring the descriptions.
The statecharts model extends state machines along three orthogonal dimensions – **hierarchy, concurrency and communication** – resulting in a compact visual notation that allows engineers to structure and modularise system descriptions.

1. **Hierarchy** is the ability to cluster states into a superstate (**an OR state**), or refine an abstract state into more detailed states; when a system is in a superstate, it is in exactly one of its sub-states. Visually the hierarchy is denoted by physical encapsulation of states (rounded rectangles).
2. **Concurrency** denotes orthogonal subsystems that proceed (more or less) independently and is described by an **AND decomposition of states:** when a system is in a composite AND state, it is in all of its AND components. Visually an AND state is depicted by physically splitting a state using dashed lines.
3. **Communication** between concurrent components is via a broadcast mechanism. Variants using directed communication along named channels is also common. Both synchronous and asynchronous styles of communication have been proposed.

Statechart:

Here's an example of a state in a state machine, with some extra features:



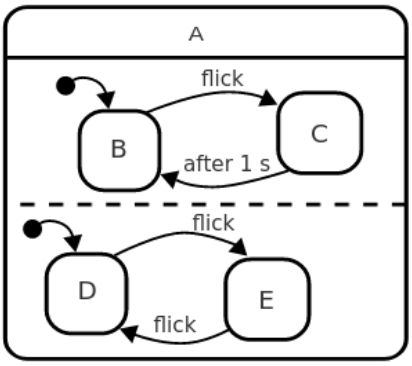
- The state is called "D"
- E and F are substates
- E even has a sub-state G
- "entry" and "exit" actions are mentioned

When the state machine enters this state D it also starts the state machine within it. The initial state of this machine inside D is in fact E so it enters E too. And since E has a single sub-state G, it is also entered. A state with no sub-states is called an atomic state. A state with sub-states is called a compound state.

Entering a state enters one of its sub-states

- When a state is entered, its sub state machine starts and therefore, a sub-state is entered
- When a state is exited, its sub state machine is exited too, i.e. any substates also exit.

A compound state can be split up into completely separate ("orthogonal") regions. Each region specifies its own state machine. When a state machine enters such a state, it also enters *all* of the regions of the state at the same time.



If this state is entered, then it enters both the top and bottom regions, and the following statements will hold true so long as A is active:

- Exactly one of B and C is active
- Exactly one of D and E is active

Such a state is called a parallel state, and the regions are often called "orthogonal regions".

Every Statechart model contains 4 elements:-

- S: Set of one or more states
- sc: Single unique state which will contain all elements such as states, transitions etc for the statechart.
- A: Transitions to execute i.e. Entry, Exit, make a transition.
- E: To trigger the transitions by an external interaction by the model.

Every State is represented by 9 elements:-

- n: Name of the state
- p: unique parent
- V: set of variables - static, local, parameter depending on their scope.
- Entry action
- Exit action
- T: Set of transitions
- C: Child state set
- h: History marker
- i: Initial sub-state

Every transition is represented by 7 elements:-

- n: Name of the transition
- p: unique parent state p for the transition
- src: source state
- e: event e which triggers the transition
- g: guard statement for the transition
- a: entry/exit
- dest: Destination state

All these features entail a rather complex structure on the transition:

1. **Transitions can be guarded**

When an event happens, and the state machine would normally transition from one state to another, statecharts introduce the concepts of *guarding* the transition. A guard is a condition placed upon the transition, and the transition is essentially ignored if the guard condition is false.

2. **States can have multiple transitions for the same event**

The addition of guards allows a state to have *more than one* transition that reacts to the same event: Two transitions fire on the same event, but only one is picked at run-time depending on which one's guards evaluate to TRUE.

3. **Transitions can happen automatically**

When entering a state, a transition can be defined which is automatically taken. This is useful in conjunction with guards, to move out of a state immediately upon entering it when certain conditions hold, or as soon as those conditions hold.

4. **Transitions can be delayed**

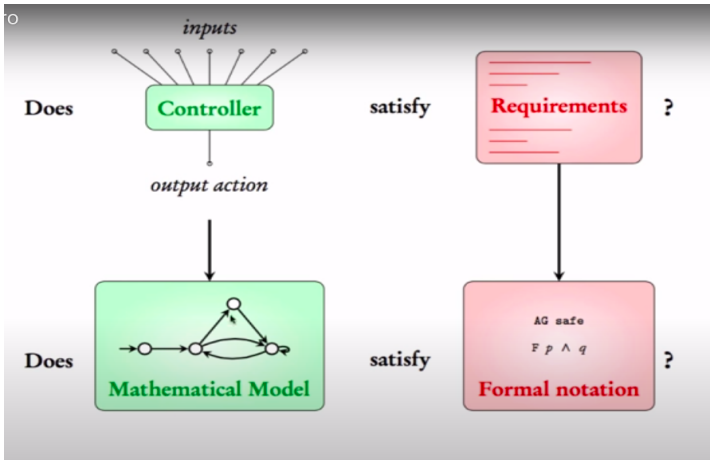
Simply being in a state for a duration of time can be enough to transition to a different state. This is accomplished by way of a delayed transition, which defines that the transition should take a specific period of time after entering a state.

5. **History**

When exiting a compound state and its sub-state, it is sometimes useful to be able to return to exactly the state that you left. When a transition goes to a history state, it re-enters the state that "was last active".

Model Checking:

Model checking constructs a mathematical model to describe the underlying system and represents its requirements as properties in some formal notation . Then, checking if the mathematical model satisfies the properties is equivalent to verifying if the system satisfies the requirements.



Given an

1. input model
2. property in temporal logic(logic specifying properties over time)

A model checker determines whether the property holds , by exploring all possible system states in a brute-force manner and examining all possible system scenarios in a structured manner.
It returns a counterexample trace in case the property fails. In this way, it can be shown that a given system model truly satisfies a certain property.

A model checker is a software tool that

- given a description of a Kripke model M ...
- ... and a property Φ,
- decides whether $M \models \Phi$,

Phases in Model checking are:

• **Modeling phase:**
– model the system under consideration using the model description language of the model checker at hand; (**StateTransition diagrams!)**
– as a first sanity check and quick assessment of the model perform some simulations;(Eliminating these simpler errors before any form of thorough checking takes place may reduce the costly and time-consuming verification effort.)
– formalize the property to be checked using the property specification language.(CTL,LTL, OBDDs etc)

• **Running phase**:
run the model checker to check the validity of the property in the system model.
This is basically a solely algorithmic approach in which the validity of the property under consideration is checked in all states of the system model. The algorithm varies based on how the properties are defined.

• **Analysis phase:**
– property satisfied? → check next property (if any);
– property violated? →
1. analyze generated counterexample by simulation;
2. refine the model, design, or property;
3. repeat the entire procedure.

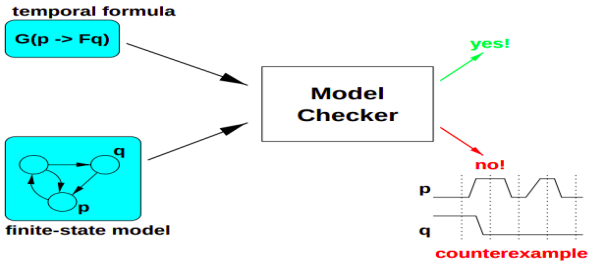
– out of memory? → try to reduce the model and try again.

Model - checking for statecharts:

In the model checking framework, the system is represented as a transition system.
Thus if the system is specified as a statechart, it is usually converted to represent a transition system.
The state hierarchy however, is present only in statecharts. The straightforward way to represent a statechart as a transition system is to flatten its hierarchy. However, this can lead to an exponential blowup in size, particularly when there is a lot of sharing of states and transitions, in which case other methods can be used like abstraction.

Temporal Logic:

As mentioned before, the properties in model checking are represented in temporal logic formulae. The Model is usually a transition system.



1. A standalone property is called an atomic property (p).
2. The set $AP = p_1, p_2 \ldots p_k$ represents the set of all atomic properties.
3. Every state in the transition diagram satisfies a subset of properties in AP. Thus for each state $S \subseteq 2^{AP}$ represents the set of properties it satisfies.
4. If a trace of the transition system is mapped to the properties the states in the trace satisfy $t(S)$, it will result in an infinite word over the powerset . ($\Sigma = 2^{AP}$).

AP-INF = set of **infinite words** over *PowerSet*(**AP**)

Property 1: p_1 is always true

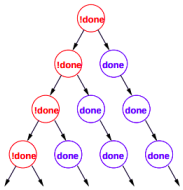
$\{ A_0A_1A_2 \cdots \in \text{AP-INF} \mid \textbf{each } A_i \text{ contains } p_1 \}$

$\{p_1\}\{p_1\}\{p_1\}\{p_1\}\{p_1\}\{p_1\}\{p_1\}\cdots$
$\{p_1\}\{p_1,p_2\}\{p_1\}\{p_1,p_2\}\{p_1\}\{p_1,p_2\}\cdots$

5. A property $\phi = f(p_1, p_2 \ldots p_k)$ itself is a subset of the set of infinite words over the power set .
6. A transition system is said to satisfy a property, if the set of all possible traces (from all initial states) in the transition system is a subset of AP-INF. $T(S) \subseteq AP - INF$

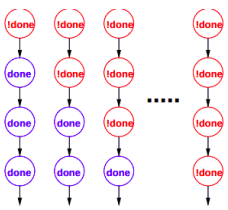
Now, the execution of any transition system can be viewed in two ways:

An infinite computation tree:



CTL used

A infinite set of paths



LTL used

Now once the property is defined, an exhaustive check of the property is done over all states of the flattened statechart.

The bottleneck:

1. Exhaustive analysis may require to store all the states
2. The state space may be exponential in the number of components
3. State Space Explosion: too much memory required

The solution:

1. Symbolic Model Checking:
2. Different search algorithms

---

2 examples of complete model checking are explained in presentation (slides 14-19)
Example1:
Property expressed as: **LTL Formulae**
Model Checking algorithm: **Bounded Model Checking**
Example 2:
Property expressed as: **CTL Formulae**
Model Checking algorithm: **Bounded Model Checking**

Paper Review:

Exploiting Hierarchy in the Abstraction-Based Verification of Statecharts Using SMT Solvers

As the set of possible configurations for a system becomes unmanageably large or even infinite, the above techniques fail.

A possible solution to overcome this issue is to use **abstraction**, which is a generic technique for reducing complexity by hiding details that are not relevant for the property to be verified. However, it is a difficult task to find the proper precision of abstraction, which shall be coarse enough to avoid complexity issues but fine enough to prove the desired property.

Counterexample-Guided Abstraction Refinement (CEGAR) is an automatic technique that initially starts with a coarse abstraction and refines it iteratively based on the counterexamples until the proper precision is obtained.

A challenge in applying verification algorithms on statechart models is the presence of

1. Parallelism
2. Hierarchy

In order to overcome this problem the authors of the paper have come up with an unique encoding function.

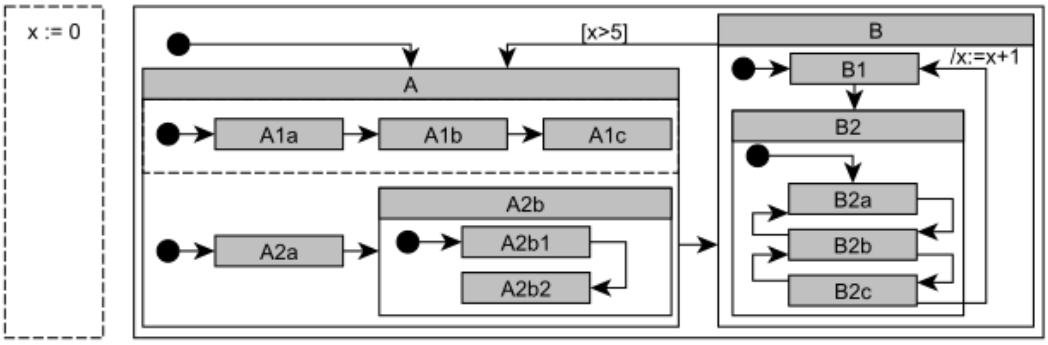
The algorithm used ensures that parallelism and hierarchy of statecharts is maintained when encoding states to bit vectors.

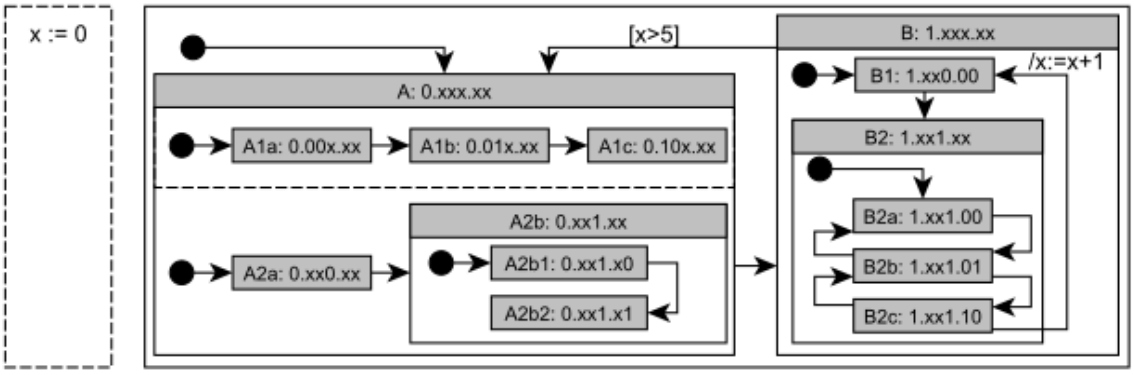
The transitions are then modeled as formulae on the bit strings.

Eg: For a transition t to fire at step i, each of these formulas have to be satisfied,

$$form(t)_i = form(src(t))_i \wedge form(trgt(t))_{i+1} \wedge form(trig(t))_i \wedge grd(t)_i \wedge form(act(t))_i.$$

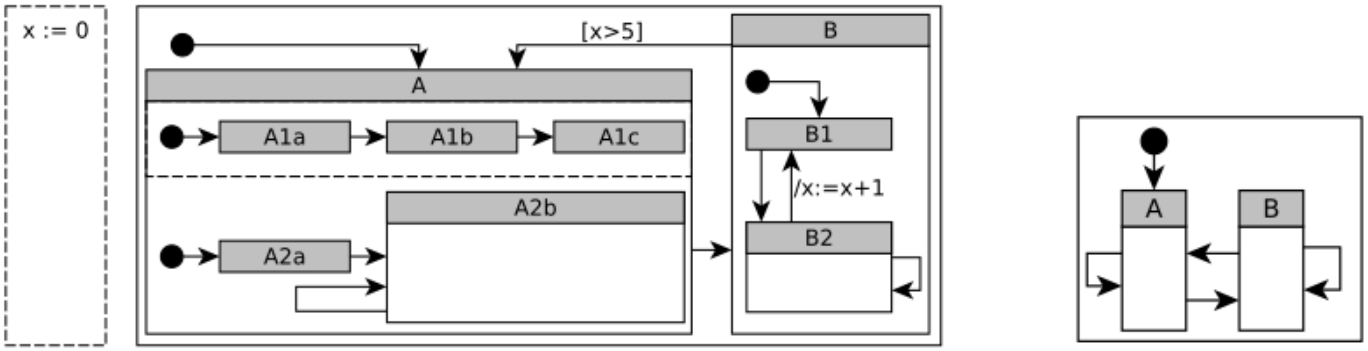
A statechart before and after encoding:





Here we can see the encoding takes care of both regions and depth of a state. Thus we can know the parents, children of a state instantly. Also in depth transition information is not lost!

Abstraction + Model Checking.



1. The statechart is first converted to the coarsest abstraction Sc.
2. The verification of the model may be performed by exploring the abstract state space, which is significantly smaller than the state space of the actual statechart. Due to the existential property of the abstraction, each concrete path in the statechart has its corresponding abstract path by simply mapping the states and transitions to their abstractions.
    a. Therefore, if no error configuration can be reached in the abstract, then it cannot be reached in the concrete statechart and the algorithm reports that the statstatechart Sc is safe.
    b. On the other hand, if an error configuration can be reached, an abstract path is returned as a counterexample. However, it is not guaranteed that there is a corresponding counter example in the concrete statechart. We begin to concretise the abstract path:
        i. if concretization succeeds, the concretized counterexample is returned.
        ii. Otherwise, if concretization succeeds until the ith iteration, but fails in the (i + 1)th iteration, then i is called a failure configuration. The abstract statechart has to be **refined** based on the failure configuration. This means more levels are added into the abstract statechart and the process repeats.

[P.T.O]

Paper Review:

Verification of UML Statechart Models of Embedded Systems

- Automated verification of behavioural requirements through model transformation and application of an off-the-shelf model checker.

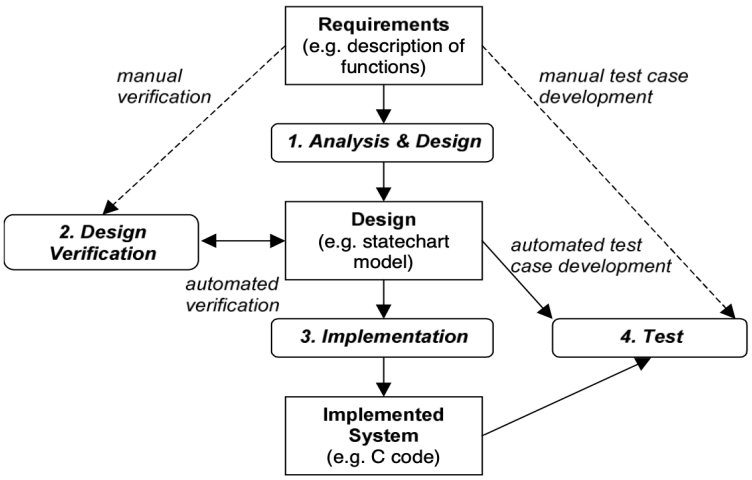


Figure 1: The relationship of the different phases of system development

- The test phase is intended to recover the differences between the design and the implemented system, i.e. to find the faults that arise in the implementation phase.
- Test phase activities - test case definition and test case execution.
- Test case execution automation on the basis of design is supported by CASE tools
- Automated test case definition can only be implemented if the design is described by a formal model.
- Design verification phase to find faults in the design itself (like design inconsistency or divergence of the design from original requirements).
- Concurrency and distribution are common principles in embedded systems. Object-oriented (OO) paradigm is popular these days for concurrent system design.
- Active objects represent (independent) threads of control, passive objects represent data structures used in the computation.

- Interact through signals. A state transition in an object results in a send action, which raises a signal. Another object, through an event queue, receives the signal which triggers a state transition (and thus possible other actions).
- Designing concurrent OO systems usually necessitates a thorough verification of concurrency control, i.e. synchronization and communication to prevent deadlocks or other hazardous states.
- Implemented a tool set which is able to transform UML models of concurrent OO systems to the input format of the model checker SPIN. SPIN is a widely used tool for analyzing the logical consistency of distributed systems. This tool was selected since its formal modelling language Promela (Process Meta Language) allows creation of concurrent processes and communication can be directly modelled using both synchronous and asynchronous channels.

Statechart diagrams to Promela in two steps:

1. Each statechart diagram is transformed to a semantically equivalent formal model called Extended Hierarchical Automata (EHA) which describes statechart elements concisely. Resolves inter-level and composite transition problems using special labels.

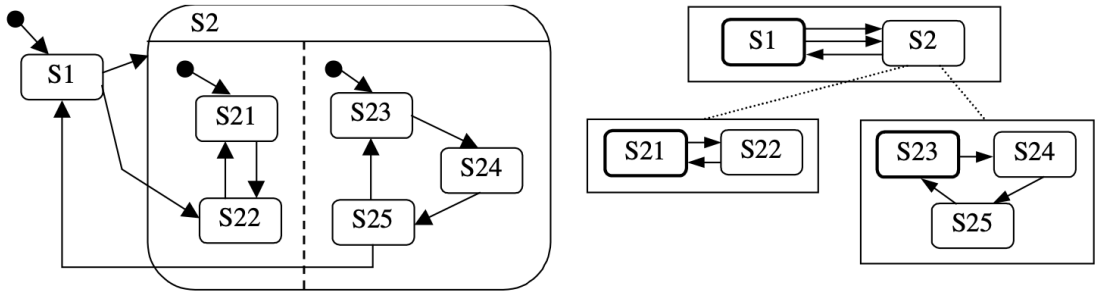


Figure 2: An UML statechart and the structure of the corresponding EHA (the concurrent regions in the refinement of state S2 are represented by two separate sequential automata in the EHA)

2. Based on the formal operational semantics of the EHA representation (Kripke structures), the EHA is transformed to Promela.
3. The papers referred to allow the verification of a single statechart and this approach was extended to allow verification of multiple statecharts(objects) of a distributed system, communicating through event queues.

Approach introduces Modelling conventions for Event queues

- EQs modelled by instances of stereotyped UML classes with attributes defining size of the queue and the policy of the dispatcher (FIFO or set)
- Objects belonging to the same EQ are grouped in packages.
- In the formal model, each EHA is assigned to an event queue belonging to the package of the corresponding object. Targets of send actions are specified by named objects, send actions without target result in broadcast events.

Implementation of model transformation

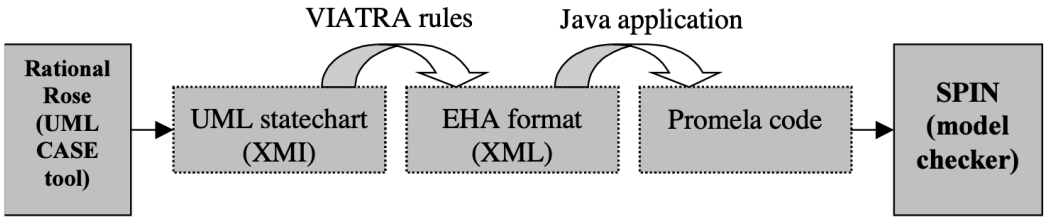


Figure 3: Implementation of the model transformation

Interrupt Controller Example

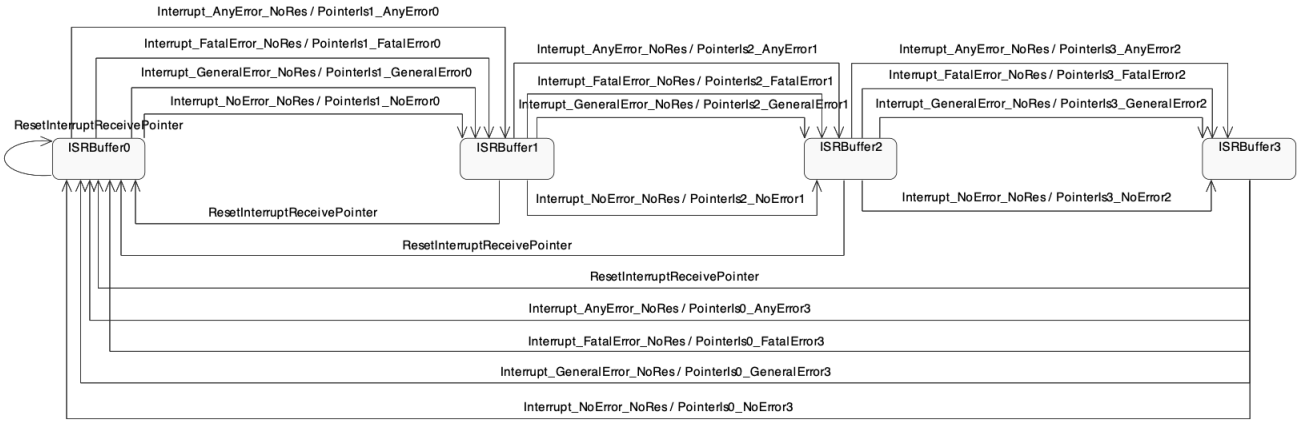


Figure 4: Statechart of the interrupt controller of the ASI Master

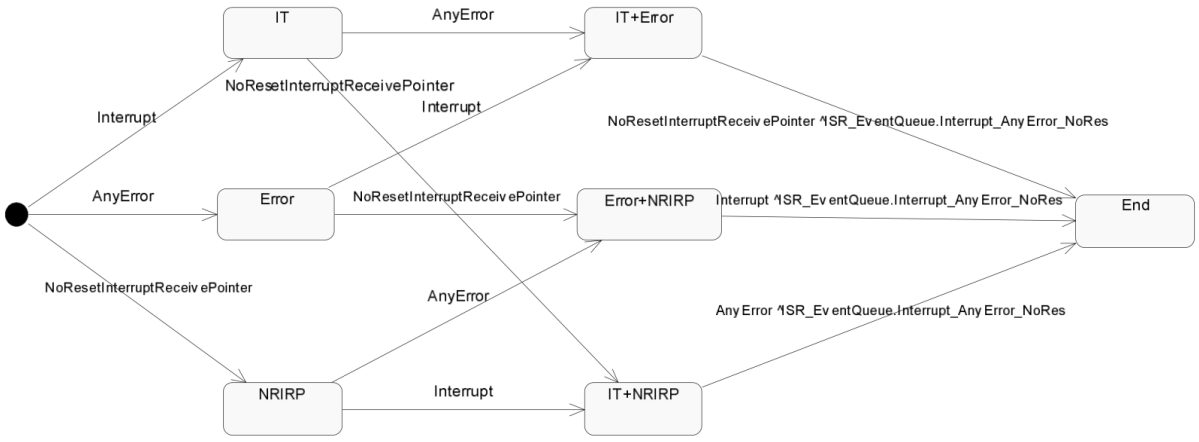


Figure 5: Statechart diagram of the event generation

The event generation condition is *Interrupt_AnyError_NoRes* .

Verification Results:

- The first verification step targeted the checking of unreachable code resulting from design errors. This helps check errors arising due to inconsistent use of names of trigger events and send actions (spelling mistake).
- Verifying invalid end states proves the existence of deadlock in the system.
- Reachablilty of some selected state configurations (Active states in separate components)

Design Verification:

- Two ways of using SPIN for verification: the system model specified in Promela can be simulated or the fulfillment of formalized requirements can be examined.
- During verification, SPIN automatically checks for deadlocks and unexecutable code.
- Valid end states are marked with an *end* label so that the tool can decide whether it is a deadlock or proper end state of the system. Further, the model may wait in a cycle and never reach an end state, so adding a label indicates that it is not a failure.
- Accept and progress labels are introduced to check validity of the cycles. The examined system can be analyzed for correctness using these labels (EX: *never* process).
- For special behavioural requirements, designers need not have knowledge about these labels since SPIN provides automatic translation from LTL formulae into the never *process*.
- Verification is performed by exhaustive state space search, i.e. examining every possible sequence of execution. If behavioural violation occurs(deadlock or terminated never process), SPIN stops verification and starts a simulation showing the violating execution.

Conclusions:

- Avoid logical design errors in an early design phase before implementation, test generation and execution begins.
- Application constrained by exhaustive nature of verification : state space explosion. So analysis is reduced to core critical parts of the system can be considered here.

REFERENCES:

[1] Czipó, Bence & Hajdu, Ákos & Tóth, Tamás & Majzik, István. (2017). Exploiting Hierarchy in the Abstraction-Based Verification of Statecharts Using SMT Solvers. Electronic Proceedings in Theoretical Computer Science. 245. 31-45. 10.4204/EPTCS.245.3.
[2] Darvas, Adam & Majzik, István & Benyó, Balázs. (2002). Verification of UML Statechart Models of Embedded Systems.
[3] Baier, Christel & Katoen, Joost-Pieter. (2008). Principles of Model Checking.
[4]Bhaduri, Purandar & Ramesh, S.. (2004). Model Checking of Statechart Models: Survey and Research Directions.
[5]Edmund M. Clarke, Jr. Computer Science Department Carnegie Mellon University Pittsburgh, PA 15213:Lecture1: Symbolic Model Checking with BDDs
[6]M. Pistore and M. Roveri,IIT Delhi - India,Feb 6, 2004: Symbolic Model Checking Slides
[7]https://github.com/AmoghJohri/StatechartSimulator/blob/master/ProjectReport.pdf
[8]NPTEL course on Model Checking: https://www.youtube.com/watch?v=piISG8bV2GI&list=PLJ5C_6qdAvBGojQMUzL4x5Y0N5gBJmT4l
[9]Introduction to model checking, Prof. Kaoten, 2018: https://www.youtube.com/watch?v=KrWSK-UzCRc