

Formal Verification of STATEMATE-Statecharts

Kay Fuhrmann, Jan Hiemer

Abstract. Our approach is a framework for the verification of statechart properties. We translate a subset of the statechart language from the tool STATEMATE into CSP for verifying temporal properties based on the CSP theory of *trace refinement*. As practical framework we have implemented a prototypical tool STATEPRO based on STATEMATE and the CSP-based verification tool FDR.

1 Introduction

During the software development process it is important to use powerful techniques for proving the expected behavior of the system and hence avoiding failures in real applications later on. Therefore, an effective means to increase confidence in the development process is to use analysis techniques based on the formal descriptions used in early phases (requirements specification).

The STATEMATE tool HLN⁺90 is widely used for developing industrial software systems, e.g. avionic systems, automotive systems, computer operating systems etc. This paper concentrates on the STATEMATE-*statecharts* Har87 as a description language for dynamic system behavior. Statecharts is a highly structured language extending conventional state transition diagrams by *hierarchy*, *concurrency* and *communication*. Within the STATEMATE tool users specify and validate these descriptions by a simulation tool which is driven interactively. Simulation is bounded in the way that it only shows some parts of a system and cannot provide evidence of requirements being satisfied over the whole life time of a system. The industrial practice of Statechart naturally leads to the point where developers require means for **ensuring** properties, e.g. the safety or security properties in avionic systems. Unfortunately there are no practical tools implemented for such verification tasks.

This paper introduces an approach which provides software engineers with fully automatic verification. Starting from a first system description of requirements the developer describes the dynamic behavior of the system in statecharts and formulates the requirements either as statecharts or as so-called *temporal properties*. Afterwards he can check automatically whether the system satisfies a requirement or not. Fig 1 shows our framework for translating statecharts and properties into a verification tool.

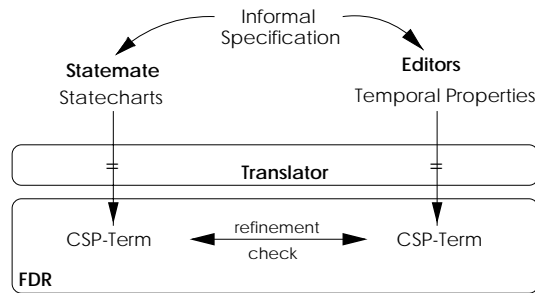


Fig. 1. The Verification Framework

At first we use the timeless, synchronous model of Statechart as underlying formal semantics of statecharts HN95. The translation is defined over a subset of the statechart language which represents a fully abstraction from data. Secondly we chose CSP Hoa78, Hoa85 as underlying formal semantics because there exists tool support Sca95, in particular the process-algebraic verification

tool FDR (Failure Divergence Refinement) For97. FDR is a commercial tool which stands under continuous development. The verification task is done by using the implemented refinement calculus of CSP which will be explained later on.

In section 2 we introduce the translation of STATEMATE statecharts into CSP processes in detail. This is the premise for the analysis of statechart properties shown in section 3. We conclude with an overview of the current work and give an idea of a *verification pool* consisting several verification tools for different purposes.

2 Statecharts in CSP

The model of a statechart specification in CSP is an embedding of the syntax and the step semantics of STATEMATE. The model is divided into three separate components:

- **statechart term**
A statechart is represented as a CSP process. Different processes for sequential, parallel, and hierarchical statecharts are specified.
- **environment**
For the representation of the STATEMATE environment two processes are defined for every statechart event which provide (*pre-process*) and receive (*post-process*) events during a step.
- **step semantical process** The semantics of a STATEMATE-step is modeled by a global step process called *global scheduler*.

2.1 Statecharts Terms

The translation of statecharts can be carried out compositionally by restricting the original language, i.e. you can not use interlevel-transitions. Separate parts of the statechart structure are translated separately too. XOR-terms and their parallel as well as hierarchical compositions are introduced gradually. A modification of the well known *mutual exclusion problem* is used to illustrate the translation.

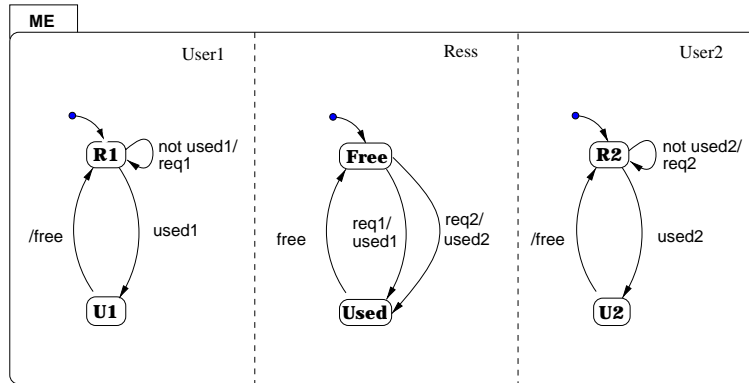


Fig. 2. Mutual Exclusion

Fig. 2 shows such a modification of the Mutual Exclusion. Two users, represented by two XOR-terms *User1* and *User2*, compete for the use of a resource which is represented by the XOR-chart *Ress*. *User1* can request the use of a resource by *req1*, which will block *User2* if *User1* is accepted by *Ress*. This means *User2* can only request *req1* if the resource is clear. Both users will be blocked by waiting for the signal *used*, which excludes a simultaneous use of the resource¹.

¹ We assume that in the initial status no external events are active.

Basic Term (XOR) A XOR-Statechart S is translated into a CSP process which is parameterized by a *label* and has the same name as the statechart. The parameter represents the current state. The modeling of the transitions is realized by IF-THEN-ELSE-constructions in which action labels are communicated. Accordingly, a state (e.g. *Free*) is modeled as a case of the constructions, together with its outgoing transitions (*if Label == Free ...*). A state transition (e.g. from *Free* to *Used*) is represented in such a way that after a guard (e.g. $req1 \vee req2$) has been evaluated, and the event *used* of the action part of the label has been sent, the CSP process is activated with a new parameter *Used* representing the target state of the transition. Furthermore, the event *post_in_Used* was activated for saving the new statechart state and a signal *ackn_Ress* is sent which informs the global scheduler about the execution of a transition in the XOR-chart *Ress*. The functionality of the step process will be explained later on.

A statechart event is modeled in CSP as a communication channel which holds value 0 (not set) or 1 (set). When interrogating events, the channel value is allocated to a local variable (*used?x*) and the conditional part will be logically evaluated. If the evaluation results in TRUE the transition is used, i.e. the transition actions will be executed. If the conditional parts of all outgoing transitions from a status are not fulfilled, the statechart process remains in its current state, i.e. the process is called by the parameter representing the current state.

The sending events in the action part of a label are realised by CSP events with the prefix *post*. The events are stored by special processes, and globally sent in the next step. In the section *Environment* these event processes will be introduced. The CSP process shown below codes the statechart *Ress* of the mutual exclusion example.

```

Ress (Label, Default) =
  act_Ress →
  - state Free
  if Label == Free then
    req1?x1 → req2?x2 →
    if (x1 == 1) then
      (post_used → post_used1 → post_in_Used → ackn_Ress → Ress(Used,no)
      else
        if (x2 == 1) then
          (post_used → post_used2 → post_in_Used → ackn_Ress → Ress(Used,no)
          else
            - static reaction
            post_in_Ress → ackn_Ress → Ress(Free,no))
    else
      - state Used
      if Label == Used then
        free?x →
        (if x == 1 then post_in_Ress → ackn_Ress → Ress(Free,no)
        else
          - static reaction
          post_in_Used → ackn_Ress → Ress(Used,no))
      else STOP

```

Parallel Composition (AND) The simulation or static analysis of a parallel system on sequential machines is based on the sequentialisation of the parallel structures, i.e. one introduces nondeterminism by sequential structures². Every possible combination in regard to the order of operations has to be potentially considered. According to the referred STATEMATE semantics of statecharts lacking data, we will obtain an Interleaving semantics! This means, that within one statechart step the order of execution of parallel statecharts is not fixed, because all actions are temporarily stored in an environment. Reading and writing accesses take place in separate domains.

² This refers to the well-known state explosion problem.

This property can be used by representing the parallel composition of statecharts in a **sequential process** in CSP. An optional but fixed order of execution is set which will not restrict the visible behaviour of a statechart, i.e. the semantics remains unchanged. The example of mutual exclusion has been modeled as a parallel statechart. The following process shows a possible translation.

```

ME =
  act_root → act_ME
  act_User1 → act_User2 → act_Ress →
  ackn_User1 → ackn_User2 → ackn_Ress →
  ackn_ME → ackn_root → ME

```

The process *ME* is activated by *act_ME*, followed by the separate parallel processes *User1*, *User2*, and *Ress*. As we have seen in the modeling of XOR-Terms, every basic component sends an acknowledgement to its parent statechart. After all acknowledgements have been received, the "parallel" process *ME* sends its acknowledgement to the global scheduler.

Hierarchical Composition (XOR-Refinement) The process algebra CSP permits the description of an implicit order by its hierarchy of activation. In this way systems can be built and analysed modularly. A hierarchical statechart is transferred into a hierarchical CSP process where, first of all, the transitions of the parent status are examined. If none of them can be activated, the states of the statechart lying beneath will be considered. This procedure is executed recursively for every further hierarchical level.

Consider an extension of our mutual exclusion example where the state *Used* will be refined by a statechart. In the state *Used* the resource changes from the initial status *Used1* by reactivation into the *Used2* state. The state can only be left by leaving the state *Used* via the outer transition which has priority.

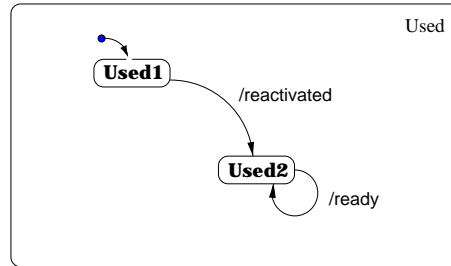


Fig. 3. Refinement of *Used*

The process *Ress* will be extended by the process of refinement in the case *Label == Used*. If no transition can be activated in this case the process lying one hierarchical level below will be activated by *act_Used*. Afterwards, the process *Ress* waits for an acknowledgement *ackn_Used* of its "child" process. The changed process *Ress* has the following form:

```

Ress (Label, Default) =
  act_Ress →
  - state Free
  if Label == Free then
    ...
  else
  - state Used
  if Label == Used then

```

```

    free?x →
    (if x == 1 then post_in_Free → ackn_Ress → Ress(Free,no)
    else
    - activating the refinement
    act_Used → ackn_Used → post_in_Used → ackn_Ress → Ress(Used,no))
else STOP

```

The statechart *Used* itself is an ordinary XOR-statechart and is coded in CSP as follows:

```

Used (Label, Default) =
  act_Used →
  - state Used1
  if Label == Used1 then
    post_reactivated → post_in_Used2 → ackn_Used → Used(Used2,no)
  else
  if Label == Used2 then
    post_ready → post_in_Used1 → ackn_Used → Used(Used1,no)
  else STOP

```

2.2 The Environment of STATEMATE

As already mentioned there is a process, according to the simulation in STATEMATE, which controls and navigates the communication with the environment. This means that all events that are generated during a simulation step are globally stored and made available for the next step.

For the communication of events additional, so-called *environment processes* are introduced which model the STATEMATE environment. Here, two processes are introduced for every event e : the *pre-process*, which makes the set event available for the statechart process, and the *post-process*, which stores the value of the event e during the step (post condition). After a step has been completed the post-processes create the environment for the next step, i.e. the information is copied into the corresponding pre-processes. The functionality of the event processes is represented as extended, finite state automaton. Fig. 4 shows a general pre-process, where e can stand for any statechart event. We will mainly concentrate on the status 0 and 1, where the event is either not set or set.

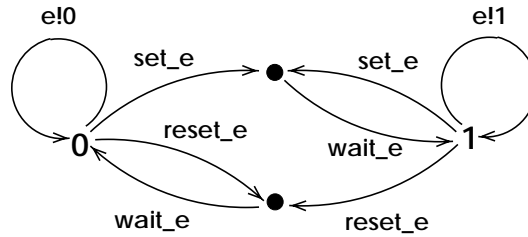


Fig. 4. The Pre Process for event e

The pre-process communicates with the post-process by *set_e* and *reset_e*, with the statechart process by e , and with the step process by the signal *wait_e*. During a step the pre-process makes an event available by the value 1 on the communication channel (event $e!1$), if it has been activated in the previous step. The value of the channel will be 0 (event $e!0$) if it has not been activated. The statechart process communicates those events with the pre-process that have to be considered as conditions of transitions. Fig. 5 shows the post-process which is also represented as extended, finite state automaton. The status 0 and 1 are interpreted as presence or absence of an event in the current environment. The post-process communicates with the pre-process via the events *set_e* and *reset_e*, with the statechart process via *post_e*, with the step process via the event *start_e*.

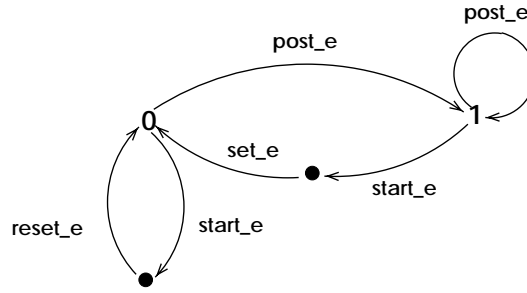


Fig. 5. The Post Process for event e

Within one step the post-process receives, by actions in the form $(post_...)$ in the statechart process, the information about whether a certain event has been set. Since a statechart event can be sent by parallel components during one step, the post-process can communicate an event as often as desired. After a step is completed in the statechart process, the step process initiates (by means of events in the form $start_...$) the copying of current event values from the post to the pre-process. In this way generated events are made available.

Since the events $(wait_e)$ communicate with every event process, the step process recognizes when a copy process from post- to pre-processes has been completed. Only then it starts a new step by recursive activation. The following shows the process pair pre / post for an event e .

$Pre_e(X) =$
 – sending event to statechart process
 $e!X \rightarrow Pre_e(X)$
 – communication with post process: event was not generated
 $\square \text{ reset_e} \rightarrow wait_e \rightarrow Pre_e(0)$
 – communication with post process: event was generated
 $\square \text{ set_e} \rightarrow wait_e \rightarrow Pre_e(1)$

$Post_e(X) =$
 if $X == \text{unset}$ then
 – communication with statechart process
 $post_e \rightarrow Post_e(\text{set})$
 – communication with pre process: event is not set
 $\square \text{ start_e} \rightarrow reset_e \rightarrow Post_e(\text{unset})$
 else
 – communication with statechart process
 $post_e \rightarrow Post_e(\text{set})$
 – communication with pre process: event is set
 $\square \text{ start_e} \rightarrow set_e \rightarrow Post_e(\text{unset})$

2.3 Step Semantics

The step semantics is implemented by the Global Scheduler (GS). The previous sections have in principle prepared the navigation of a step via events. The step process is therefore, as such, very simple and basically consists of two phases:

In the first phase the statechart process is activated by the event act_Root . This process sends the signal $ackn_Root$ after the execution of selectable transitions relating to the environment. The visible part of a step is restricted syntactically by this event, i.e., act- and ackn-signals do not belong to the visible behaviour of the statechart. The second phase simulates the non-visible operations of STATEMATE for the reworking and preparation of steps. In this phase the re-copying of the post-condition of the step (generated events) into the pre-condition for the next step of every

environment process is started by *start_event*. After having received all receipts (*wait_event*) of the environment processes, the next step will be once again activated by the GS.

$$\begin{aligned} \text{GS} = & \text{act_Root} \rightarrow \text{ackn_Root} \rightarrow \\ & \text{start_event1} \rightarrow \text{start_event2} \rightarrow \text{start_...} \rightarrow \\ & \text{wait_event1} \rightarrow \text{wait_event2} \rightarrow \text{wait_...} \rightarrow \\ & \text{GS} \end{aligned}$$

2.4 Overall System

The functionality of the CSP Model System in reference to the communication is shown in Fig. 6 as Message Sequence Chart (MSC). This MSC shows the concrete sequence of events for exactly one step in the CSP model.

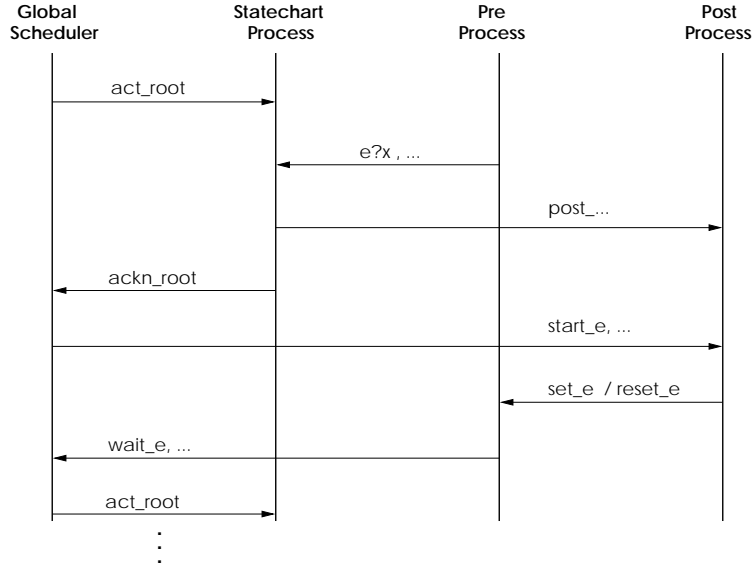


Fig. 6. Step as Message Sequence Chart

The statechart process interacts with the environment by receiving and sending events. It is controlled by the Global Scheduler, i.e. the separate actions are combined into steps by this communication. The environment process consists of the amount of all post- and pre-processes. Every statechart event is represented by the two CSP processes pre and post. The post- and the pre-processes are responsible for receiving and sending events respectively. The copy process which updates the environment for the next step is controlled by the Global Scheduler. The global step process which models the step semantics of STATEMATE controls the overall process of the CSP model. It starts the step with the signal *act_Root*. Afterwards, by using the events made available by the environment, the statechart processes are executed successively, and generated events are sent to the environment.

Next, the statechart process sends the signal *ackn_Root*, whereby the step process initialises the copy process of the post-status into the pre-status of the following step: a new step can start.

Environment The environment comprises all the environment processes Pre and Post of the separate statechart events. Since the environment processes are to act independently of each other they are composed by means of interleaving:

$$\text{Pre} = \text{Pre_e1} \parallel \text{Pre_e2} \parallel \dots$$

$$\text{Post} = \text{Post_e1} \parallel \text{Post_e2} \parallel \dots$$

The whole process environment is defined by the parallel composition of the environment processes Pre and Post. Here, the synchronization of the corresponding process pairs is coded by the events *set_e* and *reset_e* in a combined alphabet of communication:

$$\text{alpha_Environment} = \text{---set_e1, reset_e1, set_e2, reset_e2, ...---}$$

$$\text{Environment} = \text{Pre} [\text{---alpha_Environment---}] \text{Post}$$

The Statechart Process Since the separate statechart processes do not communicate directly with one another, they are integrated into one SC process by interleaving. The parallel composition of the processes SC and Environment are represented by the process SC_Environment. The synchronisation takes place via events *e* which are made available by the environment, and generated events *post_e* which are sent to the environment.

$$\text{SC} = \text{SC1}(\dots) \parallel \text{SC2}(\dots) \parallel \dots$$

$$\text{alpha_SC_Environment} = \{|e1, \text{post_e1}, e2, \text{post_e2}, \dots|\}$$

$$\text{SC_Environment} = \text{SC} [| \text{alpha_SC_Environment} |] \text{Environment}$$

The Overall System The system process, finally, combines the environment and the statechart (process SC_Environment) with the controlling step process Global Scheduler (GS). They communicate via the events *act_Root* and *ackn_Root* with which the start and the completion of steps are associated, as well as via events *start_e* and *wait_e* which mark the start and completion of the updating of the environment:

$$\text{alpha_System} = \{| \text{activate_Root}, \text{ackn_Root}, \text{start_e1}, \text{wait_e1}, \text{start_e2}, \text{wait_e2}, \dots | \}$$

$$\text{System} = \text{GS} [| \text{alpha_System} |] \text{SC_Environment}$$

Fig. 7 shows the general communication structure of the CSP Model which consists of the components statechart, environment, and the step process Global Scheduler.

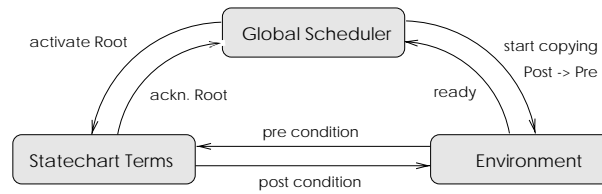


Fig. 7. Communication of the CSP Model

3 Analysis of Properties

For the verification of STATEMATE models we want to use the CSP-based modelchecker FDR. For this reason we presented the translation of statecharts into CSP processes in chapter 2. The aim of our approach is to verify behavioral properties of STATEMATE models. In the next step we have

to define a representation for the considered properties.

Mostly one wants to check safety or security properties, i.e. which hold for the whole life time of the system. As already mentioned, we use CSP as description language for the formal semantics of statecharts and temporal properties. A property will be verified against the system by trace refinement of the corresponding CSP processes. The underlying verification technique of the used tool FDR is strongly based on the refinement calculus of CSP. What this verification technique actually does is to solve the classical *language inclusion problem* which belongs to **modelchecking**. Hence there are two possible ways of verification w.r.t. the representation of properties (shown in figure 8):

- *Explicit* representation as temporal logic formula (e.g. linear temporal logics Emm90),
- *Implicit* representation as state-transition system (e.g. statecharts).

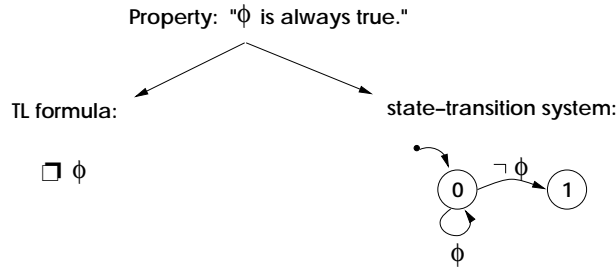


Fig. 8. Representation of requirements

In the first case the semantics of a temporal logics formula is implemented as a CSP process, the second case means trivially using the existing translation. In both ways we check the trace refinement between two CSP processes, one is the state model and another is the property.

In order to satisfy the first case we use a linear temporal logics (LTL). Accordingly, we have developed a library of useful properties with their semantics as CSP processes.

For showing an example of an analysis driven by the presented approach we again use the statechart specification of the mutual exclusion problem (Fig. 2). The obvious requirement is: the resource must never be used by both users at the same time. W.r.t. the specification we can write this property as an LTL formula.

$$\Box \neg (\text{in}(U1) \wedge \text{in}(U2))$$

Now we translate this formula into a CSP process. The process *Prop* codes the introduced property.

```

Prop =
  act_Root → in_U1?x1 → in_U2?x2 →
    if x1==1 and x2==1 then
      STOP
    else
      ackn_Root → Prop

```

The general translation of LTL formulas into CSP processes w.r.t. the introduced CSP model is a complex task. We are currently working on a library of properties that are often used in practical application.

Now the task is to check the trace refinement relation between the property *Prop* and the CSP model of the mutual exclusion problem specification *SYSTEM*. For that reason it is necessary to hide all the (internal) communication of *SYSTEM* which is not part of the alphabet of the property process *Prop*.

$$\text{Prop} \sqsubseteq_T \text{SYSTEM} \setminus \alpha(\text{System}) - \alpha(\text{Prop})$$

The used modelchecker FDR executes the refinement check of the two models by bisimulation and returns: *TRUE* (see fig 9). The CSP model *SYSTEM* meets the Property *Prop*.

```

Refine checked 228 states
With 538 transitions
Stopped timer
Resource   Start           End           Elapsed
Wall time   536199         536199         0
CPU (self)    0             0             0
CPU (sys)     0             0             0
(inc children)
CPU (self)    3             3             0
CPU (sys)     0             0             0
true

```

Fig. 9. Analysis Result from FDR

3.1 Case Study Cruise Control System

A practical case study is the automotive application cruise control system CL96. The user interface is a lever with four different positions (see fig 10)

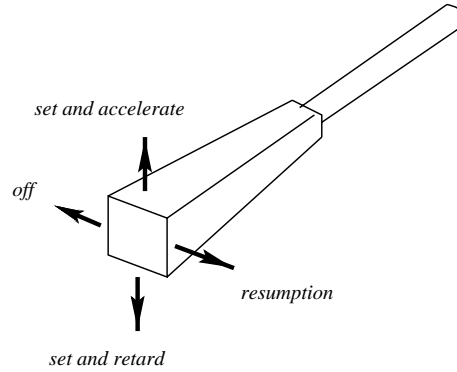


Fig. 10. lever of the cruise control system

Every position covers one of the following functions.

- *set and accelerate*
- *set and retard*
- *off*
- *resumption*

The cruise control will be activated by moving the lever into the positions *set and accelerate* or *set and retard* after reaching the desired velocity. From this point the actual velocity will be held constantly. The system saves the actual speed. The actual velocity will be increased (decreased) by moving the lever into the position *set and accelerate* (*set and retard*). The driver deactivates the cruise control by moving the lever into the *off* position or pushing the brake.

The following statechart represents the described system.

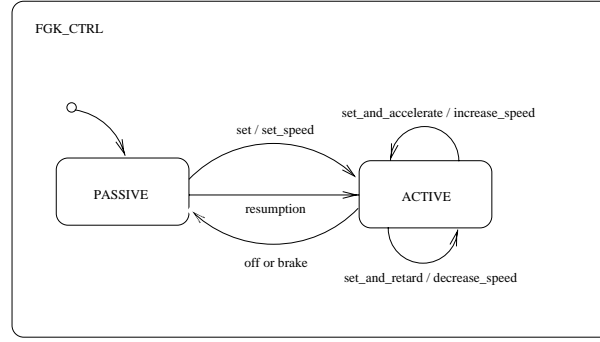


Fig. 11. statechart of the cruise control system

Following the introduced translation rules in chapter 2 we formulate the corresponding CSP process *FGK_CTRL*.

```

FGK_CTRL(Label) =
  -- State PASSIVE
  if Label == PASSIVE then inPASSIVE ->
    set?v1 -> resumption?v2 ->
    (if (v1 == 1) then
      set_speed -> post_in(ACTIVE) -> ackn_FGK_CTRL ->
      FGK_CTRL(ACTIVE)
    else
      if (v2 == 1) then
        post_in(ACTIVE) -> ackn_FGK_CTRL -> FGK_CTRL(ACTIVE)
      else
        if (v1 == 1) and (v2 == 0) then
          post_in(PASSIVE) -> ackn_FGK_CTRL -> FGK_CTRL(PASSIVE)
        else
          -- static reaction
          post_in(PASSIVE) -> ackn_FGK_CTRL -> FGK_CTRL(PASSIVE))
    else
  -- State ACTIVE
  if Label == ACTIVE then inACTIVE ->
    off?v1 -> brake?v2 -> setAcc?v3 -> setRet?v4 ->
    (if (v1 == 1 or v2 == 1) then
      post_in(PASSIVE) -> ackn_FGK_CTRL -> FGK_CTRL(PASSIVE)
    else
      if (v2 == 1) then
        set_speed_acc -> post_in(ACTIVE) -> ackn_FGK_CTRL ->
        FGK_CTRL(ACTIVE)
      else
        if (v3 == 1) then
          set_speed_ret -> post_in(ACTIVE) -> ackn_FGK_CTRL ->
          FGK_CTRL(ACTIVE)
        else
          -- static reaction
          post_in(ACTIVE) -> ackn_FGK_CTRL -> FGK_CTRL(ACTIVE))
    else STOP
  
```

We want to check the following safety property of the cruise control system: if the driver pushes the brake then afterwards the system is passive. By considering the statechart we can say: whenever the event *BRAKE* is active then the system must change to state *PASSIVE* in the following step. This property can be formulated as a LTL-formula.

$$\square (\text{BRAKE}=\text{TRUE} \Rightarrow \circ \text{in(PASSIVE)})$$

This formula can be written as a CSP process.

```

Prop = activate_FGK_CTRL ->
      -- brake was pushed
      (brake.1 -> ackn_FGK_CTRL ->
        -- next step in state PASSIVE
        activate_FGK_CTRL ->
        inPASSIVE -> ackn_FGK_CTRL -> Prop)
    []
      -- brake was not pushed
      (brake.0 -> ackn_FGK_CTRL -> Prop)

```

Now the task is to check the refinement relation between the system model FGK_CTRL and the introduced property Prop.

Prop \sqsubseteq_T FGK_CTRL

This is done by the used model-checker FDR (see fig 12).

```

Refine checked 3478 states
With 6079 transitions
Stopped timer
Resource   Start           End           Elapsed
Wall time   537203         537204         0
CPU (self)    10           11           0
CPU (sys)     0            0           0
(inc children)
CPU (self)     8            8           0
CPU (sys)     0            0           0
true

```

Fig. 12. Analysis Result from FDR

4 Related and Future Work

We have presented an alternative method for verifying STATEMATE-statecharts by an interpretation of statecharts as a CSP program. This approach includes the translation of statecharts as well as the representation of certain properties to verify them. This framework was prototypically implemented as additional tool for STATEMATE.

Current work. One of the main questions in an industrial context is the integration of a new method or technique in existing development processes. As next step in the application of our approach we want to give an idea of how to connect existing *testing* activities with verification tasks. Our approach is useful for supporting software- and system tests as well, e.g.

- 1.) proving *test assumptions* (deadlock-freedom, determinism etc.) and
- 2.) computing the set of traces to accept / to refuse by the system systematically.

A testing framework for the second case could be constructed as follows:

- Calculate validation model of the specification
(described translation of statecharts and requirements)
- Calculate the set of traces to accept / to refuse by the specification
(described modelchecking resulting in a set of *counterexamples*)
- Specify and Implement test signals (test instrumentation)

- Execute the implementation (by the calculated traces)
- Evaluate tests by checking accepted traces

Future work. Furthermore one can imagine that for different application fields one requires different representations of properties, e.g. as timing diagrams or message sequence charts. Therefore it is useful to have a *verification pool* consisting of several verification tools with different representations (fig. 13). The translation of statecharts will be adapted to the current purpose, e.g. for considering timing aspects or asynchronous semantics of statecharts you need to use another tool which can handle time or asynchronism.

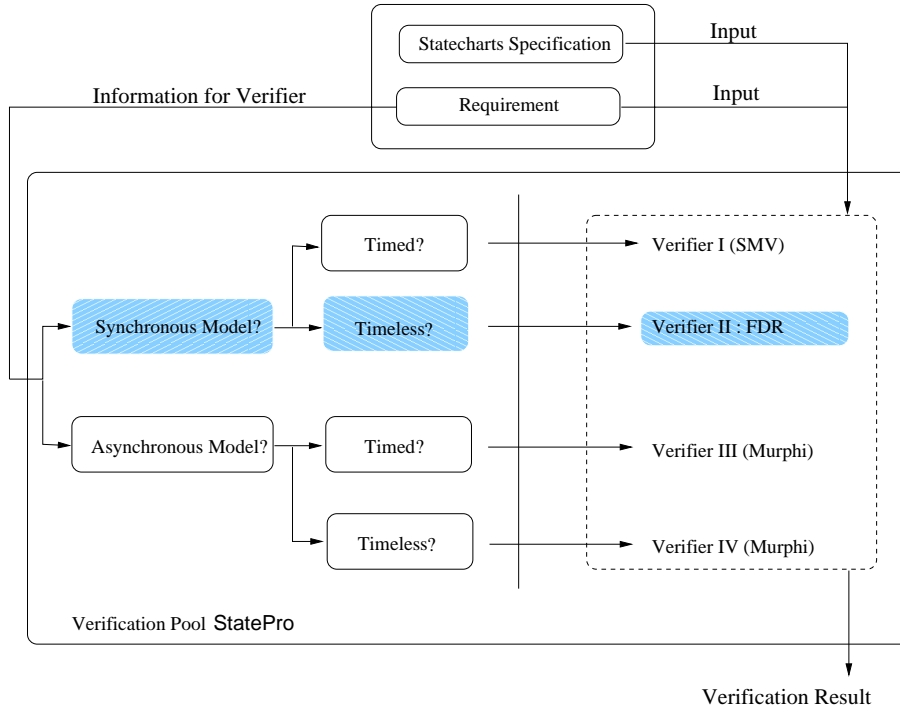


Fig. 13. StatePro as Verification Pool

References

- M. Conrad and E. Lehmann. *Anforderungsspezifikation des Fahrgeschwindigkeitskonstanters mit Z und Statemate*. Technischer Bericht Nr. F3S/K-96-03, Daimler Benz AG, Berlin, 1996.
- E. A. Emmerson. Temporal and modal logic. *Handbook of Theoretical Computer Science MIT Press*, 1990.
- Formal Systems (Europe) Ltd. *Failures Divergence Refinement, FDR2 User Manual*, 1997.
- D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- David Harel, Hagi Lachover, Amnon Naamad, Amir Pnueli, Michal Politi, Rivi Sherman, Aharon Shtull-Trauring, and Mark Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16 No. 4, April 1990.
- David Harel and Amnon Naamad. The statemate semantics of statecharts. *Technical Report, i-Logix*, October 1995.
- C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM, Number 8*, 21:666–667, 1978.
- C.A.R. Hoare. *CSP - Communicating Sequential Processes*. Prentice Hall International, 1985.
- J.B. Scattergood. *Tools for CSP and Timed CSP, D.Phil.* Oxford University Computing Laboratory, 1995.