Run Time Monitoring of Reactive System Models

Mikhail Auguston
Naval Postgraduate School
Monterey, CA, USA
auguston@cs.nps.navy.mil

Mark Trakhtenbrot
Academic Institute of Technology
Holon, Israel
ilmarktr@yahoo.com

Abstract

In model-based development of reactive systems, statecharts are widely used for formal design of system behavior, and provide a sound basis for analysis and verification tools, as well as for code generation from system models. We present an approach for dynamic analysis of reactive systems via run-time verification of code produced with Statemate C and MicroC code generators [10], [15]. The core of the approach is automatic creation of monitoring statecharts from formulas that specify the system's behavioral properties in a proposed assertion language. Such monitors are then translated into code together with the system model, and executed concurrently with the system code. This approach leads to a more realistic analysis of reactive systems, as monitoring is supported in the system's actual operating environment. For models that include design-level attributes (division into tasks, etc.), this is crucial for performance-related checks, and helps to overcome restrictions inherent in simulation and model checking.

1. Introduction

Development of reliable reactive systems is a significant challenge, especially due to their complex behavior. There has been a great deal of research on the development of formal methods for specification, design, analysis and verification of reactive systems.

For precise specification of system behavioral properties, various types of temporal logic are widely used. These include LTL [14], which offers special temporal operators for reasoning about past and future properties of behavioral sequences, and MTL [5], which supports expression of real-time constraints through definition of duration for future temporal operators. Some specification formalisms suggest various kinds of syntax sugar that make the specification task more user friendly for designers who are not logicians. For example, with the LA language in [18], temporal properties look like a combination of stylized English with C-like expressions. In [3], the temporal logic details are hidden "behind the scenes", and instead, patterns are used that allow to specify common properties (such as existence, absence, response, precedence, etc.) and scope in which the property should hold. This approach is used, for example, in a Statemate verification tool called ModelCertifier [16] that offers a rich library of pre-defined property patterns, where each pattern looks like a parameterized natural language sentence. Paper [6] introduces a language for pattern definition as a way to create extendable sets of property patterns. Sugar [19] provides several layers for property specification and verification; in particular, extended regular expressions are used to describe execution sequences on which temporal properties are checked.

On the other hand, model-based system development has become the way to design, implement and validate reactive systems. Statecharts, first introduced in [9], have become a standard for behavior design in popular model-based methodologies such as structured and object-oriented design [7]. Various tools (e.g., Statemate [10], Rhapsody [9], BetterState [20]) support the creation of executable models using statecharts, and their analysis through simulation, execution of automatically generated code, and, in Statemate, verification. Ongoing research on model-based testing covers, among other issues, test generation from statechart models [4].

One powerful method of dynamic analysis is run-time monitoring of system execution. A number of tools have been developed for monitoring various types of programs (including real-time systems), see, for example [1], [2], [18]. The relevant assertion languages allow for expressing a wide range of properties in terms of events that occur in the running code, and for defining tool reactions when a violation is found or when the run was successful. An important problem here is the gap between the system specification, which usually refers to high-level objects, and monitors, which refer to implementation-level events (such as function calls, etc.). Some issues related to derivation of monitors from system specification are considered in [17].

Model-based development leads to a narrowing of this gap, as monitoring can be performed on the model (rather

than the implementation) level. Statemate [10] supports the use of the so-called watchdog (testbench) statechart. Such a chart is not part of the system model; its role is either that of a driver (acting as an environment and producing system inputs) or a monitor (watching the system for proper behavior or abnormalities). To perform its role, the watchdog is executed in parallel with the model. Violation of the monitored property can be expressed and observed as entering an error state in the monitor chart. For example, Fig. 1 shows a simple statechart for monitoring the following requirement: "Processing of a request must be accomplished within 5 seconds, and before receiving the next request".

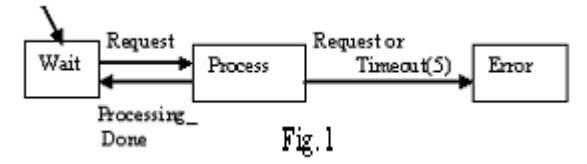


Fig. 1

An important feature of monitor statecharts is that they have access to all elements in the system model. In other words, visibility from the monitor is supported both for observable elements (events, conditions and data items) that belong to system's interface with the environment, and for internal elements such as states or events used for internal communication between system components. This allows for both black box and more detailed white box monitoring, and makes localization of design problems easier.

2. What is in this paper

This paper presents an approach to dynamic analysis of reactive systems modeled with statecharts using Statemate. The basic goal here is to reveal errors (rather than to validate or show correctness).

The analysis is based on run-time monitoring of code generated from the system model. The code is checked against the system specification describing the required and forbidden behaviors; these are expressed in a proposed assertion language described below. The main idea underlying this approach is the automatic creation of monitors directly from the system specification. This is achieved through translation of the specification into an equivalent watchdog statechart(s). This step is followed by generating code from the system model and from the created monitor (using the existing Statemate C code generator), and their simultaneous execution. Appropriate diagnostics is produced during the execution and/or upon its completion.

The suggested approach has a number of advantages, and is especially helpful in situations where the use of other analysis tools (e.g. of model checkers such as Statemate ModelCertifier [16]) becomes problematic:

- There is no restriction on the size of the tested model, and execution of compiled code (for model and monitor) is fast. On the other hand, model checking may become slow for very large real-world models.

- Generated code for the system and its monitor is executed in ***real time***. Even though such code is usually considered prototype quality, it is fast enough and allows for meaningful checks of time constraints (unless they are tighter than the code performance). Such checks are beyond the scope of simulation and model checking tools that are based on simulated time schemes described in [12]: synchronous (for clock-driven systems) and asynchronous (for event-driven systems). In the synchronous scheme duration of all steps is the same, regardless of how "heavy" the executed actions are. In the asynchronous scheme, steps take zero time, and the system executes a chain of steps until stabilization; only then is the clock advanced and inputs accepted. These abstractions assume that the system is fast enough to complete its reactions to external stimuli before the next stimulus arrives. Real time monitoring allows one to check whether this assumption is correct.

- Our approach allows monitoring of code generated from the Statemate model augmented by design attributes (showing the system division into tasks of various types, mapping model elements into events of the target RTOS, etc.). For such models, the MicroC code generator [15] automatically creates a highly optimized production quality code for the OSEK operating system, widely used in the automotive industry for embedded microcontroller development. Thus the code can be executed and monitored in its realistic hardware-in-the-loop operating environment. This kind of analysis is impossible with model checking.

- Model checking requires that all data be properly restricted to guarantee that a finite state model is analyzed. This requirement is problematic for input data, if there is not enough information about the system environment. No such restrictions are relevant for monitoring, and moreover, monitored code derived from the system model can be connected to real sources of input data.

3. Assertion language

To specify and monitor real-time properties of reactive systems, we use an assertion language that integrates a number of powerful features found in temporal logic and in FORMAN language introduced in [1], [2], and used in a number of tools:

- Boolean expressions can refer to any elements in the system model, and express properties of system configurations. For example: *in(S) and (x>5)* means that currently the system is in state *S* and *x* is greater than 5.

- Regular expressions allow for description of state sequences. Consider for example, the expression:

[SELECT (Open | Read | Write | Close) FROM ex_program] SATISFY Open (Read | Write) Close*

This assertion requires to select execution trace states matching one of the given patterns, and to check the sequence of selected states for conformance with the regular expression.

- Temporal formulas express order properties fulfilled by system execution sequences. They are built using unrestricted future temporal operators *NEXT, ALWAYS, EVENTUALLY, UNTIL* and their past counterparts: *PREVIOUS, ALWAYS_WAS, SOMETIME_WAS, SINCE*. Following [14], we consider formulas for the following types of properties (where *P* is a past formula):

Safety: *ALWAYS (P)*
Guarantee: *EVENTUALLY (P)*
Obligation: Boolean combination of safety and guarantee
Response: *ALWAYS (EVENTUALLY(P))*
Persistence: *EVENTUALLY (ALWAYS(P))*
Reactivity: Boolean combination of response and persistence.

According to [14], any temporal formula is equivalent to a reactivity formula; the other five types of formulas are allowed for more flexibility. For convenient expression of real-time constraints, we support also a restricted version of the above operators; it is obtained by attaching appropriate time characteristics. For example, *ALWAYS(10)P* means that *P* is continuously true during 10 time units after the current moment, while *SOMETIME_WAS (10) P* denotes that *P* was true at least once in the 10 previous time moments. With this extension, *P* in the above formulas is now allowed to be a restricted (future or past) formula. Note that in our version of assertion language an unrestricted temporal operator can not be nested within a restricted one.

- Actions define what should be done when a property violation is found, or when the property holds for the checked run. Typically, this includes sending an appropriate message. In general, any user-defined functions can be used here to provide a meaningful report that may include, for example, interesting statistics and other profiling information (frequency of occurrence for certain event, total time spent by the system in certain state, etc.). For this, actions can use the appropriate attributes of the referred objects (e.g., the time at which a certain interval was entered).

The examples in section 4 illustrate the use of this assertion language. Since the language is based on constructs described elsewhere (see [14], [12] and [1]), detailed description of its syntax and semantics is omitted from this paper. Nevertheless, one delicate issue should be mentioned here. System specification usually assumes infinite execution sequences (as a reactive system has an ongoing interaction with its environment). Correspondingly, the traditional semantics of temporal operators is also defined for infinite execution sequences. However, monitoring usually deals with finite (truncated) runs, and this requires a proper definition of the semantics for cases when there is doubt as to what would have been the property formula value if the execution had not been stopped. Paper [7] studies several ways of reasoning with temporal logic on truncated executions. We follow the neutral view of [7], illustrated by the following example. Consider the following assertions:

ALWAYS (P → EVENTUALLY (10) Q)
ALWAYS (P → ALWAYS (10) Q)

and suppose that the run is completed (truncated) 4 seconds after the last occurrence of event *P* (we assume that each of the properties held for all earlier occurrences of *P*). If there was no *Q* after the last *P*, then the first assertion is considered to be false for this run (even though continuation of the run could reveal that *Q* does occur in 10 seconds after *P*, as required). On the contrary, if *Q* held continuously after the last *P* and until the end of the run, then the second assertion is considered to be true. In general, it is the user's responsibility to make the on-satisfy and on-failure actions detailed enough, so that he can better understand the monitoring results (e.g. whether a real violation was found, or it is in doubt due to the state at which the execution was truncated).

4. Examples

To illustrate our approach, we consider the Early Warning System (EWS) example from [12]. We present its verbal description followed by the statechart presenting the behavioral design of the system. We then give examples of assertions and, for one of them, show its translation into a monitor statechart according to our translation scheme.

The EWS receives a signal from an external source. When the sensor is connected, the EWS performs signal sampling every 5 seconds; it processes the sampled signal and checks whether the resulting value is within a specified range. If the value is out of range, the system issues a warning message on the operator display. If the operator does not respond to this warning within a given time interval (15 seconds), the system prints a fault message and stops monitoring the signal. The range limits are set by the operator. The system is ready to start monitoring the signal only after the range limits are set. The limits can be redefined after an out-of-range situation has been detected, or after the operator has deliberately stopped the monitoring.

Fig. 2 shows a statechart describing the EWS, similar to the one in [12]. The main part of EWS behavior is detailed in state *ON*. It contains two *AND*-components that

represent the EWS controller and the sensor acting concurrently. Events *DO_SET_UP, EXECUTE,* and *RESET* represent the commands that can be issued by the operator. Timing requirements are represented by delays that trigger the corresponding transitions. The *AND*-components can communicate; e.g., see event *CONNECT_OFF* sent from the controller component to the sensor component.

Following are four examples of assertions that reflect some of the above requirements for EWS:

1) *ALWAYS (EXECUTE → SOMETIME_WAS (DO_SET_UP))*

(monitoring of signal should be preceded by setting range limits)

2) *ALWAYS (OUT_OF_RANGE →*

EVENTUALLY (15) (RESET or started(PRINT_ALARM))

(in the out-of-range situation, within 15 seconds either the operator responds or a fault message is printed)

3) *ALWAYS (*

ALWAYS_WAS (15) (in(DISPLAY_ALARM) & not RESET)

→ started(PRINT_ALARM))

(a similar property, this time expressed using the past temporal operator)

4) *ALWAYS (FINISHED_SAMPLING →*

ALWAYS (5) in(IDLE) or EVENTUALLY(5)CONNECT_OFF)

(after signal sampling is finished, there is a 5-second pause before the next sampling, unless the sensor is disconnected)

Note that the first assertion is violated for the given statechart; this happens in the following scenario: *POWER_ON; CONNECT_ON; EXECUTE*. The other assertions are valid as long as the system remains in its *ON* state (i.e., *POWER_OFF* doesn't occur), but otherwise can be violated.

Fig. 3 shows how the second of these four assertions is translated into a monitor statechart. Suppose *POWER_OFF* occurs 7 seconds after *OUT_OF_RANGE*, and there was no *RESET* in this interval. If the system remains in state *OFF* for at least the following 8 seconds, then the monitor will enter its state *D*, thus indicating a violation of the monitored assertion.

5. Implementation Outline

Statemate Boolean expressions obtained from basic predicates (like *in(DISPLAY_ALARM)*, guarding conditions, and event occurrences are directly visible from monitor statechart; in this sense, their monitoring is trivial. In monitors created to watch temporal and timing properties, such expressions can be used as transition triggers, similar to the example in Fig.1.

In the rest of this section, we present an outline of a translation scheme for restricted and unrestricted temporal formulas allowed by our assertion language (see section 3 above). Though not fully formalized here, the presentation clearly shows the technique used for generation of monitors from assertions.

Let *P, Q, S* denote basic Boolean formulas, which do not contain any temporal operators, and let *FRM* denote any formula.

Then *P → Q* means that *P* is used as a trigger to start monitoring of formula *Q*; for each occurrence of *P*, a new thread of *Q* monitoring should be started. Absence of the trigger (*P→ ...*) means that the start of execution is the only trigger event.

If a formula includes only restricted future temporal operators, like in

FRM ≡ P → TL_Operator (N1) TL_Operator (N2) .... TL_Operator (Nk) S

then its value becomes known after (i.e. it needs to be monitored during), at most, *t(FRM) = N1 + N2 + ... + Nk* time units from the triggering event P. For example:

P→ ALWAYS(5) EVENTUALLY(10) S

is monitored during, at most, 15 time units from the triggering event *P*. For each step within the monitoring interval we have to know the Boolean values of all basic sub-formulas in the *FRM*. This is sufficient to determine, after *t(FRM)* time units, whether *FRM* is true or false for the particular occurrence of the trigger event *P*.

Every restricted future formula is translated into a chart containing two designated states: accepting state *F*, and rejecting state *D*; there are no transitions exiting from *F* and *D* in such a chart. The value of the formula is true when computation ends in *F*, and false when it ends in *D*. If execution of the monitored system is truncated before completion of the formula computation, then (in the spirit of the neutral view as defined in [7]) the value is decided to be true for the *ALWAYS*-formula and false for the *EVENTUALLY*-formula.

As an illustration, Fig. 4 schematically shows the translation pattern for *FRM≡ALWAYS (N) P*, where *P* itself is either a basic or a restricted future formula. Translation is defined by structured induction, starting from the case when *P* is a basic formula. Note that each advance of the clock by one time unit causes a new thread of computation for *P* to be started. Each thread is represented in the chart by a separate *AND*-component; there are *N* such components. This number is known based on an analysis of the translated formula.

Fig. 5 shows a sample translation pattern for the case when the unrestricted operator *ALWAYS* is applied to the restricted formula *P* (the actual structure of state *P* in each thread is defined by translation rules for restricted formulas). In this case, as long as *P* holds the value true, we should continue the ongoing computation of *P*. Whenever the monitor enters its *D* state, the value of the

formula becomes false; otherwise (including the case of truncated execution), the value is true. Note that since obtaining a value of *P* may require up to *t(P)* time units, there are *t(P)* threads computing *P*. When a cycle of *P* computation is completed with the value true (the component reaches its *F* state), it is restarted again. Also note the delays: *RESTART_P_i* is defined in such a way that with each advance of the clock by one time unit, a new cycle of *P* computation is started. Restarting *P* immediately upon its completion in state *F* would have caused a violation of such synchronization in case a certain cycle takes less time than *t(P)*. This, in turn, could lead to wrong computation of the entire formula.

To implement *EVENTUALLY (ALWAYS(P))*, we have to restart computation of *ALWAYS(P)* whenever it gets the value false, i.e., when the chart in Fig. 5 enters state *D* (at the top level of the hierarchy). In other words, such implementation can be obtained by redirecting the transition from *D* back to the *AND*-state.

Implementation of dual formulas (where *ALWAYS* is replaced by *EVENTUALLY* and vice versa) is similar, with appropriate replacement of *F*-states by *D*-states and vice versa.

For restricted past formulas we need to monitor only the finite segment of the execution in order to decide whether the formula is true or false. Consider, for example, *ALWAYS_WAS (N) P* which means "during *N* time units preceding the current moment, *P* was continuously true". Implementation uses a counter *CP* associated with the formula; on each advance of the clock, if *P* is true then *CP* is incremented, and if *P* is false then *CP* is set to 0. Now *ALWAYS_WAS (N) P* is true at the current moment, iff *CP=N*.

Similarly, for *SOMETIME_WAS (N) P* that means "from the current moment in at least *N* previous steps *P* was true at least once", the implementation will use the counter *CP* in the following way: On each advance of the clock, if *P* is true then *CP* is set to *N,* and if *P* is false then *CP* is decremented by 1. Now, *SOMETIME_WAS (N) P* is true at the current moment, iff *CP > 0* at the current moment.

6. Conclusions and future work

The paper presents an approach to dynamic analysis of reactive systems via run-time verification of code generated from Statemate models. The approach is based on the automatic creation of monitoring statecharts from formulas that specify the system's temporal and real-time properties in a proposed assertion language. The promising advantage of this approach is in its ability to analyze realistic models (with attributes reflecting the various design decisions) in the system's realistic environment. This capability is beyond the scope of simulation and model checking tools.

Several experiments have been carried out, that included manual creation of monitor charts from assertion formulas and their use with C code generated from Statemate models (EWS considered in section 4, and some others). This helped in a more accurate definition of the translation scheme.

The natural next step is actual implementation of the translation from the assertion language into statechart monitors, which is the core of the suggested approach, and use of created monitors with real-world system models.

The assertion language needs to be more convenient for designers. A possible way to achieve this is to adopt some of the ideas discussed in [3], [6], [18], [19]. This will require an appropriate adaptation of the translation scheme.

The system described above for statechart run time monitoring is under development. The suggested translation scheme provides a uniform mechanism for automatic creation of monitors, although some examples show that, in certain cases, more compact and optimized monitors can be produced. Further research is needed to define a more efficient translation scheme, both for synchronous and asynchronous time models.

Finally, an interesting challenge is to check a similar approach with a UML-based design paradigm that uses an OO version of statecharts for behavior description. Here an additional advantage could be in monitoring of systems where objects are created dynamically such that their amount is not limited in advance (model checking analysis of such systems is clearly problematic).

7. Acknowledgements

This work has been supported in part by the U.S. Office of Naval Research Grant # N00014-01-1-0746.

8. References

[1] M. Auguston, Program Behavior Model Based on Event Grammar and its Application for Debugging Automation, *2nd Int'l Workshop on Automated and Algorithmic Debugging, AADEBUG'95*, May 1995, pp. 277-291.

[2] M. Auguston, A. Gates, M. Lujan, Defining a Program Behavior Model for Dynamic Analyzers, *9th International Conference on Software Engineering and Knowledge Engineering, SEKE'97*, June 1997, pp. 257-262.

[3] G.S. Avrunin, J. C. Corbett, and M. B. Dwyer, Property Specification Patterns for Finite-State Verification, *2nd Workshop on Formal Methods in Software Practice*, March 1998, pp.7-15.

[4] K.Bogdanov, M.Holcombe, H.Singh. Automated Test Set Generation for Statecharts. In D. Hutter, W. Stephan, P. Traverso and M. Ullmann, editors, *Applied Formal Methods*

- *FM-Trends 98*, LNCS, v.1641, Springer Verlag, 1999, pp. 107-121.

[5] E.S. Chang, Z. Manna, and A. Pnueli. Compositional Verification of Real-time Systems. In *Proceedings of the 9th IEEE Symposium Logic in Computer Science (LICS 1994)*, IEEE Computer Society Press, 1994, pp. 458-465.

[6] J.C. Corbett, M.B. Dwyer, J. Hatcliff, Robby. A Language Framework for Expressing Checkable Properties of Dynamic Software. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, LNCS, v.1885, Springer-Verlag, 2000, p.205-223.

[7] B. P. Douglass, D. Harel and M. Trakhtenbrot. Statecharts in Use: Structured Analysis and Object-Orientation. *Lectures on Embedded Systems* (F. Vaandrager and G. Rozenberg, eds.), LNCS, v.1494, Springer-Verlag, 1998, pp. 368-394.

[8] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, D. Van Campenhout. Reasoning with Temporal Logic on Truncated Paths, In *Proceedings of 15th Computer-Aided Verification conference* (*CAV'O3*), LNCS, v.2725, Springer-Verlag , July 2003, pp.27-39,

[9] E. Gery, D. Harel and E. Palatchi. Rhapsody: A Complete Lifecycle Model-Based Development System, In *Proc. 3rd Int. Conference on Integrated Formal Methods*, IFM 2002, pp.1-10.

[10] D.Harel. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 8, 1987, pp. 231-274.

[11] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems, *IEEE Trans. on Software Engineering* 16:4 (1990), pp.403-414.

[12] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering Method.* 5:4 (1996), pp.293-333.

[13] D.Harel, and M.Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach* McGraw-Hill, 1998

[14] Z.Manna and A.Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1991.

[15] M.Thanne and R.Yerushalmi. Experience with an Advanced Design Flow with OSEK Compliant Code Generation for Automotive ECU's. *Dedicated Systems Magazine*, *Special Issue on Development Methodologies & Tools*, pp. 6-11, 2001

[16] OSC – Embedded Systems AG. *Statemate ModelCertifier.* http://www.osc-es.de/products/en/modelcertifier.php

[17] D.Richardson, S.Leif Aha, T.Owen O'Malley. Specification-based Test Oracles for Reactive Systems, In *Proc. Fourteens Intl. Conf. on Software Engineering*, Melbourne, 1992, pp.105-118.

[18] O.Strichman, R.Goldring. The 'Logic Assurance (LA)' System - A Tool for Testing and Controlling Real-Time Systems, In *Proceedings of the 8th Israeli Conference on Computer Systems and Software Engineering,* 1997, pp.47-56.

[19] I.Beer, S.Ben-David, C.Eisner, D.Fisman, A. Gringauze and Y.Rodeh. The Temporal Logic Sugar. In *Intl. Conference on Computer Aided Verification* (*CAV'01*), LNCS, v.2102, July 2001, pp.363-367.

[20] Wind River Systems, Inc. *BetterState* http://www.windriver.com/products/betterstate/index.html

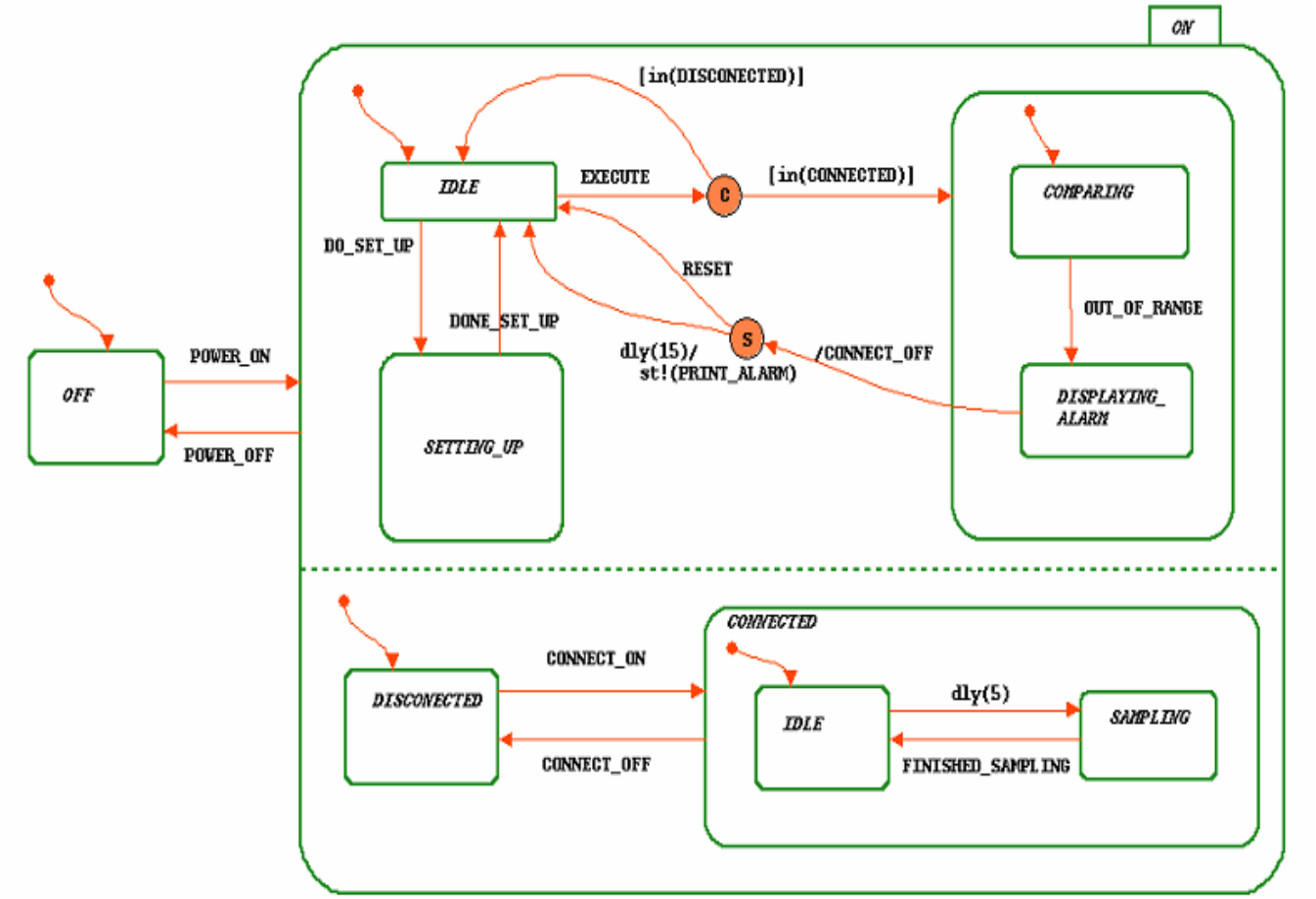


Fig. 2 Statechart for Early Warning System

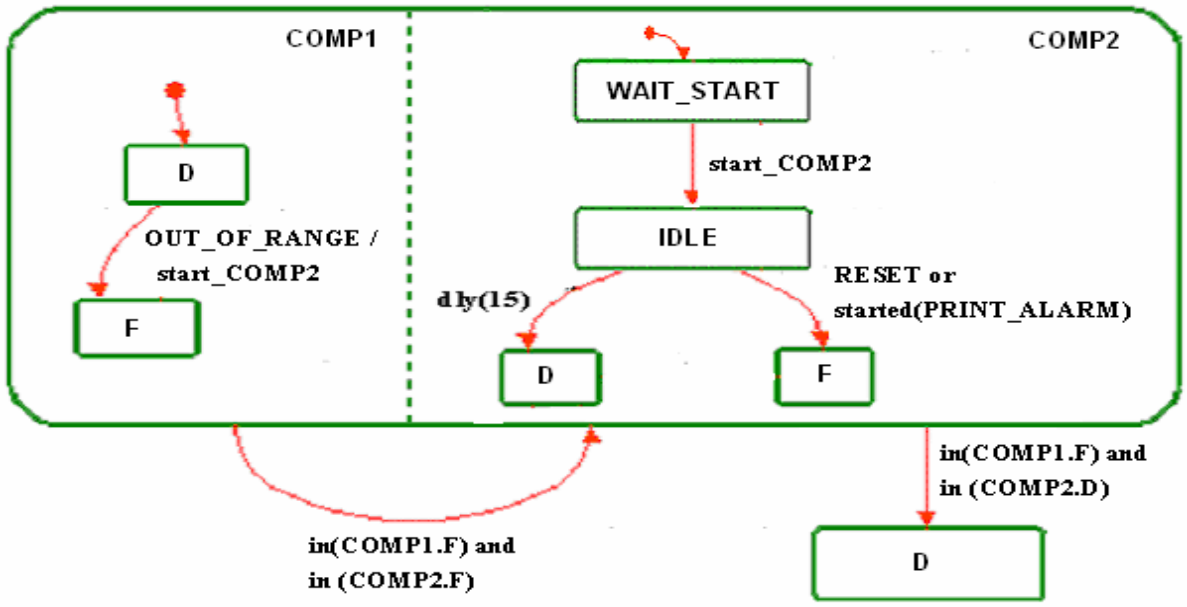


Fig. 3 Monitor chart for the assertion *ALWAYS (OUT_OF_RANGE → EVENTUALLY (15) (RESET or started(PRINT_ALARM))*

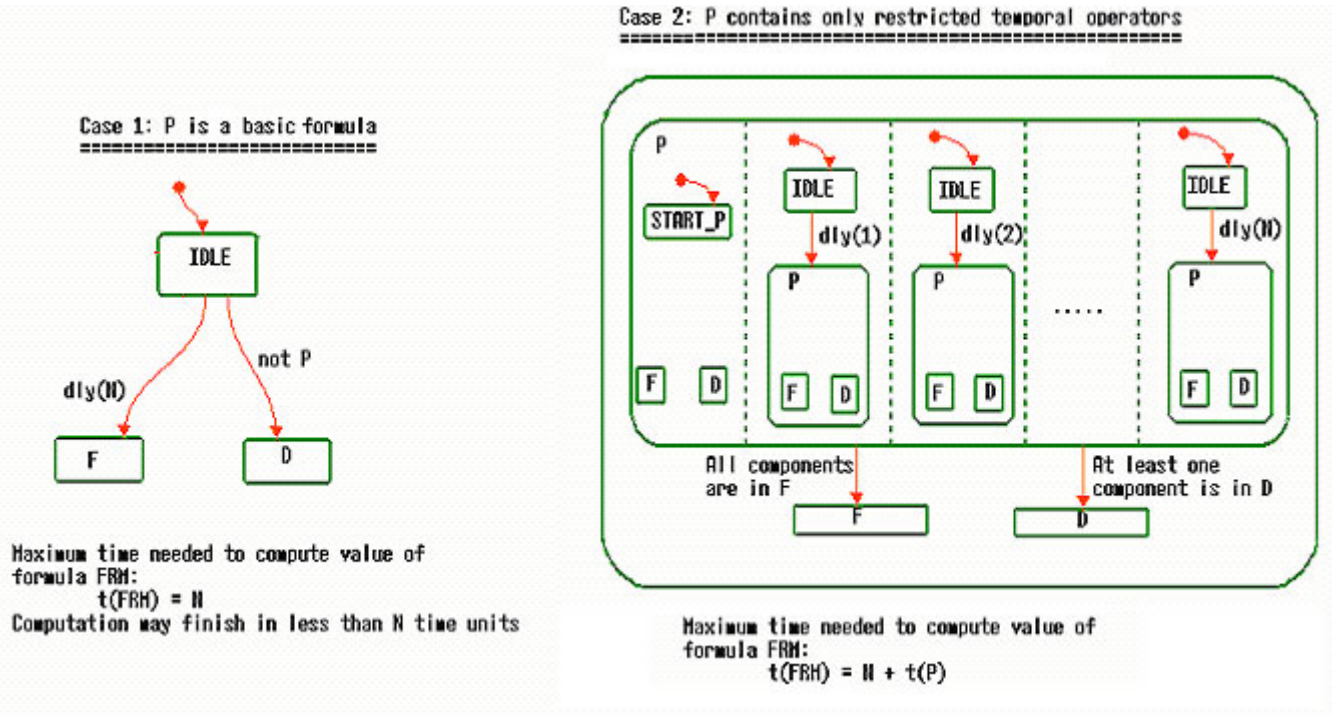


Fig. 4 Translation patterns for formula *ALWAYS (N) P*

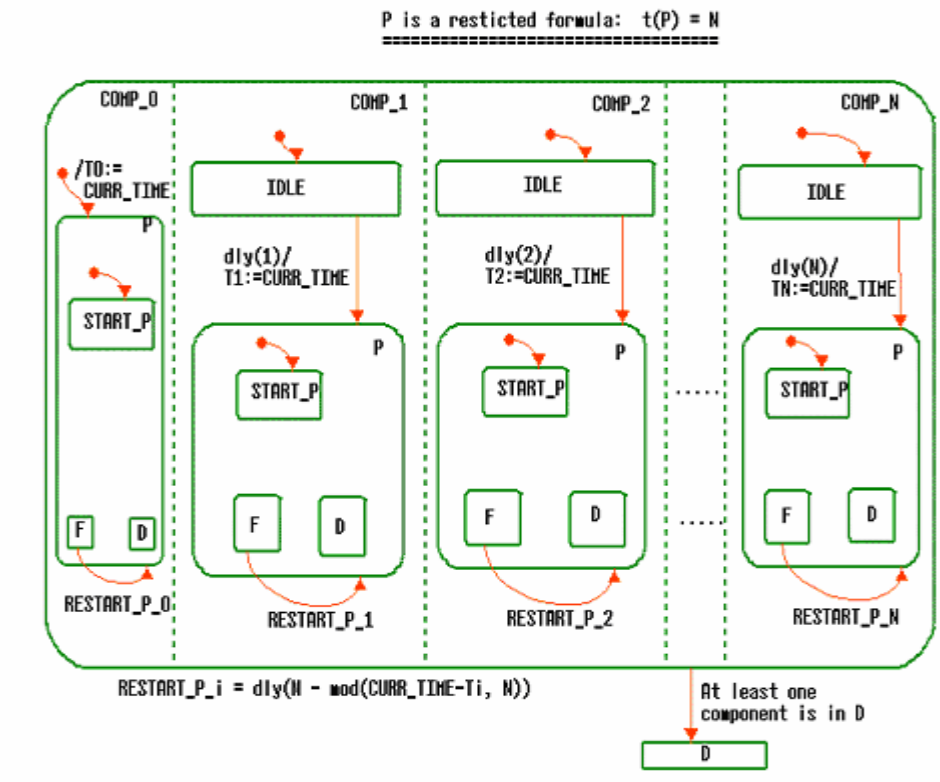
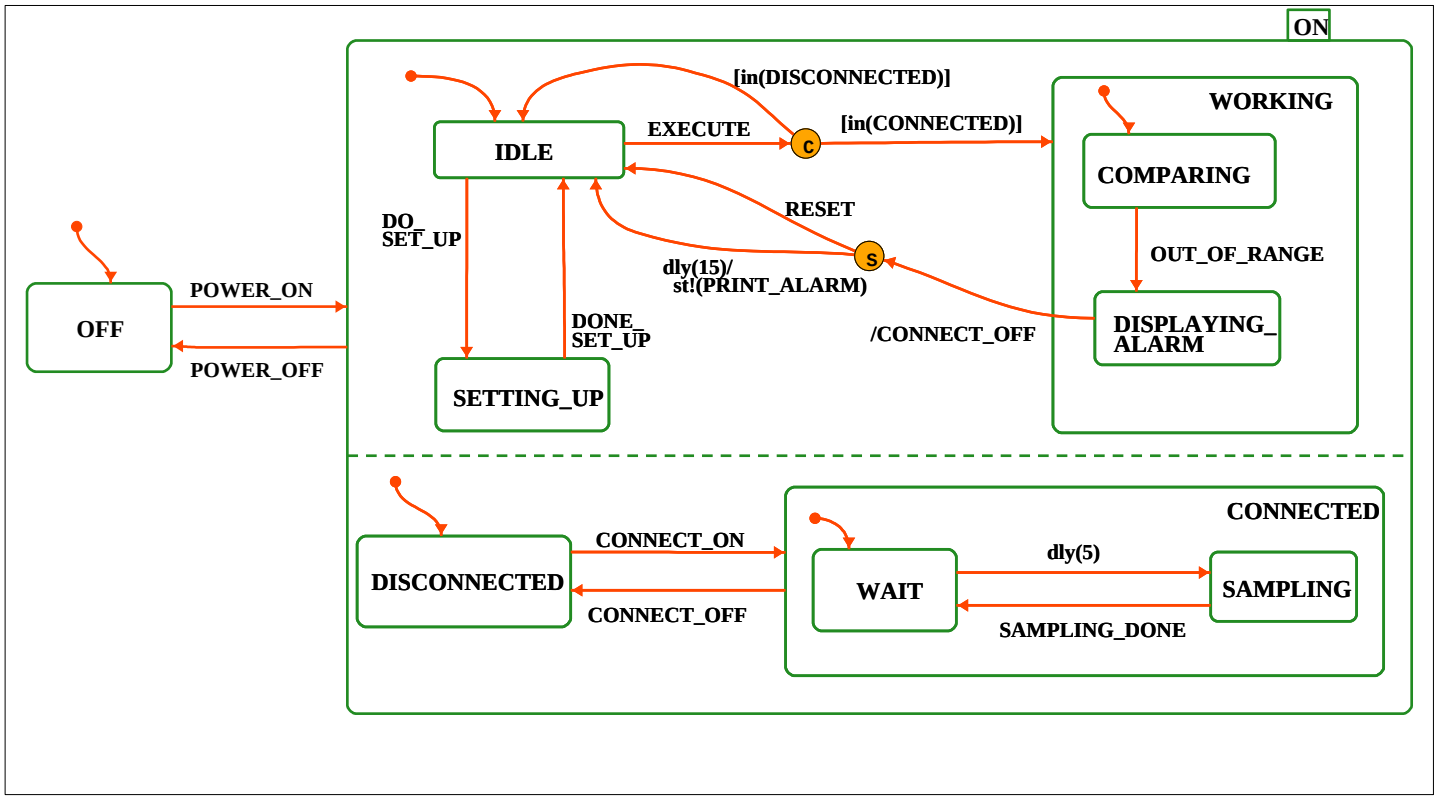
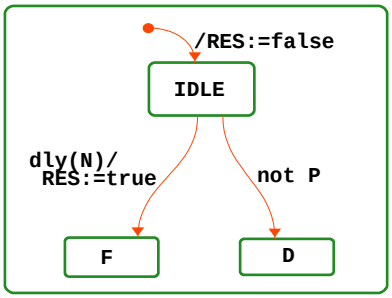


Fig. 5 Translation pattern for formula *ALWAYS P*

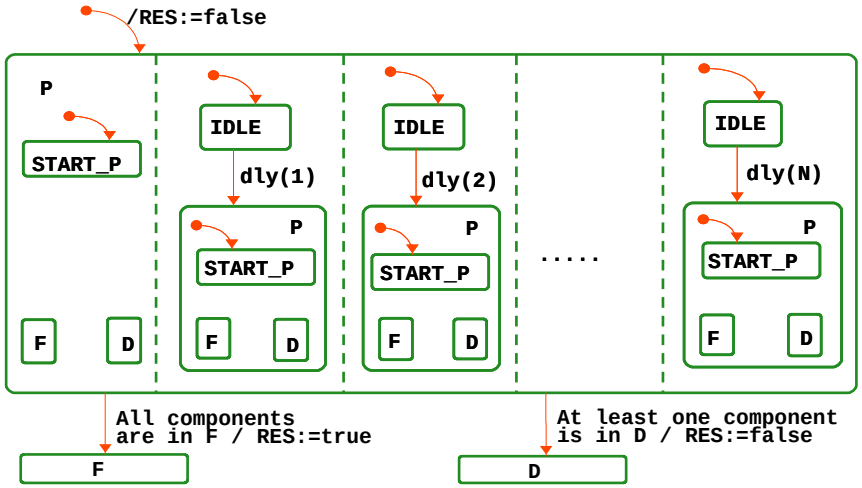

ON
OFF
POWER_ON
POWER_OFF
IDLE
[in(DISCONNECTED)]
EXECUTE
C
[in(CONNECTED)]
DO
SET_UP
DONE_
SET_UP
SETTING_UP
RESET
S
dly(15)/
st!(PRINT_ALARM)
/CONNECT_OFF
WORKING
COMPARING
OUT_OF_RANGE
DISPLAYING_
ALARM
DISCONNECTED
CONNECT_ON
CONNECT_OFF
CONNECTED
WAIT
dly(5)
SAMPLING
SAMPLING_DONE

Case 1: P a is basic formula



Maximum time needed to compute value of formula FRM:
t(FRM) = N
Computation may finish in less than N time units

Case 2: P contains only restricted temporal operators



Maximum time needed to compute value of formula FRM:
t(FRM) = N + t(P)