Real-Time and Embedded System Verification Based on Formal Requirements

B. Fontan *,**, L. Apvrille***, P. de Saqui-Sannes*,**, J.-P. Courtiat*

*LAAS-CNRS, 7 avenue du Colonel Roche, 31077 Toulouse Cedex 04, France
**ENSICA, 1 place Emile Blouin, 31056 Toulouse Cedex 05, France
*** GET/ENST, 2229 route des Crêtes, B.P. 193, 06904 Sophia-Antipolis Cedex, France
bfontan@ensica.fr ; ludovic.apvrille@telecom-paris.fr ; desaqui@ensica.fr ; courtiat@laas.fr

Abstract

TURTLE is a real-time UML profile supported by a toolkit which enables application of formal verification techniques to the analysis, design and deployment phases of systems design trajectory. This paper extends the TURTLE methodology with a requirement capture phase. SysML requirement diagrams are introduced. Temporal requirements (TR) are formally expressed using a dedicated language based on Allen's interval algebra. TRs serve as starting point to automatically synthesize observers and to guide the verification process applied to the TURTLE model of the system. Verification results are automatically collected in traceability matrices. A Hybrid Sport Utility Vehicle serves as example.

1. Introduction

The concept of "profile" has extensively been used to customize the Unified Modeling Language (UML [14]) standardized by the Object Management Group. OMEGA [10], SYNCHARTS [2] and TURTLE [3] are examples of real-time UML profiles supported by toolkits that enable application of formal verification techniques. These profiles have mostly been developed outside the OMG standardization process. They nevertheless contribute to add formality to the OMG-based notation and they demonstrate the power of formal verification techniques not implemented by commercial real-time UML tools such as TAU G2 [18] or Rhapsody [18].

This paper specifically addresses TURTLE, a real-time UML profile based on the RT-LOTOS [6] formal language. TURTLE is supported by TTool [20], an open-source tool which includes a TURTLE diagrams editor and a RT-LOTOS code generator interfaced with RTL [16] and CADP [4]. RTL implements reachability analysis of RT-LOTOS specifications derived from TURTLE models. CADP is used to minimize the reachability graphs generated by RTL.

The TURTLE methodology and toolkit enable application of formal verification techniques to the analysis, design and deployment phases of systems design trajectory. In this paper, we propose to extend the TURTLE methodology with a requirement capture phase. SysML [17] requirement diagrams are introduced. Any type of informal requirement may be expressed. Nevertheless, specific attention is laid on temporal requirements. The latter are expressed using a dedicated language based on Allen's intervals. An important contribution is that we establish links between temporal requirements and formal verification. Indeed, observers are automatically synthesized from temporal requirements and synchronized with the appropriated TURTLE diagrams of the system under design. Further, verification results are automatically collected in a traceability matrix.

The paper is organized as follows. Section 2 presents the TURTLE profile. Section 3 explains how the TURTLE methodology is extended to capture requirements. Section 4 extends TURTLE with requirement diagrams and defines a language for temporal requirement specification. Section 5 discusses formal verification guided by observers and outlines 'requirement to observers' synthesis algorithms. Section 6 discusses application of extended TURTLE to an Hybrid Sport Utility Vehicle (HSUV). Section 7 surveys related work. Section 8 concludes the paper.

2. TURTLE

The TURTLE profile has been designed with formal verification in mind. Both analysis and design diagrams may be translated into RT-LOTOS so as to reuse already existing verification tools [16] [4] for the profit of TURTLE. This section briefly presents analysis and design diagrams. Then it explains the verification approach that may be applied to either category of diagrams.

2.1. Analysis diagrams

The designer builds up one use-case diagram and one Interaction Overview Diagram (IOD) which structures a set of scenarios expressed as Sequence diagrams (SD). A TURTLE IOD makes it possible to express that one scenario (SD) may interrupt another scenario at any time. Further, a scenario may explicitly refer to absolute or relative dates, respectively.

1.1. Design diagrams

A TURTLE design [3] starts with one class/object diagram which defines the architecture of the system. Like processes in a process algebra language, TURTLE objects may be composed to formally express parallelism, synchronization (via gates), and preemption between two objects.

The behaviors of the objects are separately defined in activity diagrams. TURTLE extends UML activity diagrams with synchronization actions. New temporal operators make it possible to express a deterministic delay, a non deterministic delay that may be associated to a deterministic one to create a time interval, and a time limited offer that prevents a ready-to-synchronize object to be blocked for ever.

1.2. Formal verification

We start from either analysis or design diagrams and generate the corresponding RT-LOTOS specification. Assuming the system under design is bounded and of reasonable size, the RTL tool may generate the reachability graph of that RT-LOTOS specification. The graph is often too complex to be analyzed by hand. Therefore, its transitions are decorated with those actions identified as important with respect to the set of requirements to be verified. The result is a labeled transition system that may me minimized using, e.g., Milner's observational equivalence [13]. The minimization process outputs one quotient automaton which gives an abstract view of the system modeled in TURTLE.

Reachability analysis and minimization take into account the complete set of objects declared in the TURTLE model. This set of objects may be limited to those objects which are parts of the system under design. Observers may be added to that TURTLE model. Verification applies to a model made up of two types of objects: the system objects and observer objects.

So far, observers have been built up manually. In this paper, it is shown that observers linked with temporal requirements may be automatically synthesized from these requirements and added to the TURTLE model of the system in order to build up a verification-oriented model.

2. Extended Methodology

This section introduces the four-step methodology depicted by Fig. 1.

The requirement capture phase starts with a requirement diagram (RD) definition. Each node in the RD defines one requirement in plain text, which means that the requirement in question is informal (in the sense that it is written in, e.g., English and not using a language whose syntax and semantics are formally defined). Both functional and non functional requirements may be expressed.

Temporal requirements (TRs) are identified in the previously created requirement diagram (RD). TRs are expressed using the language defined in section 3. These formally expressed TRs are added to the RD. They are used to synthesize observers intended to guide formal verification and to help achieve traceability.

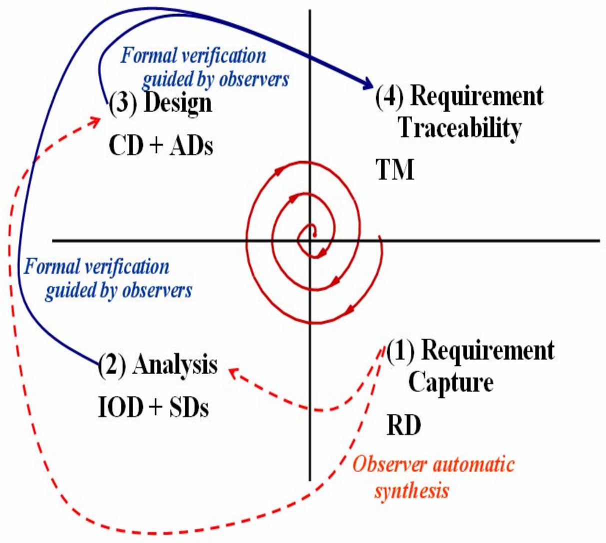

Formal verification guided by observers
(3) Design
CD + ADs
(4) Requirement Traceability
TM
Formal verification guided by observers
(2) Analysis
IOD + SDs
(1) Requirement Capture
RD
Observer automatic synthesis

Fig.1. The TURTLE methodology with requirement expression and verification

The two dashed lines in Fig.1 indicate that observers may be generated to be associated with analysis or design, respectively. Automatic synthesis algorithms are described in section 4.2.

Also, a traceability matrix is automatically generated from the results collected by the observers. Again, automatically generated observers are used to produce quotient automata demonstrating whether observed properties hold or not. More details on that matrix are provided in section.4.4

3. Requirement capture in extended TURTLE

3.1. Informal presentation of the proposed extension

In SysML, a requirement is a test case stereotyped by <<requirement>> and characterized by four attributes: an *identifier*, a *text* (an informal description of the requirement), a *type* (functional, performance, etc.), and a *criticality* level.

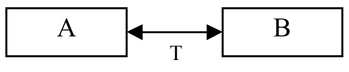
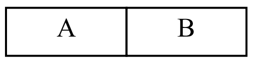
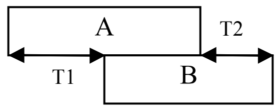
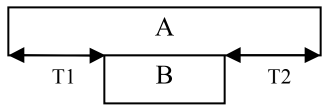
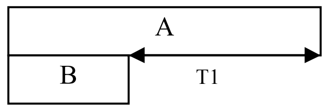
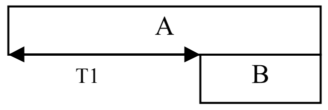
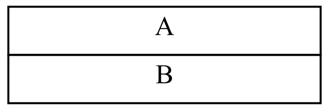
In TURTLE, a requirement has the same structure as in SysML. It may remain informal. An interesting point is that the "text" attribute may also unambiguously express a temporal requirement using the language proposed in section 4.2. Further a "*RequirementViolationEvent*" attribute is created to define the label to be used in the quotient automaton output by the reachability graph minimization so as to indicate that the temporal requirement is not met.

3.2. Temporal relations

Temporal relations define the relative positions between events. The description of theses relations is based on two classes of temporal models:

- Point based models contain elementary units representing single events. They are described by four relations:
  - *A LESS_T B T*: If A occurs at Ta then B must occur within [Ta,Ta+T].
  - *A GREAT_TB T*: If A occurs at Ta then B must occur in [Ta+T ; +∞ [.
  - *A BETWEEN B [T1,T2]:* If A occurs at Ta then B must occur in [Ta+T1 ; Ta+T2].
  - *A OUTSIDE B [T1,T2]*: If A occurs at Ta then B must occur in] -∞ ; Ta+T1] U [Ta+T2 ; +∞ [.

- Interval based models defined by Allen's interval algebra.
  According to [1], the relative positions of two processes characterized by their respective departure and completion dates may be characterized by thirteen patterns that reduce to seven patterns (Tab. 1) when symmetrical situations are taken into account.

Relation	Example
Before	
Meets	
Overlaps	
During	
Start	
End	
Equal	

Tab. 1. Allen's relations [1]

3.3. Temporal requirement expression language

The language for temporal requirements with relative dates is defined by the following BNF:

```
Relative-point-temporal-requirement =
            event "LESS_T" event integer
          | event "GREAT_T" event integer
          | event "BETWEEN" event "[" integer "," integer "]"
          | event "OUTSIDE" event "[" integer "," integer "]"
```

where "event" denotes an interaction between two TURTLE objects. For simplicity we express dates as integers (as logical unit).

The language is extended to cope with relative intervals.

```
Relative-interval-temporal-requirement =
  "(" b-event "," e-event ")" "BEFORE" "(" b-event "," e-event ")"
| "(" b-event "," e-event ")" "MEETS" "(" b-event "," e-event ")"
| "(" b-event "," e-event ")" "OVERLAPS" "(" b-event "," e-event ")"
| "(" b-event "," e-event ")" "DURING" "(" b-event "," e-event ")"
| "(" b-event "," e-event ")" "STARTS" "(" b-event "," e-event ")"
| "(" b-event "," e-event ")" "FINISHES" "(" b-event "," e-event ")"
| "(" b-event "," e-event ")""EQUALS" "(" b-event "," e-event ")"
```

The above BNF assumes there exists two "processes" which begin and end by performing b-event and e-event, respectively. Three among the above operators may be extended to deal with fixed duration. The BNF is extended as follows:

```
Relative-interval-temporal-requirement =
…
| "(" b-event "," e-event ")" "BEFORE-T" "(" b-event "," e-event ")"
integer
|"(" b-event "," e-event "OVERLAPS-T" "(" b-event "," e-event ")
"["integer "," integer"]"
| "(" b-event e-event "DURING-T" "(" b-event "," e-event ")" "["integer
"," integer"]"
```

The above extension may be used, e.g., to say that one process A may end before one process B starts, and that three time units must elapse between A's completion and B's departure.

4. Formal Verification guided by Observers

In TURTLE, formal verification relies on reachability analysis and labeled transition system minimization. Besides the reachability graph minimization, the verification process is further guided by observers. We add the TURTLE model one (or several) <<*tobserver*>> object(s) that we synchronize with appropriate object(s), i.e. those TURTLE objects which are expected to implement the requirement to be verified.

4.1. Observer Taxonomy

To the question "Is there a unique type of observer working for all the operators accepted by TURTLE requirements diagrams?" the answer is "no Tab. 2 identifies various types of observers.

Features	Type	Description
Behavior	Passive	Remains passive if the requirement is not satisfied
	Active	Cuts all objects behavior if the requirement is violated
Nature	Time /Point	Based on relative position of events
	Time /Interval	Based on Allen's interval algebra
Construction	Local Observer	Synchronized with the object which contributes to requirement satisfaction
	Global Observer	Synchronized with several objects
Function	Non Intrusive	The behavior of observed objects is not modified
	Intrusive	Likely to modify the behavior of observed objects

Tab. 2. Observer taxonomy

4.2. Observer automatic synthesis

To support the methodology presented in section 2 we implement an automatic observer generation process. From formal requirements, it is possible to automatically derive observers. Currently, this generation process is proposed for TURTLE analysis and design diagrams (and not deployments ones).

A user must first describe which requirements he/she desires to observe. Observers are SysML test cases, stereotyped as <<*tobserver*>> (TURTLE observers) that may be added to TURTLE Requirements Diagrams (RDs). An observer has a name and makes reference to one TURTLE analysis or design. The "verify" relationship defines how a test case verifies a requirement. In SysML, a test case is intended to be used as a general mechanism to represent verification methods [17]. Then, for observers linked to a formal requirement using a <<verify>> relation, an automatic generation process modifies the analysis or design TURTLE diagrams in order to integrate the observer.

For example, the RD depicted in Fig.2 contains two observers. *ObserverXRq0* is intended to verify requirement *X* (e.g. "Brake" or "Cruise") on the TURTLE design named *DesignWithObserver*. *ObserverX01Rq1* is intended to verify requirement *X01* on the same design. *ObserveXRq012* is aimed to verify requirement *X01* on the TURTLE analysis diagrams *AnalysisWithObserver*. *ObserverXRq0* is intended to check the system against an informal requirement. Therefore, that observer must be modeled by hand (not automatically) and inserted by the designer inside the appropriate diagrams. On the contrary, *ObserverX01Rq1* and *ObserverX01Rq2* check the system against a formal, temporal requirement. Consequently, they may be automatically generated using the synthesis process described afterwards.

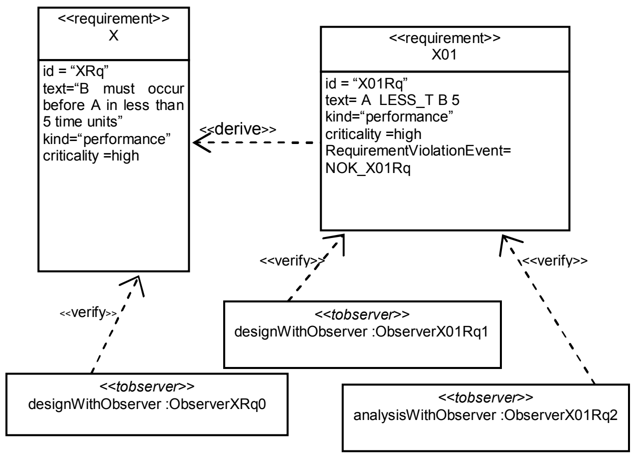


Fig.2. Example of a TURTLE requirement diagram

Automatic generation of observers works as follows. For each <<*tobserver*>> of a RD, it selects the ones observing formal requirements. Then, for each observer, it considers the diagram (analysis or design) on which the corresponding behavior must be added:

- ***For an analysis diagram***, a preemption operator is added to the high-level interaction overview diagram. This operator makes it possible for the observer to interrupt the system whenever it observes the violation of its corresponding property. Also, for each action it observes on a scenario, a synchronous message is added to this scenario. The associated arrows start from the lifeline of the object performing the action to be observed, and ends to the observer lifeline.

- ***For a design diagram***, the process modifies the class diagram. A new Tclass (TURTLE class) is inserted into the diagram. This Tclass synchronizes with all the Tclasses doing those actions listed in the requirement. Also, observed Tclasses must be modified as follows: first, for each action that must be observed, a duplicate action synchronized with the observer is generated and added to their activity diagram (see the example below for more details). The behavior of the observer's Tclass depends on the formal description of the requirement (see 4.3.). Whenever the observed property is violated, the observer stops observing actions, and therefore, observed classes are blocked because synchronization on their observed actions is not offered anymore.

Other requirements parameters are also taken into account to generate the observers, depending on the taxonomy presented in 4.1:

- *id:* used for generating the observer's name as shown in the requirement diagram in Fig.2.
- *text*: used to synthesize the appropriate observer. We have defined translation patterns for every expression of the BNF defined in 3.3. The operator (e.g. LESS_T) determines the observer's nature (as shown in Tab. 2). The operator permits to generate the appropriate observer to guide verification of the formal requirement. To each formal requirement operator corresponds one observer's pattern. These patterns are described in section 4.3.
  Events in the formal text provide information about observer's construction (as shown in Tab. 2). If the events specified in the (formal) text attribute concern the same object then the observer is local; otherwise the observer's construction is global.
- *Kind*: used for requirement documentation.
- *Criticality* defines the observer's behavior (as shown in Tab. 2). We distinguish between three levels of criticality:
  - Low: the observer is passive; it does not stop the system.
  - Medium: the observer is active; it stops all the objects concerned by the requirement.
  - High: the observer is active; it stops all the objects in the diagram.

  Note: An active observer is necessarily intrusive because it will stop the object's behavior if the formal requirement is violated.
- *RequirementViolationEvent:* specifies the label (identifier) used by the observer to denote if the requirement is not satisfied. This label will appear in the quotient automaton each time the corresponding requirement is not satisfied.

Once observers have been automatically generated from formal requirements, another process automatically generates traceability matrices to clearly establish connections between formal requirements and verification results. Thus, a quotient automaton is automatically generated to show which requirements are not satisfied and why they are not. This quotient automaton features all the events related to a given requirement.

4.3. Observer patterns

Observer patterns rely on elementary TURTLE operators: for example, *timer* in analysis and *time limited offer* in design. These operators are useful for translating temporal descriptions presented in section 3.3.

All the patterns are built upon two sub patterns called *LESS_T* and *GREAT_T*. These two patterns are described hereafter, assuming that the action to execute when the property is violated is *NOK* (Not OK).

- *LESS_T*

  This pattern is used for modeling a maximum duration.

  On TURTLE analysis diagrams, a maximum duration may be modeled using a timer operator. As shown in Fig.3, the timer is set when the observer receives the *Start* action. If the timer expires before the observer receives the *Capture* action, the observer executes the *NOK* action.

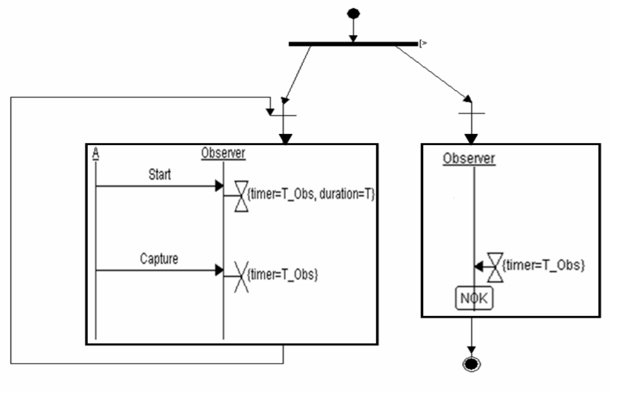


Fig.3. Analysis observer pattern generated for the LESS_T requirement. Criticality is assumed to be high.

For TURTLE designs, the *time limited offer* operator is used. As shown in Fig.4, the *time limited offer* starts just after the *Start* action was executed. The observer expects *Capture* to occur before T time units. After T time units, the observer executes the *NOK* action.
Note: depending on the requirement's criticality, the execution of the *NOK* action may stop the execution of the system or not (see 4.2).

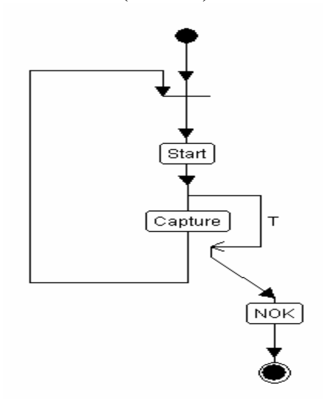


Fig.4. Design observer pattern generated for the LESS_T requirement. Criticality is assumed to be high.

- *GREAT_T*

Fig.5 shows that the *NOK* action is sent if the observer receives one *Capture* action before the *timer* expires. Thus, GREAT_T models a minimal duration.

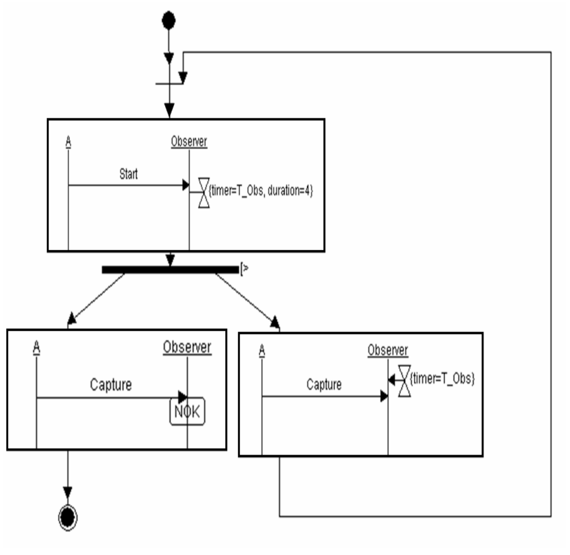


Fig.5. Analysis observer pattern generated for the GREAT_T requirement. Criticality is assumed to be high.

In Fig.6, a *time limited offer* on action *Capture* starts once action *Start* has been performed. Then, if *Capture* is received before T time units, the observer sends the *NOK* action.

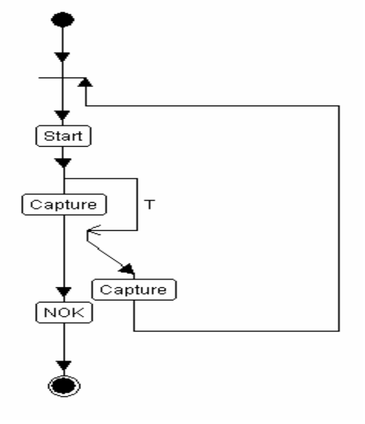


Fig.6. Design observer pattern generated for the GREAT_T requirement. Criticality is assumed to be high.

All observer patterns are built upon LESS_T and GREAT_T (Tab. 3).
Note : A—B means "A is immediately followed by B".

Requirement expression	Excerpts of patterns
A BETWEEN B [T1,T2]	A GREAT_T B T1—A LESS_T B (T2-T1)
A OUTSIDE B [T1,T2]	A LESS_T B T1—A GREAT_T B (T2-T1)
(Ab,Ae) BEFORE (Bb,Be)	Ae GREAT_T Bb 1
(Ab,Ae) MEETS (Bb,Be)	Ae LESS_T Bb 1
(Ab,Ae) OVERLAPS (Bb,Be)	Ab GREAT_T 1 Bb—Ae GREAT_T Be 1
(Ab,Ae) DURING (Bb,Be)	Ab GREAT_T 1 Bb—Be GREAT_T Ae 1
(Ab,Ae) STARTS (Bb,Be)	Ab LESS_T 1 Bb
(Ab,Ae) FINISHES (Bb,Be)	Ae LESS_T 1 Be
(Ab,Ae) EQUAL (Bb;Be)	Ab LESS_T 1 Bb—Ae LESS_T 1 Be
(Ab,Ae) BEFORE_T (Bb,Be) T	Ae GREAT_T Bb T-1—Ae LESS_T Bb 2
(Ab,Ae) OVERLAPS_T (Bb,Be) [T1,T2]	Ab GREAT_T T1-1 Bb—Ab LESS_T Bb 2—Ae GREAT_T Be T2-1—Ae LESS_T Be 2
(Ab,Ae) DURING_T (Bb,Be) [T1,T2]	Ab GREAT_T T1-1 Bb—Ab LESS_T Bb 2—Be GREAT_T Ae T2-1—Be LESS_T Ae 2

Tab. 3. Observer patterns derived from requirement expressions

4.4. Requirement Traceability

The methodology presented in section 2 proposes to generate a traceability matrix from observer-guided verification results. An example of TURTLE traceability matrix is given in Tab. 4.

Req_ID	Form_Req	Satisfaction
XReq	Temporal requirement specification defined in 3.3 (e.g. A LESS_T B 5)	Response (YES/NO) QA_XReq

Tab. 4. Example of TURTLE Traceability matrix

The TURTLE Traceability Matrix (TM) contains:

- Req_ID: Requirement Identifier defined in the requirement diagram.
- Form_Req: Formal Requirement defined in the requirement diagram (described in section 3.3)
- Satisfaction part: described by
  - Response collected using observers. If the labeled action associated with the requirement violation (NOK_Xreq defined in Fig.2) appears in the reachability graph, a "NO" response is displayed in the TM. In the opposite case a "YES" response is displayed in the TM.
  - One quotient automaton is generated so as to highlight those actions in the reachability graph which deals with the temporal requirement. Also, this quotient automaton possibly contains one or several transition(s) labeled by the action introduced to characterize the violation of that requirement.

5. Application to the Hybrid Sport Utility Vehicle (HSUV) example

5.1. Requirements Capture

The following sentence defines an informal requirement for the Hybrid Sport Utility Vehicle (HSUV). "The hybrid SUV shall have the braking and acceleration of a typical SUV. It is expected to be dramatically better fuel economy". "HSUV" requirement is split up into "Cruise" and "Brake" requirements.

The "Cruise" requirement concerns the interaction between *CombustionEngine* and *ElecMotor* objects. The combustion process (Fig.7) must overlap the electrical one by two time units at the beginning and two time units at the end (Fig.7). Both engines have three functioning modes.

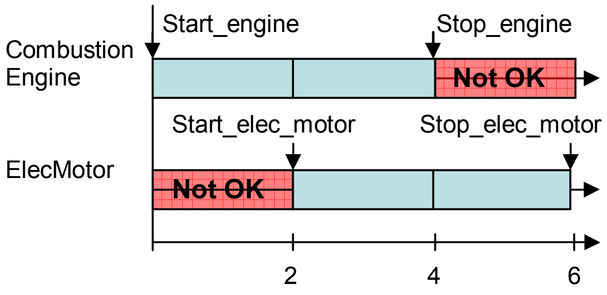
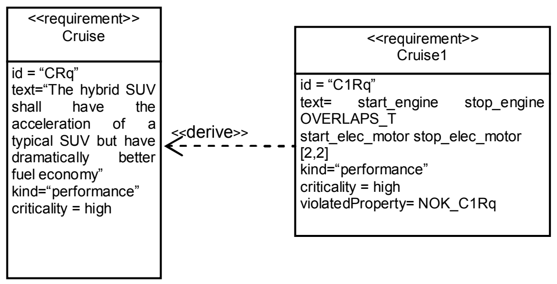


Fig.7. Requirement description for Cruise

It shall be quite difficult, if not impossible, to formally prove that the hybrid SUV has the acceleration of a typical SUV but has dramatically better fuel economy. By contrast, it is possible to formalize the temporal constraints (shown in Fig.7) using Allen's *OVERLAPS* operator. As depicted by Fig.8, the relation between the two requirements is depicted by one "derive" arrow. The formal requirement (right part of Fig.8) is the one considered for formal verification.



<<requirement>>
Cruise
id = "CRq"
text="The hybrid SUV shall have the acceleration of a typical SUV but have dramatically better fuel economy"
kind="performance"
criticality = high
<<derive>>
<<requirement>>
Cruise1
id = "C1Rq"
text= start_engine stop_engine OVERLAPS_T start_elec_motor stop_elec_motor [2,2]
kind="performance"
criticality = high
violatedProperty= NOK_C1Rq

Fig.8. Requirement diagram for Cruise

The "Brake" requirement concerns the duration between the call of Brake (e.g. push the brake pedal) and the Braking action which represents the end of the braking process. Arbitrarily, the maximum time between the *Brake* and *Braking* actions is 2 time units (TU). If this process takes more time than 2 TU, the Brake requirement is not satisfied (Fig.9).

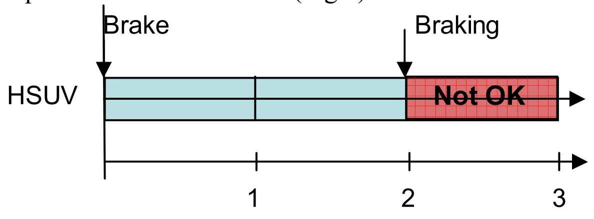


Fig.9. Requirement description for Brake

The temporal requirement expressed in Fig.9 is used to create the Brake requirement depicted in Fig.10.

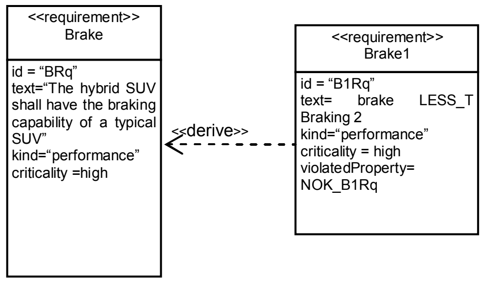


Fig.10. Requirement diagram for Brake

5.2. Analysis

Analysis diagrams of the HSUV are performed once requirements have been captured. The IOD depicted in Fig.11 describes the interactions between a *Start_Vehicle* scenario (not presented here), a *Cruise* scenario (e.g. push the acceleration pedal, see Fig.12), and a *Brake* scenario (e.g. push the brake pedal, see Fig.13). Indeed, a driver pushes either the cruise pedal or the brake one (as with standard vehicles).

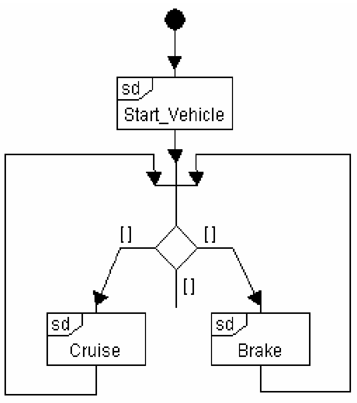


Fig.11. Interaction Overview Diagram of the HSUV

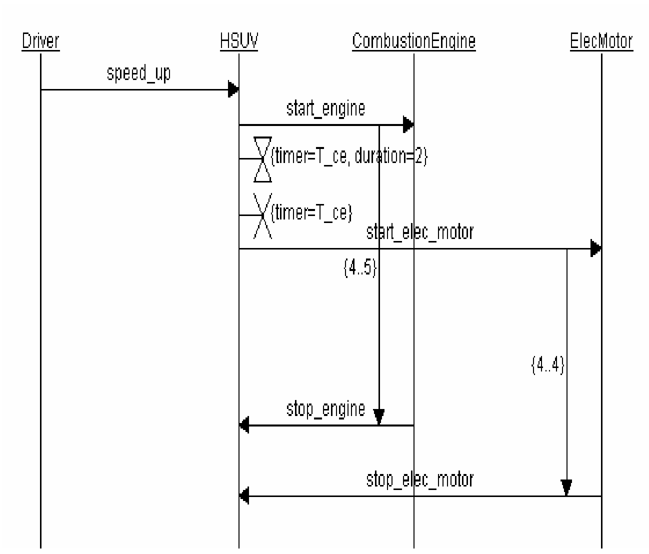


Fig.12. Cruise scenario of the HSUV

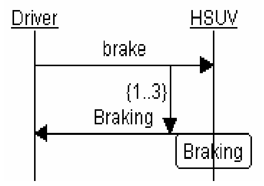


Fig.13. Brake scenario of the HSUV

Once modeled, the second step is to verify the requirements depicted in Fig.8 and Fig.10 over these diagrams. Tab. 5 characterizes the observer for the formal and temporal requirements depicted by Fig.8 and Fig.10, respectively. The taxonomy was defined in section 4.1.

ID	Behavior	Nature	Construction	Function
Observer B1Rq	Active	Time LESS_T	Local Observer (HSUV)	Intrusive
Observer C1Rq	Active	Time OVERLAPS _T	Global Observer (ElecMotor, CombustionEngine)	Intrusive

Tab. 5. Applying the taxonomy to the HSUV's requirement

The generation of observers modifies analysis diagrams as follows. First, the IOD (see Fig.14) is enhanced with a reference to a new IOD named *iod_observer*. *iod_observer* preempts the other sequences diagrams to cut those objects whose behavior contribute to implement the two requirements to be verified (high criticality). The new IOD models the two requirements to be verified. Their behaviors correspond to the patterns introduced in section 4.3.

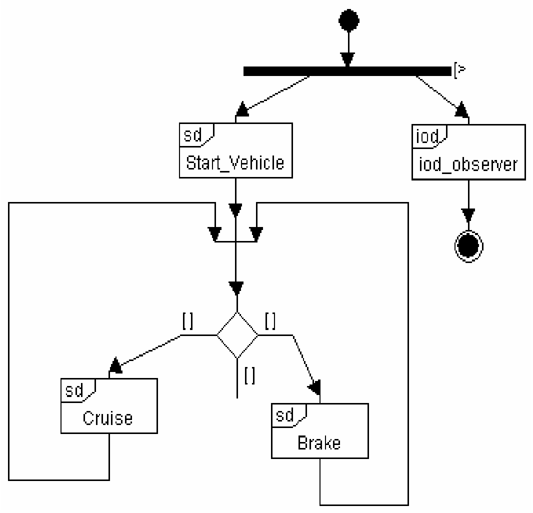


Fig.14. IOD with automatically generated observers

5.3. Design

Also, in the design phase, the TURTLE class diagram is automatically enhanced with two observers (as shown in Fig.15). This corresponds to the two formal requirements described by.Fig.8 and Fig.10. They are connected to other Tclasses, according to the taxonomy defined in 4.1.

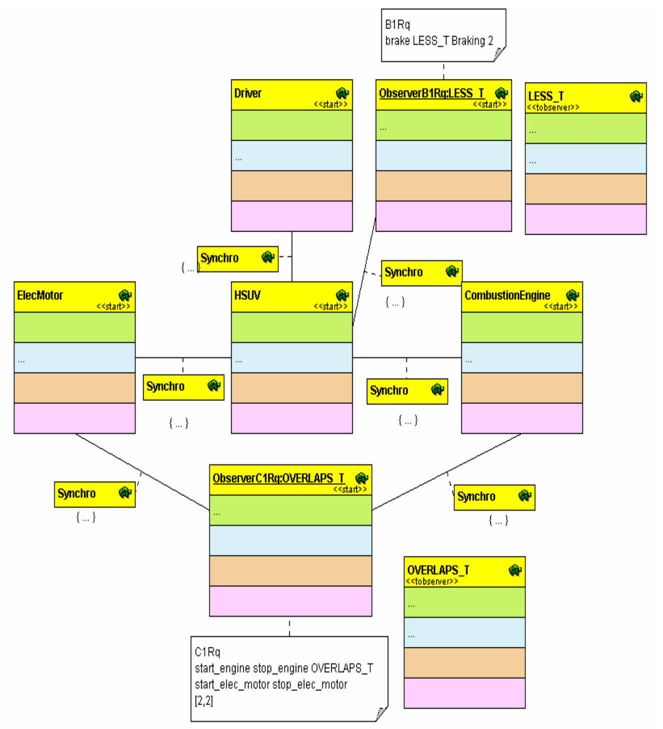


Fig.15. Class diagram of HSUV including automatically generated observers

5.4. Requirement Traceability

A traceability matrix (see Tab. 6) is generated from the verification results obtained for either analysis or design diagrams enhanced with observers. It gathers all information regarding the verification of formal requirements (see section 4.4).

Req_ID	Req_Text	Satisfaction
B1Rq	brake LESS_TBraking 2	NO **QA_B1Rq**
C1Rq	start_engine stop_engine OVERLAPS_T start_elec_motor stop_elec_motor [2,2]	NO **QA_C1Rq**

Tab. 6. Traceability matrix for the case study

The above matrix indicates that none of the requirements is satisfied:

- o The Brake process takes more time (maximum 3 time units, see Fig.16) than expected by its requirement (2 time units).

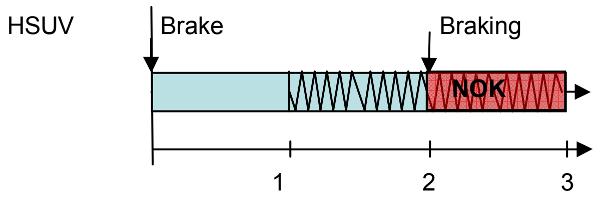


Fig.16. Brake process anomaly

- o In the cruise process, the ElecMotor starts on time. One problem occurs at the end of CombustionEngine process, which may end later than expected (see Fig.17). The duration of CombustionEngine must be modified to fix this problem.

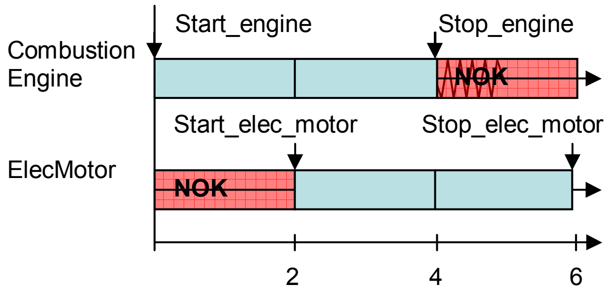


Fig.17. Cruise process anomaly

Today, the observer automatic generation and verification process is still under development. It will be supported very soon by next beta version of TTool.

6. Related Work

SysML particularly answers expectations of system engineers for a UML-based notation that would be less software centric than UML 2.0 and that would bring support for the requirement capture phase [11]. For instance, [19] proposes an extended SysML with bond graphs, a notation used to describe energy flows between mechanical blocks inside one system. Unlike [19], this paper does not reuse SysML block diagrams and ignores the functional design style inherent to diagramming with SysML blocks.

The novelty in TURTLE requirement diagrams lies in the possibility to formally express requirements and to associate them with verification results.

Several requirement expression languages have already been proposed in the literature. Parts of them are specific to a single application domain. For instance, LDE applies to avionic systems in the framework of CARROLL project [5]. Also, EAST-ADL applies to automotive architecture in the framework of EAST-EEA [9] (ITEA project). Among various general purpose languages, [8] proposes to build constraint diagrams based on duration calculus. Objectiver tool [15] is based on the KAOS [7] methodology which provides a language and a method for goal driven requirement elaboration. This tool enables analysts to elicit and specify requirements in a systematic way and to achieve traceability from requirements to goals.

A survey of the literature indicates that research work on requirements and formal verification are often closely related to each other. For instance, [12] proposes a graphical modeling language based on requirement patterns. The latter are translated into LTL formula and verified using model-checker SPIN. In [10], temporal requirements are expressed as state machines that define a so-called 'requirement and context model'. Both the system's model and the state machines are translated into IF, the Intermediate Form defined in the framework of OMEGA project. IF is supported by formal verification tools [10]. In [10] the authors say they are looking for a requirement expression language from which context state machines and observers might be automatically generated. This objective is met by the extended TURTLE discussed in this paper. Observers may indeed been generated from temporal requirements and verification results are included into traceability analysis.

7. Conclusions and future work

TURTLE is a real-time UML profile designed with formal verification in mind. The TURTLE toolkit indeed enables application of formal verification techniques throughout the life cycle which underlies the TURTLE methodology. So far, that methodology covered analysis, design and deployment diagrams. The requirement capture phase has been ignored.

This paper proposes to extend TURTLE with SysML requirement diagrams. Like SysML, TURTLE enables informal requirement description. Our profile is further extended in such a way that temporal requirements may be described in a language based on Allen's intervals algebra.

Both informal and formal requirements may be used for verification, and more precisely formal verification guided by observers. For informal requirements, it is the responsibility of the user of TTool to build up observers that are relevant for the requirements in question. The good news is that for those temporal requirements which are expressed using the language based on Allen's algebra, observers may be automatically synthesized. This applies to observers to be associated with analysis diagrams, and to observers to be synchronized with design diagrams as well.

Discussion in this paper has clearly been focused on temporal requirement expressions and formal verification. Automatic synthesis of observer is an important step towards requirement traceability. An important contribution of the paper is that traceability matrices may be automatically generated from formal verification results. The synthesis approach discussed in the paper is being implemented in TTool. The latter's diagramming capability is extended to support requirement diagrams.

Next step is to not limit formal requirements to temporal requirements based on Allen's algebra. Solutions are to be sought to include LTL formula inside the "text" field of TURTLE requirements.

References

[1] J.F. Allen, "Maintaining Knowledge About Temporal Intervals," *Communications of the ACM*, Vol. 26, No. 11, pp.832-843, Nov. 1983.

[2] C. André, "Computing SyncCharts," *ESLAP'03 (Synchronous Languages, Applications and Programming)*, Porto, Portugal, July 2003

[3] L. Apvrille, J.-P. Courtiat, C. Lohr, P. de Saqui-Sannes, "TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit", *IEEE Transactions on Software Engineering*, Vol. 30, No. 7, July 2004, pp.473-487.

[4] http://www.inrialpes.fr/vasy/7/

[5] http://www.carroll-research.org/

[6] J.P. Courtiat, C.A.S. Santos, C. Lohr., B. Outtaj, "Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique", *Computer Communications*, Vol. 23, No.12, pp.1104-1123, 2000.

[7] A.Dardenne, A.van Lamsweerde and S. Fickas, "Goal-Directed Requirements Acquisition", in *The Science of Computer Programming* 20, 1993.

[8] C. Dietz. "Graphical formalization of real-time requirements," In B. Jonsson and J. Parrow, eds, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*, Uppsala, Sweden, LNCS 1135, pp. 366-385, Springer-Verlag, 1996.

[9] http://www.east-eea.net/

[10] S. Graff, I. Ober, I. Ober, "Validating Timed UML Models by Simulation and Verification," *STTT International Journal On Software Tools for Technology Transfer*, 2005.

[11] M. Hause, F. Thom, A. Moore, "Inside SysML," *IEE Computing & Control Engineering*, pages 10-15, Aug./Sept. 2005.

[12] W. Janssen, R. Mateescu, S. Mauw, P. Fennema, P. Stappen, "Model Checking for Managers," *Spin'99*, pp.92-107, 1999.

[13] R. Milner, "Communication and Concurrency," *Prentice Hall*, 1989.

[14] Object Management Group, "Unified Modeling Language Specification", Version 1.5, http://www.omg.org/docs/formal/03-03-01.pdf, March 2003.

[15] http://www.objectiver.com/

[16] http://www.laas.fr/ RT-LOTOS

[17] http://www.SysML.org/docs/specs/SysML-v1-Draft-06-03-01.pdf

[18] http://www.telelogic.com

[19] S. Turki, T. Soriano, "A SysML extension for Bond Graph support," *5th International Conference on Technology and Automation, Thessaloniki*, Greece, October 2005.

[20] http://labsoc.comelec.enst.fr/TURTLE/