Model Checking UML Specifications of Real Time Software

Vieri Del Bianco
Politecnico di Milano
P.zza Leonardo Da Vinci, 32
20133 Milano, Italy
delbianc@elet.polimi.it

Luigi Lavazza
CEFRIEL Politecnico di Milano
P.zza Leonardo Da Vinci, 32
20133 Milano, Italy
lavazza@elet.polimi.it

Marco Mauri
CEFRIEL
Via Fucini, 2
20133 Milano, Italy
mmauri@cefriel.it

Abstract

UML is being increasingly used to model real-time software. On one hand this is reasonable, since UML is very popular and relatively easy to use. On the other hand, the semantics of UML is not well defined, thus UML does not support formal analysis, which is needed to prove properties like safety, utility, liveness, etc. This article describes a way to make UML models formally verifiable. The presented approach is made possible by extending UML in order to represent time-dependent information and time constraints, and by formalizing the resulting language. The formalization is achieved by mapping UML state diagrams to Timed Statecharts. UML state models are translated into timed automata, so that the model checking tool Kronos can be employed to verify time-dependent properties. A central issue of the work presented here is that developers can take advantage of the formal methods being employed while skipping the complex and expensive formal modeling phase.

1. Introduction

Formal methods have demonstrated to be effectively applicable in the industrial development of real-time safety critical systems. Nevertheless, formal methods are not widely used in industry. The problem is that while the demand for real-time software increases very quickly, the availability of developers who can master formal methods remains little. The consequence is that formal methods are generally considered too difficult or too expensive to be used in "ordinary" real-time software development. On the contrary, UML [1] has achieved a great popularity, essentially because it is a semi-formal notation relatively easy to use and well supported by tools. Interestingly, UML is gaining popularity also for real-time developments. However, the application of UML to the real-time domain is still limited by the lack of formal semantics. This situation can be improved by means of a dual language approach to real-time software development. In a first phase models are written in UML according to the usual development practices; in a second phase UML models are automatically translated into one or more formal notations, which provide support to activities such as the simulation, the verification of properties, the generation of test cases, etc. In this way, developers exploit the advantages of formal notations while skipping the complex and expensive formal modeling phase.

The proposed approach has been applied in a previous work [2] where UML models were translated into TRIO [3] temporal logic formulas. Although such work demonstrated the viability of the approach, there was a weak point: since TRIO is computationally complete, in general you cannot prove properties of a TRIO specification automatically. We used a history checker, which is completely automatic, but does not assure that the system will *always* behave as specified. Of course one could use a theorem prover to demonstrate the properties of the model, but this would violate our constraint to make the approach accessible by not particularly skilled developers.

Since UML is not very good at modeling time and time-dependent constraints, in [2] an extension of UML was defined. These extensions were formalized and described in [6].

In this paper we address the translation of UML models into timed automata [9], and model checking of the resulting automata by means of Kronos [4]. As UML is more powerful than timed automata, only a subset of all the possible UML models can be translated into Kronos automata: this issue is also discussed here.

Actually we do not need to use the whole expressive power of UML for modeling and verifying real-time properties. Here we consider only class diagrams, state diagrams and object diagrams: additional diagrams (e.g., use case diagrams) are simply ignored.

The proposed approach is applied to the specification of the CSMA/CD Protocol [8]. A UML model of the protocol is presented, and then translated into timed automata. Kronos is used to prove the relevant properties

of the protocol. In order to further validate the proposed approach, we also applied it to the real-time software used to control the motors of a medical device for blood analysis. The latter experience is reported very briefly for space reasons.

The main goal of the research underlying the work presented here is to demonstrate the viability of a development process based on UML modeling and the translation of the resulting models into a formal notation that allows the application of formal methods even when the developer (i.e., the author of the models) has a very little or null knowledge of the formal methods being employed. The development environment that originates from the work presented here (currently being developed at CEFRIEL) is described in Figure 1.

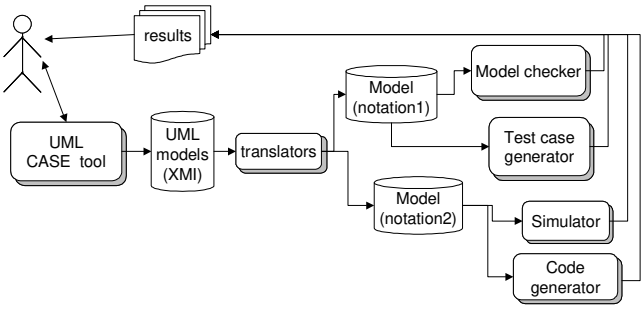


Figure 1. An abstract view of the envisaged development environment

Figure 1 shows that extended UML models can be translated into different formal notations, so that different tools (e.g., different model checkers) can be employed. We selected Kronos for practical reasons (e.g., it works on different platforms and has a very clean textual interface) although other model checkers could have been used.

The paper is organized as follows: Section 2 accounts very briefly for our time-related extensions of UML (these extensions are thoroughly described elsewhere [2][5][6]). Section 3 describes the formalization of UML statecharts by means of a mapping to Timed Statecharts [7]. Section 4 describes the translation of extended UML models into Kronos automata. Section 5 illustrates the tools developed to support the proposed approach. Section 6 illustrates the application of the proposed approach to an example, namely to the specification of the CSMA/CD Protocol. The application to an industrial case study is also briefly reported. Section 7 accounts for related work, while Section 8 draws some conclusions.

2. Extending UML for real-time

To model real-time systems we need a specification language that is both expressive enough to represent the features of real-time systems and precise enough to support property verification. We have identified some basic extensions to UML, which are described in [2][5][6]. We call the extended notation UML+ throughout the paper. Here we summarize such extensions, which concern the state diagrams of UML. Examples of these extensions can be found in the diagrams that compose the model of the CSMA/CD protocol (see Figure 4 and Figure 9).

A first limit of UML concerns the management of concurrent events. UML [1] forces the events to be treated one at a time (according to the run-to-completion semantic), thus making impossible to specify the behavior of the system when several events occur concurrently. Our extension allows the modeler to associate a set of events to a transition, indicating that the transition is triggered by the concurrent occurrence of the set of events. For instance, the transition from Wait to Trasm (Figure 6) is executed when *send* and *esc* events occur together.

A second problem with UML is that it does not allow the guards associated with transitions to contain references to events. Thus it is impossible, for instance, to specify that a given transition is executed if event A occurs while event B is not occurring. We simply removed this limitation by allowing guards to make reference to events.

In the specification of a real-time system it is often the case that a transition occurs at a time that depends on the occurrence time of another transition. It is therefore necessary to make reference to a given transition's occurrence time. UML does not provide any means to identify transitions in state diagrams, although in sequence diagrams it is possible. We removed this limitation by allowing the modeler to assign labels to transitions, and to make reference to these labels in guards.

The last limitation we identified is probably the most relevant for real-time modeling. In fact the specifications of real-time system are often non-deterministic: e.g., they specify that an event will occur in a given time interval, without saying exactly when. This is typically due to the environment with which the real-time software has to interact. The *After()* construct provided by UML is not suitable for modeling this type of situations. We removed this limitation by allowing the modeler to associate a time interval to a transition, thus indicating that the transition may occur at any time within the interval. For instance, the transition from Retry to Trasm (Figure 4) may happen at any time between 0 and 2**sigma* time units after the entry in state Retry.

3. Formalizing UML Statecharts

The extensions described above indicate the necessity to redefine UML's state diagrams with a formalism that could be easily integrated in the UML meta-model. Among the formalisms for real-time modeling, we consider particularly interesting the Timed Statecharts

(TSs), proposed by Kesten and Pnueli [7]. In the next subsection we present some of TSs' features that are relevant for our purpose.

3.1. Introducing Timed Statecharts

Times Statecharts [7] extend the traditional statecharts by specifying time limits for the execution of transitions. Their semantic is defined with reference to a dense time domain, i.e., the system may deal with arbitrarily close time events. Transitions are classified in two types:

- *Immediate transitions* do not depend on time: they are executed when a triggering event occurs. When no immediate transition is enabled, the time can flow with the state of the system remaining unchanged.
- *Timed transitions* are independent from events. They are associated with a time interval specifying a minimum and a maximum waiting time. The transition cannot be executed before the minimum waiting time and, if no event changes the current state, it must be executed before the maximum waiting time.

Such classification finds a reason in the "weak" semantics of Timed Statecharts, which is based on the fact that every generated event "persists" until the time does not flow. The time may flow only if all the transitions which were enabled by that event have been executed. In this way, several transitions triggered by the same event *e* are executed before the time becomes greater than the time of the occurrence of *e*.

Moreover, Timed Statecharts allow to express negated events among the conditions which can determine the execution of a transition.

Even from this very brief description, it is quite clear that the TSs define mechanisms and constructs that are similar to the ones that were summarized in the previous section. We therefore decided to define the semantics of UML+ in terms of TSs.

In order to make it easier to model the behavior of a system, we also introduced two modifications to the syntax of TSs. First, we allow the modeler to associate both a set of triggering events *and* a time interval with transitions. Then, we have introduced the possibility to refer to a specific transition (state) in order to use its execution (entrance) time to constraint the execution of other transitions. This modifications have a well-defined meaning in TSs, presented in [5].

3.2. Adapting Timed Statecharts to UML

Both Timed Statecharts and UML statecharts derive from the same basic notation. However, there are some features of TSs that do not match very well UML concepts. A first fundamental issue concerns the semantics of events, i.e., how the evolution of the system is determined by events. In UML events are processed sequentially, each event causing the execution of a run-to-completion (RTC) step. The next external event is processed after the current RTC step has completed and the state machine has reached a stable state. In TSs there is no RTC concept, there is no queue of events nor dispatcher. TSs evolve according to a "chain semantics": several transitions in the same step can be performed sequentially. According to our needs, we have decided to adopt the event management principles of Timed Statecharts.

Other differences between the two notations are due to the fact that UML has been provided with several constructs which are not strictly necessary, but make it easier to model object-oriented systems. It is therefore interesting to define the meaning of these constructs by means of TSs. Differences between UML state diagrams and TSs are extensively presented elsewhere [5]; here we summarize only the most relevant ones.

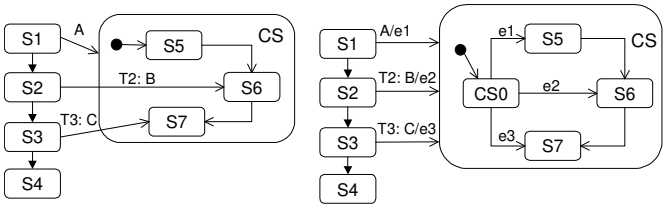
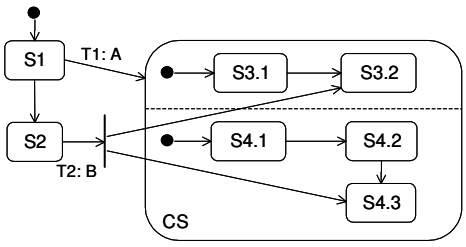


Figure 2. Diagrams with (left) and without (right) inter-level transitions

Inter-level transitions. This kind of transition involves states belonging to different hierarchy levels, or to different composite states. The syntax of Timed Statecharts do not allow inter-level transitions, so they have to be transformed as shown in Figure 2: the diagram in the right part of Figure 2 is equivalent to the diagram in the left part, but does not contain inter-level transitions.

Fork transitions. A fork transition has one origin state and multiple destination states, belonging to different concurrent regions of a composite state (see for instance Figure 3(a)). In the Timed Statecharts, a transition to a composite state implies that the initial sub-states of the concurrent regions are activated.



(a)

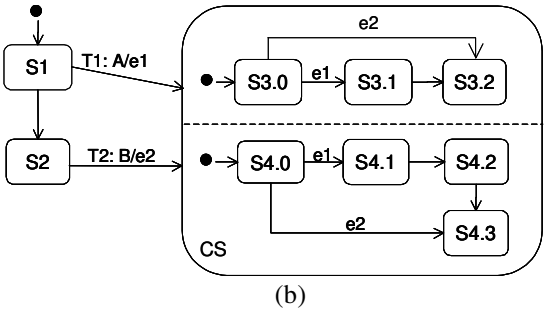


Figure 3. Diagrams with (a) and without (b) concurrent states.

Thus, transitions like the one depicted in Figure 3(a) can be described using TSs as illustrated in Figure 3(b).

- In UML a final state indicates that the evolution in a region of a composite state has reached a conclusion. A composite state can be abandoned only when all its regions reach a final state. TSs do not have final states, but can simulate them quite easily: final states are replaced by normal states; transitions to such states generate completion events. If the final states belong to concurrent regions, only the last region reaching a final state has to generate the completion event: this effect is easily obtained using a counter.
- Entry (and exit) actions are sequences of atomic actions that are executed every time a state is entered (exited). In TSs, these actions can simply be associated with every entering (exiting) transition. Attention must be paid to preserve the correct activation sequence: entry actions are performed after the actions associated with the entering transitions; exit actions are performed before the actions associated with the exiting transitions.

4. Converting UML+ state models into timed automata

We consider the conversion of UML+ state models into equivalent timed automata [9].

As already mentioned, we must cope with the limited expressiveness of Timed Automata (TA) with respect to Timed Statecharts.

The most relevant issues that must be taken into account are the following:

- In timed automata constraints and guards can involve only clock variables, while Timed Statecharts allow boolean expressions on the shared variables.
- Timed Statecharts make distinction between activation events and generated events, while in timed automata there is no such distinction. Moreover Timed Statecharts allow negated events, while TA do not.
- Timed Statecharts' locations can be structured, while Timed Automata have a flat structure.

In the rest of the section we present the basic rules for the process of translation. All the rules will be presented referring to the running example, i.e., the CSMA/CD protocol.

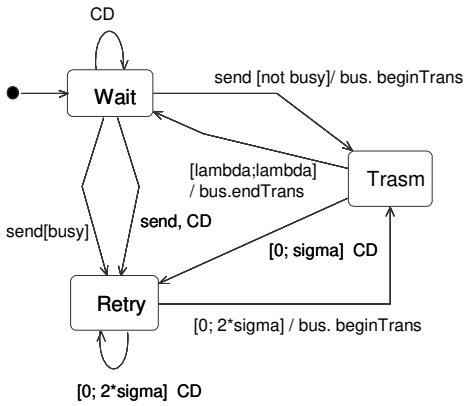


Figure 4. UML+ statechart of a station transmitting according to the CSMA/CD protocol

The transformation begins considering the static view of the model. In Kronos every object is represented by an automaton, while in UML a statechart represents the behavior of *all* the instances of a class. In order to determine how many timed automata have to be generated the translation takes into account UML object diagrams: every instance of a class is associated with a timed automaton. Concurrent states in the state diagram of an object can require to associate more than one automata with this object.

The core of the transformation is performed analyzing the dynamic view of the UML+ model, i.e. the timed statecharts associated to each class. We provide here a first set of translation rules that apply only to non-composite states. First of all, every state of a state diagram is converted into a location of the target automaton. The initial state of the statechart is the initial location of the automaton. Every automaton has an associated clock. In order to deal with transitions, it's useful to classify them in three main classes: *Immediate transitions* with no time constraints, but only triggering events, *Time transitions* with only execution time intervals, and *Mixed condition transitions* having both an execution time interval and triggering events. The latter are executed only if a triggering event occurs within the given time interval.

Let us consider transitions with no guards. The statechart diagram shown in Figure 4 models the behavior of stations sending messages according to the CSMA/CD protocol (hypothesis: $lambda \geq sigma$). We translate such diagram into the corresponding automaton illustrated in Figure 5. This simple example illustrates several rules:

- The clock must be reset whenever a location has an outgoing timed transition. For instance the transition

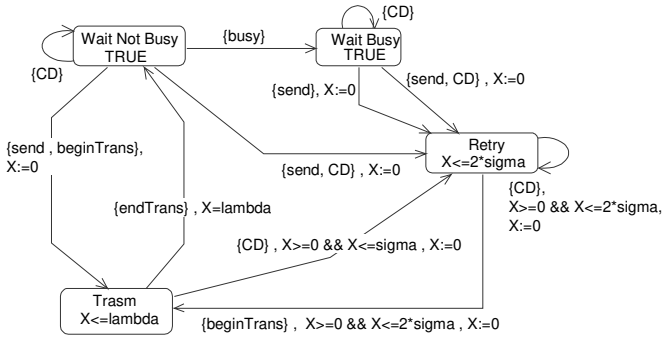


Figure 5. Timed automaton representing a station transmitting according to the CSMA/CD protocol

from state Trasm to state Retry requires clock X to be reset by all the transitions entering Trasm.

- The invariance condition for a state is established according to the analysis of outgoing transitions. States having at least an outgoing timed transition must have an invariance condition of the type $clock \leq MaxUpp$, where *MaxUpp* is the maximum of the upper bounds specified in outgoing transitions. States having no outgoing timed transition must have a TRUE invariance condition.
- Immediate transitions have guard condition = TRUE.
- Transitions having an associated time interval [Low,Upp] have guard condition $clock \geq Low \wedge clock \leq Upp$. If *Low=Upp*, the guard is *clock=Upp*, see for example the transition from Trasm to Wait_Not_Busy in Figure 5.
- Every transition is associated with a set of events that is the union of the triggering events and of the generated events. Negated events are not considered.
- Events generated from a transition and sent to other objects (different from the one owning the transition) are added to the synchronization events (they have to be declared as synchronization events in the automaton associated with the receiving object). The receiving objects are identified by means of the object diagram.

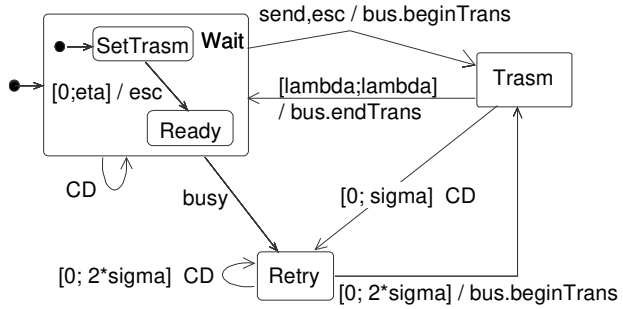


Figure 6. UML+ statechart with nested states

As already mentioned, there are some constructs of UML that cannot be mapped onto any concept of the Kronos notation.

- Negated events cannot be represented in Kronos automata. Thus, statecharts including negated events cannot be correctly translated.
- UML+ allows the specification of variable time intervals associated with transitions. However, Kronos does not allow guards of the type *ck1 # ck2 - ck3* (where # is one of the relations <, ≤, >, ≥, etc.). Therefore, statecharts containing such transitions cannot be translated into equivalent timed automata.

Guard conditions

In general, guard conditions cannot be modeled using timed automata. However, transitions controlled by guards involving boolean variables can be translated in the notation of TA splitting the source state of the transition in different substates, one for each value assumed by the boolean variable. Of course this kind of translation is viable only when guards concern boolean variables or variables which can assume a limited number of values, otherwise the size of the resulting automaton may grow significantly.

Composite sequential states

A whole state machine can be nested in a state that is thus called composite. Of course, the nested state machine can be in only one of its states (called sub-states) at a time. A UML+ statechart containing one or more composite sequential states can be translated into a single timed automaton, having a location for each state or sub-state of the original statechart. It is necessary to define a specific clock variable for every composite sequential state, in order to model time constraints on transitions among sub-states (like for instance the transition between SetTrasm and Ready in Figure 6).

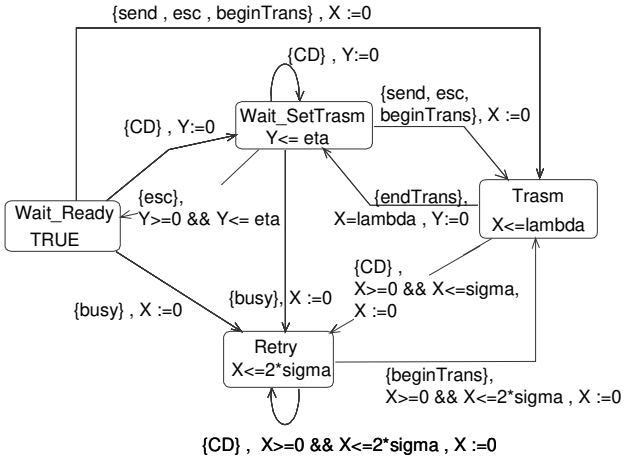


Figure 7. Timed automaton equivalent to the TS given in Figure 6

The translation rules for this case have to assure that every transition which is "factorized" in the original UML+ state diagram is correctly split into several transitions (one for each interested substate). Such rules - described in detail in [5]- are not reported here for space reasons. By applying these rules to the statechart reported in Figure 6, the automaton described in Figure 7 was obtained. Note that the new states corresponding to former sub-states are given composite names, e.g., Wait_Ready corresponds to the substate Ready of state Wait.

Parallel composite states

A unique nested statechart can contain several concurrent states. Orthogonal components of the statechart are processed in parallel. The translation procedure involves the creation of a timed automaton for each orthogonal state (region) of the original state. This set of automata must be activated at the same time: this is obtained by means of a synchronization event (event start). Moreover, when the concurrent composite state is left the corresponding automata must be "disabled", i.e., positioned in a waiting condition. This is done by means of the synchronization event stop.

The translation of each orthogonal region is done following the rules introduced in former sections, with two noticeable exceptions, due to the need of synchronizing sub-states automata when the composite state is entered:

- An initial waiting state is added to every automaton derived from a parallel sub-state. This state is left when the start event is received (which happens at the same time for all the automata of the orthogonal regions).
- For each state of the "sub-automata" an outgoing transition, triggered by the stop event, is added, having destination to the initial waiting state. In this way, when the location corresponding to the parallel composite state is left, all the automata corresponding to its orthogonal regions switch (together) to the waiting situation.

The definition of Timed Statecharts prescribes that when a parallel composite state is entered a statechart is created for each orthogonal region. These statecharts are destroyed when the composite state is left. Unfortunately, Kronos does not allow to create or destroy automata during verifications, thus we have defined a translation procedure that –thanks to the waiting states– makes the behavior of the timed automata undistinguishable from the behavior of the TS. More details can be found in [5].

Translation of multiple instances

In this section we use as a driving example the object diagram reported in Figure 8, that illustrates a system composed of two transmitting stations which compete for a common channel (bus).

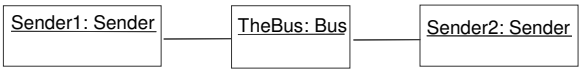


Figure 8. Transmission system: object diagram

The object diagram in Figure 8 contains two instances of class Sender, thus we must be able to distinguish the send events associated with each station. In UML the simplest solution to this problem is to adopt parametric events: e.g., *Send(i)* indicates that the event was originated by the i-th instance. This solution is applicable in general, since instances can always be numbered. Unfortunately, Kronos does not allow parametric events. As a consequence we have to duplicate automata (one for each instance) as well as events (i.e., in this case we will introduce *send1* and *send2*, concerning the first and second instance of Sender, respectively). The translation procedure is as follows:

- For each parametric event a set of events is defined, corresponding to the instances of the classes indicated by the object diagram.
- Synchronization events are associated to the corresponding objects.

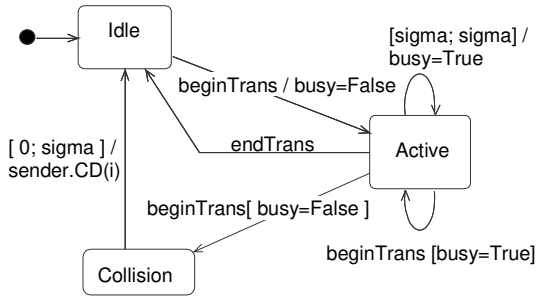


Figure 9. UML+ statechart describing the behaviour of the Bus

Objects (like TheBus) that interact with several instances of the same class (like our Senders) deserve attention. In our example event *CD(i)* defined in Bus statechart (Figure 9) must be translated into CD1 and CD2.

5. Tool support

With respect to Figure 1, currently we are able to define a UML+ model and to translate it in two different formal notations: TRIO [3] and the timed automata of Kronos [4].

The envisaged development process starts with the construction of a model by means of a UML CASE tool. Since existing UML tools are not consistent with UML+ we modified an open source tool (namely ARGO/UML [21]), in order to allow the modeler to write state diagrams conforming to the Timed Statecharts notation. An important feature of the tool concerns the support of XMI

(i.e., the XML-compliant language defined by OMG to represent textually UML models) [22]. The modified ARGO/UML tool saves the UML+ models in an extended XMI file (see [5]) without losing any piece of information that is relevant for the translation into a formal notation. In fact the operation of parsing XMI files is the first step of the translation: in order to perform it successfully the XMI files have to be consistent with the meta-model defined for UML+.

The automation of the approach is completed by the translator (which adopts standard techniques to convert XML files into Java structures) and by the Kronos (or TRIO) tools.

6. Validation

This section describes how the models written in UML+ can be verified using Kronos. In Kronos correctness criteria can be specified as formulas of the timed temporal logic TCTL [10][11], for which Kronos implements model-checking algorithms.

6.1. Verification of the CSMA/CD Protocol

The extended UML diagrams that model the system composed of two transmitters and a bus channel are illustrated in Figure 4 and Figure 9. The corresponding timed automata (T.A.) are described below. A complete description of Kronos syntax can be found in [4], however the syntax is quite intuitive: there is a prologue (indicating the number of states – or locations– and transitions, and the list of clocks and synchronisation events) followed by a description of each state, including an atomic proposition, the state invariant and the set of outgoing transitions (each in the form: `condition => synchronisation events; clock reset; target state`).

The CSMA/CD protocol was used as an example by Kronos' authors [4]. It is interesting to note that the automata that were obtained translating UML diagrams are very similar to those written directly for Kronos.

```
/* T.A. of the transmitting station Sender1 */

   #locs 5
   #trans 11
   #clocks X1 CKAPP1
   #sync BEGINTRANS1 ENDTRANS1 BUSY1 CD1

   loc: 0
   prop: WAIT_NOT_BUSY1
   invar: TRUE
   trans:
   TRUE => SEND1; CKAPP1:=0 ; goto 4
   TRUE => SEND1 CD1; X1:=0 ; goto 2
   TRUE => CD1; ; goto 0
   TRUE => BUSY1; ; goto 3

   loc: 1
   prop: TRASM1
   invar: X1<=808
   trans:
   X1<=52 => CD1; X1:=0 ; goto 2
   X1=808 => ENDTRANS1; ; goto 0

   loc: 2
   prop: RETRY1
   invar: X1<=52
   trans:
   X1<=52 => BEGINTRANS1; X1:=0 ; goto 1
   X1<=52 => CD1; X1:=0 ; goto 2

   loc: 3
   prop: WAIT_BUSY1
   invar: TRUE
   trans:
   TRUE => CD1; ; goto 3
   TRUE => SEND1 CD1; X1:=0 ; goto 2
   TRUE => SEND1; X1:=0 ; goto 2

   loc: 4
   prop: TRANSIENT1
   invar: CKAPP1=0
   trans:
   CKAPP1=0 => BEGINTRANS1; X1:=0 ; goto 1
```

The automaton of the second transmitting station is similar: it has the same locations and transitions, and its own clocks and events (BEGINTRANS2, ENDTRANS2, BUSY2, CD2).

Note that the number of locations defined in the previous Kronos representation does not correspond to the number of states modeled in the corresponding Timed Statechart reported in Figure 4.

In facts, transitions controlled by guards involving the boolean variable *Busy* require the translation of the source state of the transition (i.e. *Wait*) into a set of states, one for each value assumed by *Busy*: *Wait_Busy1* and *Wait_not_Busy1*. This is due to the impossibility of including conditions on boolean variables in guards: only conditions on clocks are allowed in Kronos guards.

Moreover, since in Kronos automata it is not possible to represent transitions associated with both synchronization and local events, auxiliary states (e.g., TRANSIENT1) are generated by the translation, in order to produce an equivalent automaton compliant with Kronos rules.

The Kronos timed automaton representing the behavior of a bus connected with two stations is represented below. Here *lambda* is set to 808 time units, and *sigma* to 26 time units.

```
/* T.A. representing the behavior of the Bus */

   #locs 4
   #trans 23
   #clocks Y
   #sync BEGINTRANS1 ENDTRANS1 BEGINTRANS2
         ENDTRANS2  BUSY1 BUSY2 CD1 CD2

   loc: 0
   prop: IDLE1
   invar: TRUE
```

```
trans:
TRUE => BEGINTRANS1; Y:=0 ; goto 1
TRUE => BEGINTRANS2; Y:=0 ; goto 1

loc: 1
prop: ACTIVE_NOT_BUSY1
invar: Y<=26
trans:
TRUE => BEGINTRANS1; Y:=0 ; goto 2
TRUE => ENDTRANS1; ; goto 0
Y=26 => BUSY1 BUSY2; ; goto 3
TRUE => BEGINTRANS2; Y:=0 ; goto 2
TRUE => ENDTRANS2; ; goto 0

loc: 2
prop: COLLISION1
invar: Y<=26
trans:
Y<=26 => CD1 CD2; ; goto 0

loc: 3
prop: ACTIVE_BUSY1
invar: TRUE
trans:
TRUE => BEGINTRANS1; ; goto 3
TRUE => ENDTRANS1; ; goto 0
TRUE => BEGINTRANS2; ; goto 3
TRUE => ENDTRANS2; ; goto 0
```

The model has been verified in order to check three desired real-time properties.

Non-Zenoness

The condition of non-Zenoness holds when the system does not contain states that prevent time from diverging.

In Kronos this property is expressed in TCTL as follows:

```
init impl ab ( ed{=1} true )
```

Kronos verified (by means of backward analysis) that this property actually holds for the given model.

Bounded delay for collision detection

The second property concerns the time required by senders to become aware of a collision. It is required that when a collision occurs, because two distinct senders are transmitting simultaneously, they both detect the collision at most sigma time units later.

The property is expressed in TCTL as follows:

```
(TRASM1 and TRASM2) impl ad{<=26}
   (RETRY1 and RETRY2)
```

Kronos (applying a forward analysis) verifies that the property holds.

Successful transmission

Finally we want to assure that whenever one of the senders begins transmitting, there exists a behavior that leads to a successful transmission without collisions, such that the message is delivered in a time lambda.

We can formally state this property in TCTL as follows (for the first transmitter):

```
init impl ab ((TRASM1 and X1=0 and
  ab(not COLLISION1)) impl ed{<=52}
    (ad{<=808} WAIT_NOT_BUSY1))
```

Kronos (applying a forward analysis) verifies that the property holds.

6.2. Verification of the ACL/8000 motor control software

The ACL/8000 is a device used to analyse samples of human blood according to specific medical procedures. Although its software is composed of several functional blocks, we considered only the management of the motor that drives the rotor where the samples are positioned during the data acquisition phase. The analysis is performed by several sensors, that take measures of the samples while the rotor is spinning. The behaviour of the machine is described in detail in [5] by means of UML+ models. Here we give only a brief overview, concentrating on the time constraints that characterize the model of the system. The ACL rotor is driven by a stepping motor, where every pulse produced by a programmable clock causes the motor to make a step. The data acquisition phase is characterized by a specific rotation speed of the disk. According to the specification of the ACL machine, it is necessary to reach this speed gradually, following a precise schema of acceleration steps. Accelerations are obtained by gradually increasing the frequency of pulses produced by the clock. Decelerations are obtained in a similar way, decreasing gradually the frequency of pulses. A first real-time constraint is that the motor management operations must be carried out in the interval between two consecutive pulses. Otherwise it could happen that a new step is requested while the previous one is not yet completed. The acquisition is performed by means of a sequence of operations, according to a given time diagram. Every acquisition must be done when the sample slot has its center under the sensor (this is needed to maximize the precision of the measure). Therefore, a second strict real-time constraint is that all the data acquisition operations must be completed in a limited time, such that every sample can be correctly analyzed by the sensor, without loosing any measure. According to the system requirements, we modeled the ACL motor control software using UML+. The class diagram includes two relevant classes, “Acquisition” and “MotorController”: the first one models all the operations necessary to acquire data from blood samples, while the second one drives and controls the disk rotor according to the pulses generated by a timer.

The model has to satisfy properties that do not involve the states of several different objects, so we can express constraints based on single states of the statecharts. By applying to the specifications of the ACL/8000 the translation techniques described previously, four Kronos automata were obtained. The TCTL expressions defined for these automata follow.

Non-Zenoness

The condition of non-Zenoness is the same as in the CSMA/CD protocol. Kronos verified that the system is non-Zeno.

Slot data reading

The following condition states that the data acquisition for each slot must not take more than 20 time units.

```
init impl ab (SLOTCENTERWAIT1 impl
(ad{<=20} READY_TO_READ1))
```

This property is also certified by Kronos (via forward analysis).

Motor control time

This condition states that the motor management operations must be carried out in the interval between two consecutive TimeOut signals coming from the timer. In this way we prevent the execution of a new step while the previous one is not yet completed.

```
init impl ab (SETMOTOR impl (ad{<=TMIN}
(ACCELERATION or DECELERATION or
STEADYSPEED)))
```

Kronos checked (via forward analysis) that the automata of ACL/8000 verifies this property (with TMIN=4 time units).

7. Related work

Several researchers adopted approaches based on translation, in order to obtain a specification that can be used as the input of a model checker. An example is the translation [13] from statecharts [15] to Promela that allows to verify the properties of statecharts by means of the model checker SPIN [16]. Our work adopts a similar approach, but is significantly different as long as we consider UML models (composed of class, object and state diagrams) instead of just statecharts. Moreover we can take into account time-related properties, which is not easy using plain UML and Promela.

Selic et al. [23] have proposed the Real-Time Object-Oriented Modeling (ROOM) language, which has been merged with standard UML to form a proposal of 'UML for real-time'. This language is being used in several tools, like Rose for Real-Time and Anylogic. ROOM favours the design and coding activities, at the expenses of the specification phase. The language provides rich information on the structure of the system thanks to the facilities provided to describe active objects and their communications, and tools are available to translate the models into executable code which behaves like the models. The limits of ROOM are in the lack of constructs to describe non-trivial time constraints, as well as in the lack of formal semantics (semantics is hardwired in the ROOM virtual machine). As a consequence, a ROOM model cannot be analysed in order to prove whether the model has desirable properties such as safety.

ROOM notation is now converging in UML 2.0 (still in draft version).

Silva Bastos and D'Almeida Sanchez proposed the translation of UML models into SDL [14]. This kind of translation is however performed in a semi-automatic way *"because UML models are informal and miss definitions that are normally required by SDL models"*. Our approach is clearly different as we define a formal semantics for UML by means of Timed Statecharts. Thus the translation of UML+ models is completely automated, and no "interpretation" is required, so that misinterpretation is not possible.

vUML [17] is a tool that translates UML models into Promela, so that the behavioral properties of the models can be verified by means of SPIN. Our approach is very close to the one implemented in vUML, since in both cases the user is not required to know/understand the model checker and its language (SPIN/Promela or Kronos/timed automata). A difference between the two approaches concerns the semantics of UML. While we re-define the meaning of UML statecharts in terms of Timed Statecharts, Lilius and Porres Paltor provide their own definition of UML state machine (keeping the RTC semantics). Finally, a strong limitation of vUML is that it is not able to verify linear temporal logic formulas, i.e., it does not deal with time, which is not a native concept in standard UML.

A translation of a suitable extension of UML into an axiomatic formal description is proposed by Alagar et al. [18]. A tool developed by the same authors can be used to simulate the system, and the PVS theorem prover can be used to prove properties of the system. Of course in this case it is required that the user has the (non trivial) ability to use PVS.

Finally, an effective approach to real-time system development and verification is provided by the Taxys tool [19]. Taxys works with Esterel [20] programs annotated with temporal constraints, and contains the Kronos model checker for model analysis. The Esterel programs can be obtained from a variant of UML, which features synchronous semantics by replacing StateCharts with SyncCharts. These charts are supported by tools (Esterel Studio).

8. Conclusions

At the International Workshop on Industrial-strength Formal Techniques 1998 the panel on integrating formal and informal specification techniques [12] provided a quite lucid picture of the situation concerning the practice of software specification in industry:

There is a strong demand from industry for rigor and formality. Academic research and tools vendors are asked to provide useful, powerful, cost-effective methods. Application of mathematically-based formal techniques requires much effort and skill on the part of the developer, and not many are willing to invest to acquire such skills. A technique that allows developers to reap the

benefits of rigorous analysis while using notations they are familiar with is more likely to be used in practice.

The work presented here is -in our opinion- perfectly in line with that statement. We provide developers with the means to write and formally verify the specifications of real-time systems without big investments in training on formal methods. In this way we get rid of the main hindrances to the exploitation of formal methods in industry. However, some work still remains to be done, in order to make the proposed approach effectively applicable in industry:

- Although Kronos automata are automatically generated, at the moment the user has to write the TCTL formulas to be verified. This is not always immediate: a method to facilitate this activity has to be defined.
- Another problem concerns the presentation of negative results to the user. When a property is not verified, it is generally very useful to provide a counter example. In our case this means translating the output from Kronos in a language that the user can easily understand. The possibility to provide counterexamples by means of sequence diagrams will be explored.
- Finally, we have to link the proposed approach -which is oriented to the specification phase - with the implementation phase. In pursuing this goal we have again to consider the constraints posed by industry. We are therefore studying how to convert UML+ specifications into UML-RT, the notation employed by the most popular development environments for object-oriented real-time systems.

Among future activities is also the formal description of the translation principles and decisions, which will be the subject of a future article.

Acknowledgements

The work reported here was partly funded by MIUR in the context of the DESS (Software Development Process for Real-Time Embedded Software Systems) ITEA project (http://www.dess-itea.org).

10. References

[1] OMG Unified Modeling Language Specification Version 1.4., September 2001.

[2] Lavazza, L., Quaroni, G., Venturelli, M.: Combining UML and formal notations for modeling real-time systems. In: Gruhn, V. (ed.): Proceedings of ESEC/FSE 2001, Wien, September 10-14, 2001. ACM Press, 2001.

[3] Ghezzi C., Mandrioli D., Morzenti A.: TRIO, a logic language for executable specifications of real-time systems. The Journal of Systems and Software, Vol.12, n.2. Elsevier Science, May 1990.

[4] Yovine, S.: Kronos: A verification tool for real-time systems. International Journal of Software Tools for Technology Transfer, Vol.1, N.1/2. Springer, Oct. 1997.

[5] Del Bianco, V., Lavazza, L., Mauri, M.: An introduction to the DESS approach to the specification of real-time software. Technical Report RT01002. CEFRIEL, 2001. Available from http://www.cefriel.it, "WISE" section.

[6] Del Bianco, V.,Lavazza, L., Mauri, M.: A Formalization of UML Statecharts for Real-Time Software Modeling. The Sixth Biennial World Conference On Integrated Design Process Technology (IDPT 2002). Pasadena, June 2002.

[7] Kesten, Y., Pnueli, A.: Timed and Hybrid Statecharts and their Textual Representation. In: Vytopil, J. (ed.): Proc. of FTRTFT92. LNCS, Vol.571. Springer-Verlag, , 1992.

[8] Tanenbaum, A. S.: Computer Networks, second edition. Prentice-Hall, 1989.

[9] Alur, R., Dill., D. L.: A theory of timed automata. Theoretical Computer Science, n.126, 1994, pp. 183-235.

[10] Alur, R., Courcoubetis, C., Dill, D. L.: Model checking in dense real time. Information and Computation, Vol.104, N.1., 1993.

[11] Henzinger, T. A., Nicollin, X., Sifakis, J., Yovine S.: Symbolic model checking for real-time systems. Information and Computation, Vol.111, N.2., 1994.

[12] Bruel, J.M., Cheng, B., Easterbrook, S., France, R. B., Rumpe, B.: Integrating Formal and Informal Specification Techniques. Why? How? Panel session, 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, 20-23 October 1998.

[13] Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing Statecharts in PROMELA/SPIN. 1998. Proc. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, 1999, pp. 90 –101.

[14] Silva Bastos, S.J., D'Almeida Sanchez, M.L.: Modeling Real Time Systems from Object-Oriented Models. IEEE Real-Time Embedded System Workshop, 2001.

[15] Harel, D.: Statecharts: A Visual Formulation for Complex Systems. Science of Computer Programming Vol.8, N.3., 1987, pp. 231-274.

[16] Holzmann, G.J.: The Spin Model Checker. IEEE Trans. on Software Engineering, Vol.23,N.5, May 1997, pp. 279-295.

[17] Lilius, J., Porres Paltor, I.: vUML: a tool for verfying UML models. ASE'99, 1999, pp. 255-258.

[18] Alagar, V.S., Muthiayen, D., Pompeo, F.: From behavioral specification to axiomatic description of real-time reactive systems. Fifth IEEE Real-Time Technology and Applications Symposium, Work-In-Progress Session. Vancouver, BC, June 1999.

[19] Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., Yovine, S.: TAXYS: a Tool for the Development and Verification of Real-Time Embedded Systems. Computer Aided Verification, CAV'01, Paris, July 2001.

[20] Berry, G., Gonthier, G.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming, Vol.19, N.2. (1992) 87-152.

[21] http://argouml.tigris.org/

[22] OMG XML Metadata Interchange Specification, Version 1.1, November 2000.

[23] Selic B., Gullekson G., Ward P.T., Real-Time Object-Oriented Modeling, Wiley, 1999.