On the fly model checking of communicating UML State Machines

Stefania Gnesi
ISTI - CNR
stefania.gnesi@isti.cnr.it

Franco Mazzanti
ISTI - CNR
franco.mazzanti@isti.cnr.it

Abstract[1]

In this paper we present an ``on the fly" model checker for the verification of the dynamic behavior of UML models seen as a set of communicating state machines. The logic supported by the tool is an extension of the action based branching time temporal logic μ-ACTL and has the power of full μ-calculus. Early results on the application of this model checker to a case study have been also reported.

1. Introduction

The Unified Modeling Language (UML) is a graphical modeling language for object-oriented software and systems [1,2] It has been specifically designed for visualizing, specifying, constructing and documenting several aspects of - or views on - systems. Different diagrams are used for the description of the different views.

In this paper, our aim is to define an environment for the formal verification of behavioral properties of systems modeled by a fixed number of message exchanging active objects, each of them described by a UML Statechart Diagram.

The paper proposes the use of a formal verification technique, namely model checking [3], to verify the conformance of a design with respect to desired properties. Since model checking suffers the so called "State Space Explosion" problem, that can arise when a system is composed of several parallel subsystems we have developed an on the fly algorithm for model checking UML communicating state machines. This algorithm is able to check the validity of a formula without generating the global model of the system bypassing the state explosion problem that makes other verification tools inapplicable.

UMC (UML on the fly Model Checker) is the result of a project recently started at ISTI with the purpose of doing some experimentation in three different directions.

First, we are interested in testing the appropriateness of the UML methodology (and in particular UML statecharts) for the design and the specification of the dynamic behavior of a system. The question here is if we can find a subset of UML which from one side is reasonably well defined and powerful enough to allow a specification to be written without too much effort, and from the other side is still not to complex to allow a verification/analysis tools to be developed for it.

As a second theme, we are interested in investigating the kind of user interface that might help a non-expert user in taking advantage of formal specifications and verification techniques. Nowadays formal methods still encounter many obstacles in being widely used and accepted, and we believe that the kind of human interface they usually provide does play a main role in this difficulty. UML, with all its standard graphic and yet standard layout features, might present an opportunity to convey to the user a more friendly way of specifying models and their properties.

Thirdly, we have already started an interesting experience (which is still ongoing) in designing and implementing an "on the fly" model checker for full μ-calculus for networks of automata, which has resulted in the integration of the FMC tool inside the JACK environment [4]. We are now interested in exploring the advantages given by the same "on the fly" approach for the verification of the dynamic behavior of UML models.

UMC, indeed, is essentially an experiment in the design of an integrated tool for the construction, the exploration, the analysis and the verification of the dynamic behavior of UML models described as a set of communicating state machines.

The present paper is organized as follows: in Section 2 we discuss the problems we have encountered in defining a formal verification environment for UML Statechart Diagrams, related in particular to the description of the formal model on which to perform the verification of behavioral properties; in Section 3 we debate the assumptions and limitations we have taken into account in the implementation of UMC for bypassing some of the intrinsic difficulties imposed by the complexity of UML; Section 4 presents the logical framework used to express behavioral properties; in Section 5 the on the fly algorithm is outlined. The application of UMC to a case study is shown in Section 6. We conclude the paper with a comparison with related works together with some conclusions (Section 7).

2. UML modeling difficulties

UML is a semi-formal language, since its syntax and static semantics (the model elements, their interconnection and well-formedness) are defined formally, but its dynamic semantics is specified only

[1] This work is partially supported by the AGILE project IST-2001-32747 by the proactive Initiative on Global Computing

informally. As already hinted in the introduction, a problem is that full UML is a very complex framework, which at a first glance looks even too complex for what might seem to be the needs and the capabilities of a formal verification environment. Therefore have the need to identify a more tractable subset of features of UML, which still allow in a reasonable way a well-defined and consistent definition of a model.

Several approaches have been proposed in the literature for the definition of a formal semantics of UML Statechart Diagrams, e.g. [5, 6, 7]; starting from them in the UMC project the process of UML subset identification has proceeded in a bottom up way, adding in the time more and more features while the specification needs were being better identified and more experience was gained in the project itself. The difficulty of achieving a really complete support of UML features can also be observed in the current state of art of UML supporting tools. For example, we have not yet been able to find a free or inexpensive implementation allowing the design of statecharts with all their features, and even expensive commercial environments are often lacking some of them.

A problem encountered in modeling the dynamic semantics of UML statecharts is originated by the many aspects which UML explicitly and intentionally leaves as "implementation dependent" or "implementation defined", or simply "not specified by the standard". Examples of these "intentionally not specified" aspects are the precise semantics of event queues, the precise semantics of signal transmission, the precise semantics of parallel evolutions of multiple state machine, the set of actions allowed inside a transition, and so on. For developing a formal verification tool, the only way to deal with these aspects is either to make some own implementation choice or to handle the aspect as a parametric aspect, which could be in some way defined according to some user choice.

Another problem is that we may also find "partially defined" aspects which look like just incomplete or imprecise definition of the dynamic semantics of a state machine, without any hint on the fact that this under-specification is intentional or not. Or we may encounter contradictory / inconsistent aspects of the UML definition (e.g. the behavior in presence of composite, dynamic choice transitions), or aspects defined in a particularly ambiguous way (e.g. the handling of completion transitions inside a concurrent state machine).

Again a formal verification tool is forced to make its own implementation choices, but in this case the question remain on whether or not the correct choice has been made, of if a different choice left implicit behind the UML definition should have been made (an example of that being the definition of “priority” for a join transition when all the sources have the same depth).

Another difficulty, which is intrinsically related to the UML statecharts design, is that they may have a potentially infinite number of states. This is a consequence of the fact that he definition of a state machine includes a potentially unbounded queue of events, and maybe also because the data types used and updated by the transition actions can be potentially infinite data types.

A final additional, not strictly technical, difficulty is related to the fact that it is not evident what could / should be considered as “observable” of a system configuration or of a system evolution from the point of view of the abstract semantics of a state machine. At least the runtime values of the object attributes or the signals generated during the system evolutions should probably be considered as "observable". Already this minimal assumption implies that a single system evolution may contain more than one atomic "observable event" (differently from what usually happens in most process algebras) and that a system configuration snapshot cannot not be considered simply a "black box”, because we must allow the visibility of at least the object attributes.

3. UMC: Assumptions and limitations

The difficulties mentioned in Section 2, and our desire to proceed in bottom up process by adding more and more features to successive running versions of the UMC prototype has led us to a certain number of assumptions and limitations built in the current version. As more experience is gained in the project assumptions might change and limitations might be removed.

3.1 UMC Assumptions

- The whole sequence of actions constituting the “actions part” of statechart transition, is supposed to be executed as an indivisible atomic activity, i.e. two parallel statechart transitions, fireable together in the current state-machine configuration, cannot interfere one with the other, but they are executed in a sequential way (in any order).
- Given a model constituted by more than one state machine, a system evolution is constituted by any single evolution of any single state machine. I.e. state-machine evolutions are considered atomic and indivisible.
- The propagation of signals inside a state machine and among state machines is considered instantaneous, and loss free.
- The events queue associated with a state machine is a FIFO queue.
- The relative priority of a join transition is always well defined and statically fixed.

3.2 UMC Limitations

- Events: Only asynchronous signals are supported. Call events, time events, change events, events deferring are not supported.

- States: Only initial, final, composite and parallel states are supported. State Internal transitions and State Enter / Exit/ Do activities are not supported. History states, Sync states, Choice pseudostates are not supported.
- Transitions: Only direct, fork and join transitions are supported. Initial default transitions are requied not to have actions; static and dynamic choice transitions are not supported. Completion transitions cannot appear in more than one region of concurrent state.
- Other: Sub-machines are not supported. Actions can only be simple assignments or sending of signals. The only data type for variables and signal parameters is constituted by integer values.

4. μ-ACTL+ and UML state machines

In the previous sections we have analyzed the problems related to the definition of a formal model for UML statecharts; now we introduce the logical framework used to express the behavioral properties we wish to verify on the model associated to a UML specification. The logic we consider is μ-ACTL [8], an extension with a fixed-point operator of the action based logic ACTL defined in [9], whose expressive power is the same of full modal μ-calculus [10]. μ-ACTL is a branching time temporal logic suitable to express properties of communicating systems whose behavior is characterized by the actions they perform and whose semantics is defined by means of Labeled Transition Systems (LTSs). The logic can be used to define both *liveness* (something good eventually happen) and *safety* (nothing bad can happen) properties of reactive systems (with and without *fairness* constraints).

A μ-ACTL formula can be built with the following syntax:

$$\begin{aligned}\Phi ::= {} & \texttt{true} \mid \Phi \wedge \Phi \mid {\sim} \Phi \mid \\ & \texttt{EX}\{\chi\}\,\Phi \mid \texttt{EX}\{\tau\}\,\Phi \mid \texttt{min Z:}\Phi\end{aligned}$$

$$\chi ::= \texttt{true} \mid \texttt{a} \mid \chi \wedge \chi \mid {\sim} \chi$$

where χ are action formulae that intuitively express sets of actions and "a" is an observable action belonging to a finite alphabet of actions. τ represents instead a not visible action (not belonging to the finite alphabet of actions).

The formal semantic of μ-ACTL is given over Labeled Transition Systems. Informally, a formula is true on an LTS, if the sequence of actions of the LTS verifies what the formula states. We hence say that the formula:

$\texttt{EX}\ \{\chi\}\Phi$ is true in a state S of an LTS when Φ is true in a successive state of S, reached by an action satisfying χ. The formula $\texttt{EX}\{\tau\}\Phi$ is true in S when Φ is true in successive state of S reached by a τ action.

Starting from the basic μ-ACTL operators, some derived ones can be defined in the usual way, among them: the Hennesy-Milner [11] state modalities `[ ]` and `< >`, the `EF` (eventually) and `AG` (always) and until formulae, and the $\texttt{max Z:}\Phi$ (maximal fixed point). We refer the interested reader to [8] for a more detailed description of μ-ACTL.

In this paper μ-ACTL is extended to make possible observations on UML model evolutions and assertions on explicit local state variables of UML state machines. We will call this logic μ-ACTL+. Unlike most process algebras, UML state machines have an explicit set of objects attributes. This raises the question whether or not we should allow the user to specify system requirements (e.g. logic formulae), which take into account also the values of attributes during the execution. We have decided to allow this kind of internal visibility inside the system configurations and this has been achieved adding to the μ-ACTL logic a special `ASSERT` state predicate.

Moreover χformulaebecome evolution predicates, and "a" the observation of a signal event being sent to a target object (in the syntax below square parenthesis are used to denote optional parts).

$$\begin{aligned}\Phi ::= {} & \texttt{true} \mid \Phi \wedge \Phi \mid \neg\, \Phi \mid \texttt{EX}\{\chi\}\Phi \mid \texttt{EX}\{\tau\}\Phi \mid \\ & \texttt{min Z: } \Phi \mid \texttt{ASSERT(VAR=value)}\end{aligned}$$

$$\chi ::= \texttt{true} \mid \texttt{[target.]event[(args)]} \mid \chi \wedge \chi \mid {\sim}\chi$$

`ASSERT(VAR=value)`is true if and only if in the current configuration the attribute `VAR` has value equal to "value".

Following the above syntax we will write using μ-ACTL+ formulae such as: `EX{Chart.my_event} true` that means: in the current configuration the system can perform an evolution in which a state machine sends the signal `my_event` to the state machine `Chart`.

Or the formula: `EX{my_event(3)} true` that means: in the current configuration the system can perform an evolution in which a state machine sends the signal `my_event(3)`to some other state machine.

The action expression τ is supposed to match instead any system evolution which does not send any signal.

As a last example, let us consider the following formula: `AG ((EX{my_event}true) -> ASSERT(Object.Attribute=Value))` which means that the signal `my_event` can be sent, only when the specified attribute of the specified object has the specified value.

5. The on the fly model-checking approach

Our approach to the "on the fly" model checking of a μ-ACTL+ logic formula has been initially presented in [12]. In that case the system to be verified was defined by a network of synchronized agents working in parallel. The model checker, named FMC, was included in Jack [4], an environment based on the use of process algebras, automata and temporal logic formalisms, supporting many phases of the system development process. The model checker presented here, UMC, is based on the same ideas of FMC, but working over a set of communicating (i.e. exchanging signals) UML State machines.

Even though the code for both tools FMC and UMC has been almost completely rewritten several times, the underlying logic schema has remained the same.

The basic idea behind FMC and UMC is that, given a system state, the validity of a formula on that state can be evaluated analyzing the transitions allowed in that state, and analyzing the validity of some sub-formula in only some of the next reachable states, in a recursive way, as shown by the following simplified schema (E: Env represents the "current context" in which a given subformula is evaluated, which gives a precise meaning (in terms of already started computations) to the free variables which appear in it):

```
Evaluate (F: Formula, E: Env, S: State) is
  if we have already done this computation
      and the result is available then
    return the already known result
  elsif we are are already trying
        to compute F in S with E then
    return true or false depending on maximum
           or  minimum fixed point semantics
  else
      Keep track of the fact that we are trying
        to  compute F in S with E
        (e.g. push the pair (F,E,S) in a stack)
     for each sub-formula F'and next state S'
         which needs to be computed loop
       call recursively Evaluate (F',E',S');
       if the result of Evaluate (F',E',S')
            is sufficient to establish the
             result of evaluate (F,E,S) then
         exit from the loop;
       end if
     end loop
     Now we have in any case a final result.
     Keep track of the fact that we are no
       longer trying to compute F in S with E;
       (e.g. pop (F,E,S) from the stack)
     Possibly keep track of the performed
       computation and result
       (e.g. push (F,E,S,result) in a hash
        table to speedup further future
        recomputations)
     return the final result
  end if
end Evaluate;
```

The big advantage of the on-the-fly approach to model checking is that hopefully only a fragment of the overall state space might need to be generated and analysed to be able to produce the correct result (cf. [13,14,15,16]). This approach seems particularly promising when applied to UML state machines (or groups of communicating state machines) because it can easily be extended also to the case of potentially infinite state space, as it may happen for UML state machines. Indeed, a problem of the above evaluation schema is that, in case of infinite state machines, it might fail to produce a result even for some cases in which a result might be produced in a finite number of steps. This is a consequence of the "depth first" recursive structure of algorithm. The solution taken to solve this problem consists in adopting a bounded model checking approach [17], i.e. the evaluation is started assuming a certain value as maximum depth limit of the evaluation. In this case if a result of the evaluation a formula is given inside the requested depth, then the result holds for the whole system, otherwise the depth limit is increased and the evaluation restarted (or better, resumed from from the points in which it had been halted when hitting the previous depth limit).

This approach, initially introduced in UMC to overcome the problem of infinite state machines, happens to be quite useful also for another reason. Setting a small initial depth limit, and a small automatic increment of it at each re-evaluation failure, when we finally find a result we can have a reasonable (almost minimal) explanation for it, and this could be very useful also in the case of finite states machines.

6. UMC and the Airport Case Study

Let us consider, as a toy example, a system constituted by two airports, two passengers (one at each airport), and a plane. The plane is supposed to carry at exactly one passenger and flies (if it has passengers) between the two airports. Before boarding the plane the passenger must perform the check in. After the plane is arrived at the destination airport, the passenger deplanes. We contemplate only one observable action performed by the passenger during the flight, namely the consumption of a meal. Let us consider a scenario where a `Passenger` boards a plane in `Airport1`, flies to `Airport2` and deplanes there.

6.1 Model Definition

The model under investigation is specified by a textual description of a set of UML statechart diagrams, one for each class of objects which constitute the system, and by a set of object instatiations. This description of the classes of the model can be directly edited in a simple textual form, or extracted from an UML model description given in the XMI format. In Figure 1, and 2 we show the statechart diagrams of classes Plane, Passenger and Airport. In Figure 3 we show the UMC textual notation used to represent the Passenger statechart.

6.2 Model exploration

Once a description of a system has been successfully loaded its initial configuration can be graphically visualized as a set of statecharts, in which the currently active substates are highlighted. The allowed evolutions from the current configurations are shown and they can be manually selected to visualize the subsequent configurations reachable by applying them. The LTS representing all the possible system evolution steps can be graphically visualized as a click sensitive graph. Clicking over a node leads to the visualization of the structure of the node (its active, states, the values or its variables, the content of the events queue).

The whole formal model for our airport scenario is an LTS containing 240 states and 595 transitions.

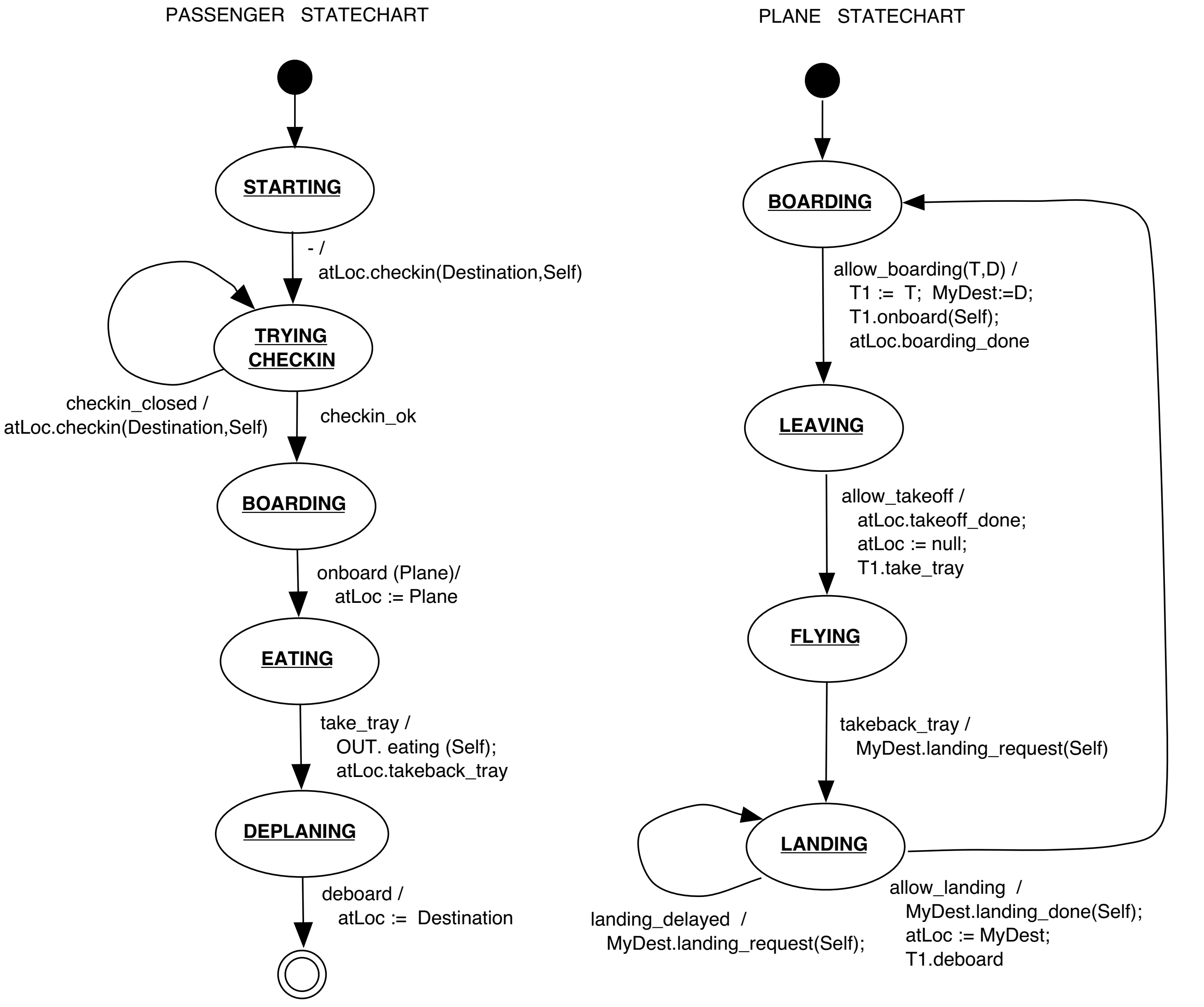

Figure 1: Plane and Passenger Statecharts

6.3 Model verification

After a system has been loaded, it is possible to enter a μ-ACTL+ formula to be evaluated over its model. Once a formula has been edited its on the fly evaluation starts.

In our case study we could be interested, for example, to check if it is true that a passenger can eat (signal `eating(Traveler1))`, only when the plane is flying (`Plane1.atLoc=null)`. This can be done by checking the truth of following μ-ACTL+ formula:

```
AG((EX{eating(traveler1)}true) ->
     (ASSERT(Plane1.atLoc=Plane1)&
       ASSERT(Plane1.atLoc=null))
```

The evaluation of this formula asks that the whole reachable state space has to be traversed, not beneficing in this way of the on the fly approach. In the next formula, instead, the advantages of the on-the-fly approach are exploited, returning the correct result after the analysis of a small fragment of the state-space.

```
max Z: EX{~ eating} Z
```

The above formula asks if there exists an infinite path along which nobody eats, and the answer is that such an infinite path does exists; if requested a path witnessing the assertion is indicated.

Let us consider now the formula expressing the property: it is always true that, if flying, Plane1 will eventually land.

```
AG( ASSERT(Plane1.atLoc=null)  ->
    AF ~ASSERT (Plane1.atLoc=null) )
```

This formula is false on our system and at a first glance it may appear surprising.

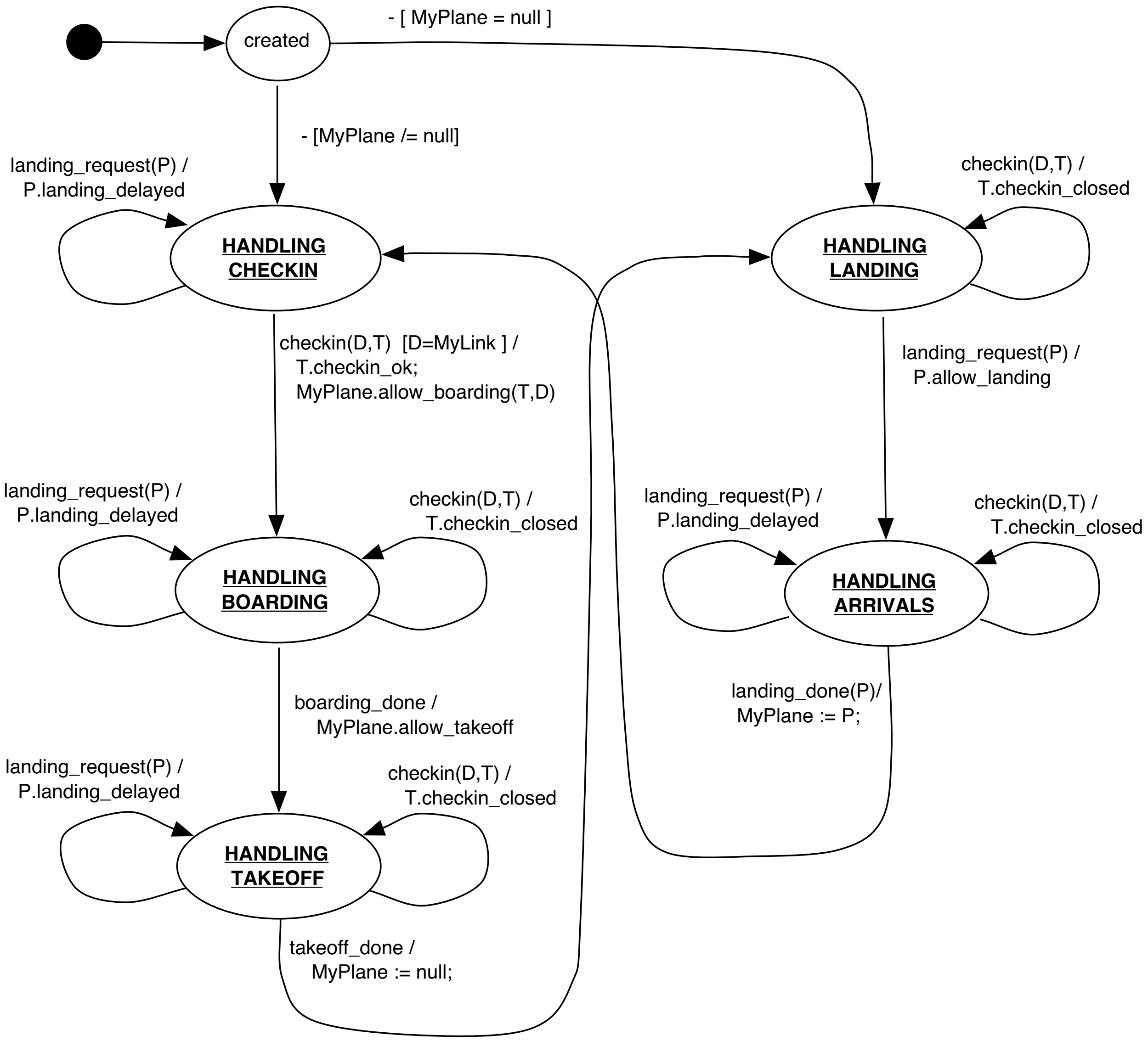

Figure 2: Airport Statechart

```
Class Passenger
Vars: atLoc:obj, Destination:obj
Events: checkin_ok, checkin_closed, onboard(P:obj), take_tray, deboard
State Top = STARTING, TRYING_CHECKIN, BOARDING, FLYING, DEPLANING, FINAL
Transitions:
  STARTING -( -/atLoc.checkin(Destination,Self) )-> TRYING_CHECKIN
  TRYING_CHECKIN -( checkin_closed / atLoc.checkin(Destination,Self) )-> TRYING_CHECKIN
  TRYING_CHECKIN -( checkin_ok ) -> BOARDING
  BOARDING -( onboard(P) /  atLoc := P )-> FLYING
  FLYING -( take_tray / OUT.eating(Self); atLoc.takeback_tray )-> DEPLANING
  DEPLANING -( deboard / atLoc := Destination )-> FINAL
```

Figure 3: textual notation for passenger statechart

Once the plane is moving from Airport1 to Airport2, it might happen that the objects Airport2 and Traveler2, interacting together, monopolize the system evolutions cycling in their attempt to try to check-in and refusing it.

Under these circumstances the unfairness of the system scheduler might prevent the Plane1 to signal its landing request. Fairness issues are likely to play a relevant role in UML as constraints for UML verifications, and the power of full μ-ACTL+ is needed to express these properties in a general way.

After a formula has been evaluated, a complete explanation, in the style of the counterexample facilities of classical model checking tools, can be visualized (both a graphic and in textual format) showing the logic steps with have led to the result. Each step of the explanation is essentially an assertion of the kind:

```
The formula:
 <F1> is true /false in configuration <C1>
    because
      <F2> is true /false <C2>
and because
      <F3> is true /false in configuration <C3>
and because
    … …
```

From each step of this explanation we can directly visualize the set of system evolution starting from the configuration to which the step refer.

7. Related Works and Conclusions

Linear-time model checking of UML Statechart Diagrams is addressed in [18], [19] and [20]. In [21] a simple (branching time) model-checking approach to the formal verification of UML Statechart Diagrams was presented exploiting the "classical" model checking facilities provided by the AMC model checker available in JACK. We are currently aware of three available tools for model checking UML systems described as sets of communicating state machines. HUGO [22] and *v*UML [23] take the approach of translating the model into the Promela language using SPIN [24] as the underlying verification engine. We have not had direct experience with these tools, but clearly in this case the properties to be verified need to be mapped into LTL logic. While *v*UML is restricted to deadlock checking, HUGO is mainly intended to verify whether certain specified collaborations are indeed feasible for a set of UML state machines. In both cases, the UML coverage of the tools is wider than ours because it includes UML call operations, history states, and internal state activities. A timed version of HUGO (called HUGO/RT [25]) has also been developed, which maps into the UPPAAL verification engine, instead than into SPIN.

A third interesting approach is that one adopted in the ongoing UMLAUT [26,] project. In this case an UML execution engine has been developed, adopting the Open Caesar standard interface of the CADP environment. In this way all the CADP [27] verifications tools (including the "on the fly" Evaluator tool [28]) can be applied also to this new engine.

As far as we now, UMC is the only "on the fly" tool for UML statecharts supporting full μ-calculus (SPIN uses LTL, CADP Evaluator the alternation free μ-calculus).

The fact of being able to state and check also structural properties of system configurations (state attributes and predicates), and not just events, opens the door to the modelling and verification of several structural properties of parallel systems, like topologic issues, state invariants, and mobility issues.

The approach adopted in UMC seems promising but there is still a lot work to do. Certainly the UMC coverage must be extended to include at least call operations, events deferring, and state internal activity. Moreover the semantic / logic issues still need to be assessed (i.e. precisely which kind of property do we want to verify, and which kind of optimizations do they allow to be implicitly performed by the tool). The current alpha-version of the UMC prototype [29] (which is now at version 2.5) is accessible "online" through its www interface at the address http://fmt.isti.cnr.it/umc/.

8. References

[1] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.

[2] OMG *Unified Modeling Language Specification*, Version 1.4, November 2000, (www.omg.org/technology/documents/formal/uml.htm)

[3] E.M. Clarke, E.A. Emerson, and A.P. Sistla. "Automatic Verification of Finite--State Concurrent Systems Using Temporal Logic Specification", *ACM Transaction on Programming Languages and Systems*, 8, pp. 244-263 (1986).

[4] A. Bouali, S. Gnesi, S. Larosa. "The integration Project for the JACK Environment", *Bulletin of the EATCS*, n.54, pp. 207-223, 1994.

[5] M. von der Beeck. "Formalization of UML-Statecharts", in Proc. *UML 2001* Conf., *LNCS* 2185, Springer-Verlag, pp. 406-421, 2001.

[6] D. Latella, I. Majzik, and M. Massink. "Towards a formal operational semantics of UML statechart diagrams", in Proc. *FMOODS '99*, IFIP TC6/WG6.1, pp. 331-347. Kluwer Academic Publishers, 1999.

[7] R. Wieringa and J. Broersen. "A minimal transition system semantics for lightweight class and behavioral diagrams" in Proc. *ICSE'98* Workshop on Precise Semantics for Software Modeling Techniques, 1998.

[8] A. Fantechi, S. Gnesi, F. Mazzanti, R. Pugliese, and E. Tronci. "A Symbolic Model Checker for ACTL", in Proc. Conf. On Applied Formal Methods, *FM-Trends 98*, *LNCS 1641*, Springer - Verlag, 1999.

[9] R. De Nicola, and F.W. Vaandrager. "Action versus State based Logics for Transition Systems", Ecole de

Printemps on Semantics of Concurrency, *LNCS 469*, pp. 407-419, 1990.

[10] D. Kozen. “Results on the Propositional μ-calculus”, *Theoretical Computer Science*, vol. 27, pp. 333-354, 1983.

[11] M. Hennessy, and R. Milner. “Algebraic Laws for Nondeterminism and Concurrency”, *JACM* vol. 32, pp. 137-161, 1985.

[12] S. Gnesi and F. Mazzanti. “On the Fly Verification of Networks of Automata”, in Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (*PDPTA'99*).

[13] G. Bhat, R. Cleaveland, and O. Grumberg. “Efficient on-the-fly Model checking for CTL*”, in Proc. *IEEE Symposium on Logics in Computer Science*, pp. 388-397, 1995.

[14] J.C. Fernandez, C.Jard, T.Jron, and C.Viho. “Using on-the-fly verification techniques for the generation of test suites”, in Proc. Conf. on Computer-Aided Verification (*CAV ’96*), *LNCS 1102*, pp. 348-359, Springer, 1996.

[15] C. Stirling, and D. Walker. “Local Model Checking in the Modal Mu-Calculus”, *Lecture Notes in Computer Science* vol. 354, Springer Verlag, pp 369-383, 1989.

[16] F.Moller and P. Stevens. *Edinburgh Concurrency Workbench User Manual (Version 7.1)* Available by http from www.dcs.ed.ac.uk/home.cwb/.

[17] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. “Symbolic Model Checking without BDDs”, in Proc. *TACAS'99*, *LNCS* 1579, Springer-Verlag 1999.

[18] D. Latella, I. Majzik, and M. Massink. “Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker”, Formal Aspects of Computing. *The International Journal of Formal Methods*. Springer, 11(6):637--664, 1999

[19] M. M. Gallardo, P. Merino and E. Pimentel. “Debugging UML Designs with Model Checking”, in *Journal of Object Technology*, vol. 1, no.2, 2002, pp. 101-117.

[20] E. Mikk, Y. Lakhnech, M. Siegel and G. J. Holzmann. “Implementing Stateacharts in Promela/SPIN” in Proc. of *WIFT'98,* 1998.

[21] S. Gnesi, D. Latella, and M. Massink. “Model checking UML statechart diagrams using JACK” in Proc. Fourth IEEE International High-Assurance Systems Engineering Symp. (*HASE 99*), pp. 46-55, A. Williams, IEEE Computer Society Press, 1999.

[22] T. Schäfer., A. Knapp and S. Merz. “Model Checking UML State Machines and Collaborations”, *Software Model Checking*, vol. 55(3) Electronic Notes in Theoretical Computer Science, Paris 2001.

[23] J. Lilus, and I. Porres Paltor. “*v*UML: a Tool for Verifying UML Models”, 14th IEEE International Conf. on Automated Software Engineering, (*ASE’99*), pp. 255-258 1999.

[24] G.J. Holzmann. The SPIN Model Checker, *IEEE TSE* vol. 23 (1997), pp. 279-295

[25] A. Knapp, S. Merz, and C. Rauh. “Model Checking Timed UML State Machines and Collaborations”, in Proc. *FTRTFT 2002*, 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems. Springer *LNCS*, 2002.

[26] W. Ho, J.M. Jézéquel, A. Le Guennec, and F. Pennaneac'h. “UMLAUT: an extendible UML transformation framework”. In *Proc. Automated Software Engineering, ASE'99,* Florida, October 1999.
(see also www.irisa.fr/pampa/UMLAUT)

[27] J-C Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighineanu. “CAPD: a Protocol Validation and Verification Toolbox”, in Proc. Conf. on Computer-Aided Verification (*CAV ’96*), LNCS 1102, Springer-Verlag 1996 (www.inrialpes.fr/vasy/cadp/)

[28] R. Mateescu, M. Sighireanu. “Efficient On-the-fly Model-Checking for regular Alternation-Free μ-Calculus”, *Science of Computer Programming* 46(3) 2003.

[29] F.Mazzanti. UMC User Guide (Version 2.5) ISTI Technical Report 2003-TR-22. Available by http from fmt.isti.cnr.it/~mazzanti/publications/UMC-UGV25.pdf