Extending CTL with Actions and Real Time

DAVID N. JANSEN and ROEL J. WIERINGA, Universiteit Twente, Faculteit der Informatica, Vakgroep IS, Postbus 217, NL-7500 AE Enschede, The Netherlands.
E-mail: dnjansen@cs.utwente.nl, roelw@cs.utwente.nl

Abstract

In this paper, we present the logic ATCTL, which is intended to be used for model checking models that have been specified in a lightweight version of the Unified Modelling Language (UML). Elsewhere, we have defined a formal semantics for LUML to describe the models. This paper’s goal is to give a specification language for properties that fits LUML; LUML includes states, actions and real time. ATCTL extends CTL with concurrent actions and real time. It is based on earlier extensions of CTL by De Nicola and Vaandrager (ACTL) and Alur et al. (TCTL). This makes it easier to adapt existing model checkers to ATCTL. To show that we can check properties specified in ATCTL in models specified in LUML, we give a small example using the Kronos model checker.

Keywords: Real-time logic, model checking, action modalities.

1 Introduction

In the past 20 years, model checking has been developed as a technique for verifying properties of software systems [10, 5, 4]. Model checking consists of verifying whether a property is true in a Kripke structure or labelled transition system (LTS), which represents the possible behaviours of a software system. One of the advantages of model checking is that a negative answer always takes the form of a counterexample of the property to be verified, usually in the form of a trace through the transition system. This counterexample gives a lot of information to the analyst about LTS.

There are several model checkers available, each with its own property language and LTS specification language. The property languages are usually variants of linear temporal logic (LTL) [16] or computation tree logic (CTL) [10]. Moreover, understanding the trace output by a model checker requires intimate knowledge of the model checker used. Our research goal is to provide a front-end to a variety of existing model checkers that allows the analyst to specify desired properties in a language suitable to the problem at hand, and to understand the counterexample without having to know the details of the output format of various model checkers.

1.1 Unified modelling language (UML)

We work towards this goal in the context of the UML, a graphical software design language defined by the OMG [19]. The UML is a complex language whose semantics is not fully defined. We have defined a small subset, Lightweight UML [21], which is enough to describe the behavioural models we require for model checking. Elsewhere, we have defined an execution semantics for LUML and UML statecharts in terms of a Clocked Labelled Kripke
Extending CTL with Actions and Real Time

We will define this structure, which is an extension of a LTS, in detail below. An example towards the end of this paper illustrates the techniques in LUML further. We have also implemented a collection of graphical diagram editors called TCM [9], with which (L)UML diagrams can be drawn, and extended it with a facility to generate the CLKS corresponding to a collection of statecharts.

1.2 Property language

In this paper, we focus on the property language and give it a semantics in terms of the same CLKS as was done for statecharts. We present a property language called ATCTL (CTL with Actions and Real Time), that can act as a front-end for various model checkers. The idea is that the analyst builds a UML model using TCM, enters a property specified in ATCTL in TCM, selects a model checker and then verifies the property with that model checker. To achieve this, TCM must generate the LTS input for the model checker and translate the property into the property language of the model checker.

ATCTL combines two known extensions of CTL, namely ACTL and TCTL. The reason to extend CTL with both actions and real time is that in LUML state-transition diagrams, we specify states, actions and real time, and our properties refer to all of these elements. The analyst therefore needs a property language that contains constructs for all these elements.

CTL was defined by Emerson and Clarke [10] as a logic for model checking. ACTL, defined by De Nicola and Vaandrager [8], extends CTL with constructs to describe actions. ACTL was mainly designed to bridge the gap between state- and action-oriented specification techniques. TCTL, defined by Alur et al. [1], extends CTL with constructs to specify real-time properties.

ATCTL combines both extensions. We show that the result can be reduced to ACTL as well as to TCTL, and therefore also to CTL. This gives a choice of tools for model checking: SMV [3] is a CTL model checker. ACTL as TCTL can be checked by a reduction to CTL. In addition, specific TCTL and ACTL model checkers have been written: Yovine [22, 2] has implemented Kronos for TCTL. ACTL model checkers can be found in [7, 12, 17].

1.3 Comparison with other work

There are some minor differences between ATCTL and both ACTL and TCTL. First, TCTL does not contain a ‘next state’ operator, but ATCTL does. Second, we have extended ACTL with proposition symbols, so that we can give a more uniform presentation of the logics: CTL, ACTL and TCTL, as we present them, are just subsets of ATCTL. Third, Alur et al. [1] defined TCTL with a slightly different syntax; we use a syntax similar to Henzinger et al. [13], which separates clearly the action- and the real-time-oriented parts.

Latella, Majzik and Massink [15] model check UML statecharts using SPIN (a LTL model checker). Their statechart semantics, however, does not handle real time. Damm et al. [6] also model check Statemate statecharts, but these have a semantics differing from UML statecharts.

1.4 Overview of the article

After the preliminaries, we define CLKS in Section 4. In Section 5 we define ATCTL. Sections 6 and 7 then define TCTL as a subset of ATCTL and the reduction of ATCTL to TCTL.
So, it is possible to use TCTL model checkers. In Sections 8 and 9, we sketch ACTL as a subset of ATCTL and the second reduction. Since there is a model checker for ACTL [7], this gives us a second way to model check ATCTL. Section 11 contains an example of model checking and of LUML.

2 Action logic

Action logic serves to specify constraints on sets of actions. Its atomic formulae are action symbols. Assume given a finite set of action symbols \( \{a, b, \ldots\} \). Action terms \( \alpha, \beta, \ldots \) are then defined by the syntax:

\[
\alpha, \beta ::= \text{any} \mid a \mid -\alpha \mid \alpha + \beta.
\]

For the sake of simplicity, we identify action symbols with their interpretation, actions. A set of actions \( A_1, A_2, \ldots \in \mathcal{P}(A) \) is said to satisfy an action term, \( A \models \alpha \), in the following cases:

- \( A \models \text{any} \).
- \( A \models a \) iff \( a \in A \).
- \( A \models -\alpha \) iff \( A \not\models \alpha \).
- \( A \models \alpha + \beta \) iff \( A \models \alpha \) or \( A \models \beta \) (or both).

Speaking informally, any set is allowed by any. The term a requires a set containing that action; \(-\alpha\) requires a set that is forbidden by \(\alpha\); and \(\alpha + \beta\) requires a set satisfying (at least) one of the constraints. We define the abbreviation:

\[
\alpha \& \beta ::= -(\neg \alpha + -\beta).
\]

3 Time model

In our view of the world, time is modelled as the non-negative real numbers. We measure time by clocks which may be reset to zero at any moment. The clocks do not run slow or fast.

Some state changes in a system may be enabled or disabled depending on time. To describe this, we use clock constraints. Assume given a finite set of clock symbols \( \{t, t', \ldots\} \). Clock constraints \( c, c' \in \text{CC} \) are then given by the syntax: (for any \( n \in \mathbb{N} \))

\[
c, c' ::= t \leq n \mid t \geq n \mid \neg c \mid c \land c'.
\]

Other operators are defined as usual, e.g. \( m < t := -t \leq m \). A time stamp is a valuation of the clock symbols: \( \chi : C \rightarrow \mathbb{R}_0^+ \). The satisfaction relation \( \chi \models c \) is straightforward; we will not go into details. Later on, we will also use partial time stamps, which are partial functions \( v : C \rightarrow \mathbb{R}_0^+ \).

Finally, we define two abbreviations, \( \chi[C := 0] \) (where \( C \) is a set of clock symbols) and \( \chi + \delta \) (where \( \delta \in \mathbb{R}_0^+ \)), by:

\[
(\chi[C := 0])(t) := \begin{cases} 0 & \text{if } t \in C \\ \chi(t) & \text{otherwise} \end{cases}
\]

\[
(\chi + \delta)(t) := \chi(t) + \delta.
\]

4 System model

We base our system models on finite automata or finite state machines (a sort of Kripke structure) with some extensions:
Real-time aspects come in from timed automata. Basically, the time for which a system is in a certain state is quantified.

The transitions are labelled similar to labelled transition systems (LTSs).

We use propositional logic as a model for data. In traditional LTSs, the ‘state’ is reduced to possible behaviour and has no data.

4.1 Clocked labelled Kripke structures

A clocked labelled Kripke structure (CLKS) is a kind of timed automaton. It consists of a set of locations, connected by transitions. Every transition bears a label containing the set of actions executed, an enabling condition on clocks (a clock constraint) that must be true when this transition is taken, and a (possibly empty) set of clocks to be reset.

More formally, assume given a set \( \mathcal{A} \) of proposition symbols, a finite set \( \mathcal{B} \) of action symbols and a finite set of clock symbols \( \mathcal{C} \). A CLKS is a quintuple \( \mathcal{M} = (L, \rightarrow, I, i_i, i_t) \) which satisfies the conditions:

- \( L \) is a finite set of locations.
- \( \rightarrow \subseteq L \times \mathcal{A} \times \mathcal{B} \times L \) is the transition relation. For any element \( (l, A, l') \), the location \( l \) is the source, \( A \) is its label; \( l' \) is the destination.
- \( I : \mathcal{A} \rightarrow \mathcal{P}(L) \) is the interpretation of the proposition symbols.
- \( i_i : L \rightarrow CC \) assigns a clock constraint to every location, the location invariant.
- \( i_t : (\rightarrow) \rightarrow CC \times \mathcal{P}(C) \) assigns to every transition a clock constraint and a set of clocks to be reset.

A location invariant serves to enforce some transitions: the system cannot stay in the location too long, but it will take action before some deadline.

If there is a transition \( (l, A, l') \in \rightarrow \) which has clock constraint \( c \) and resets the clocks in \( C \) (i.e. \( i_t(l, A, l') = (c, C) \)), we write it as \( l \xrightarrow{A \mid i} l' \). [1]

4.2 States and steps

A state is a pair of a location and a time stamp \((l, \chi)\). A system which is in a particular state may do a time step or an action step:

A time step is characterized by a delay \( \delta \in \mathbb{R}_0^+ \). It relates a state \((l, \chi)\) always to \((l, \chi + \delta)\). For a valid time step, the location invariant of \( l \) has to hold all the time, i.e. for every \( 0 \leq \varepsilon \leq \delta \), the timestamp \( \chi + \varepsilon \) satisfies \( i_i(l) \). We denote time steps by: \((l, \chi) \xrightarrow{\delta} (l, \chi')\).

An action step belongs to a transition \( l \xrightarrow{A \mid i} l' \) of the CLKS: \((l, \chi) \xrightarrow{A \mid i} (l', \chi[C := 0])\) is a step provided that the timestamp \( \chi \) fulfils the transition’s clock constraint and the location invariant of the source \( l \), and the resulting timestamp fulfils the invariant of the destination \( l' \). In a formula, \( \chi \models c \land i_i(l) \) and \( \chi[C := 0] \models i_t(l') \).

If we talk about steps in general, we may write \((l, \chi) \xrightarrow{A \mid i \mid \delta} (l', \chi')\) (where either \( \delta = - \) or \( A \mid i \)).
4.3 Runs

A run of a system is a (possibly infinite) sequence of steps. We write this as:

\[(l_0, \chi_0) \xrightarrow{A_1} (l_1, \chi_1) \xrightarrow{A_2} (l_2, \chi_2) \cdots\]

It is possible to take several time steps without doing anything in between, or to take multiple action steps without waiting in between.\(^1\) This allows us to represent, say, a superstep in the Statemate or LUMI semantics of statecharts as a sequence of steps in a run.

To identify a certain position in a run, we use the format \((i, \delta)\), where \(i \in \mathbb{N}\) is the index and \(0 \leq \delta \leq \delta_{i+1}\) is the local delay. (If \(\delta_{i+1} = -\), only \(\delta = 0\) is allowed.) In the run above, at position \((i, \delta)\), the system is in state \((l_i, \chi_i + \delta)\). We denote the time consumed up to position \((i, \delta)\) by: \(\Delta(i, \delta) := \delta + \sum_{j=i}^{\infty} \delta_j\) (if \(\delta_j = -\), count it as if \(\delta_j = 0\)).

We say that a position \((j, \varepsilon)\) lies before position \((i, \delta)\) if either \(j < i\) or \((j = i \land \varepsilon < \delta)\).

5 ATCTL

We now present a logic which may be used to specify properties of CLKSs.

5.1 Syntax

Assume given a set of proposition symbols \(P = \{p, q, \ldots\}\), a set of action symbols \(A = \{a, b, \ldots\}\) and a set of formula clock symbols: \(C = \{t, t', \ldots\}\). (The syntax of action terms is already defined in Section 2.) The syntax of ATCTL is then defined as (for \(n \in \mathbb{N}\)):

\[
\varphi, \psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \land \psi \mid t \leq n \mid \text{reset } t \text{ in } \varphi \mid \exists x. \varphi \mid \forall x. \varphi \mid [\varphi \alpha U \psi] \mid [\varphi \alpha U^\beta \psi] \mid [\forall (\varphi \alpha U \psi)] \mid [\forall (\varphi \alpha U^\beta \psi)].
\]

The common formulae have their usual meaning. \(t \leq n\) and \(t \geq n\) hold in a state where the time stamp says that clock \(t\) is in the indicated range. reset \(t\) in \(\varphi\) is a sort of hypothetical reasoning: ‘If \(t\) were zero (ceteris paribus), then \(\varphi\) would hold’. \(\exists x. \varphi\) states that there is some action step from this state satisfying the constraints \(\alpha\), such that \(\varphi\) holds afterwards. \(\forall x. \varphi\) states that every step from this state is an action step satisfying the constraint \(\alpha\) and \(\varphi\) will hold afterwards. \([\varphi \alpha U \psi]\) is valid in a state where it is possible to do several \(\alpha\) steps (during which \(\varphi\) holds), and finally \(\psi\) becomes true. \([\varphi \alpha U^\beta \psi]\) is valid in a state where it is possible to do several \(\alpha\) steps (during which \(\varphi\) holds), and finally do one \(\beta\) step, such that \(\psi\) is valid immediately afterwards.

We include two different ‘until’ operators for several reasons: in the reduction to TCTL, \(\alpha U\) is the more natural operator. However, in the example, we will see a case where \(\alpha U^\beta\) is needed. In ACTL, both are defined.\(^2\)

5.2 Abbreviations

The paper by Alur et al. [1] defines combined operators; we may introduce them as abbreviation: \(\exists (\varphi \alpha U^n \psi) \equiv \text{reset } t_{\text{new}} \text{ in } \exists (\varphi \alpha U (t_{\text{new}} \leq n \land \psi))\), where \(t_{\text{new}}\) denotes a

\(^1\)We do allow for so-called Zeno behaviour. That is, we allow a system to take infinitely many action steps in finite time. We do this because the tool Kronos does so and because it simplifies the reduction to ACTL.

\(^2\)The construct \(\exists (\varphi \alpha U^\beta \psi)\) is definable as an abbreviation: for \(\forall (\varphi \alpha U^\beta \psi)\), we couldn’t find any definition.
new clock symbol, and we can define similar abbreviations for \(\geq\) etc. In all case studies we have done so far, the combined operator could be used; we have chosen our definition because it separates real-time and action more clearly.

The simpler modal operators \(\blacklozenge, \lozenge\) are defined as usual, e.g.: \(\exists_\blacklozenge \alpha \varphi := \exists (\top \mathcal{U}_\alpha \varphi)\).

5.3 Semantics

Assume given an ATCTL-model \(M = (L, \rightarrow, I, i, i_0)\). We will interpret a formula in a state of the model \((l, \chi)\) together with a partial time stamp \(\nu\), which serves to interpret bound clock variables; \(\nu(t)\) is defined if \(t\) is in the scope of a \(\text{reset}\) \(t\) in \(\cdot\) operator.\(^3\)

The satisfaction relation \((M, l, \chi) \models_\nu \varphi\) is defined as usual for the propositional constructs. The other definitions are:

- \((M, l, \chi) \models_\nu t \leq n\) iff \(f(t) \leq n\), where \(f(t) = \begin{cases} \nu(t) \text{ if defined} \\ \chi(t) \text{ otherwise.} \end{cases}\)
- \((M, l, \chi) \models_\nu t \geq n\) iff \(f(t) \geq n\), where \(f(t)\) is defined as above.
- \((M, l, \chi) \models_\nu \text{reset } t \text{ in } \varphi\) iff \((M, l, \chi) \models_\nu[t:=0] \varphi\).
- \((M, l, \chi) \models_\nu \exists_\X_\alpha \varphi\) iff there is a run \((l, \chi) \xrightarrow{A} (l', \chi') \cdots\) which satisfies \(A \models \alpha\) and \((M, l', \chi') \models_\nu \varphi\).
- \((M, l, \chi) \models_\nu \forall_\X_\alpha \varphi\) iff every run starting in \((l, \chi)\) has the form \((l, \chi) \xrightarrow{A} (l', \chi') \cdots\) and satisfies \(A \models \alpha\) and \((M, l', \chi') \models_\nu \varphi\).
- \((M, l, \chi) \models_\nu \exists_\X_\alpha \mathcal{U}_\psi\) iff there is a run \((l_0, \chi_0) \xrightarrow{A^1} (l_1, \chi_1) \xrightarrow{A^2} (l_2, \chi_2) \cdots\) starting in \((l, \chi) = (l_0, \chi_0)\) which satisfies the simple until condition: there is a position \((i, \delta)\) in the run \((i \geq 0)\) such that
  - \((M, l_i, \chi_i + \delta) \models_\nu \Delta(i, \delta) \psi\) and
  - \((M, l_j, \chi_j + \varepsilon) \models_\nu \Delta(j, \varepsilon) \varphi \lor \psi\), for every position \((j, \varepsilon)\) before \((i, \delta)\); and
  - either step \(j\) is a time step (i.e. \(\delta_j^- \neq \varepsilon\)), or \(A^i_j \models \alpha\), for every \(j\) with \(0 < j \leq i\).
- \((M, l, \chi) \models_\nu \forall_\X_\alpha \mathcal{U}_\beta \psi\) iff every run starting in \((l, \chi)\) satisfies the simple until condition.
- \((M, l, \chi) \models_\nu \exists_\X_\alpha \mathcal{U}_\beta \psi\) iff there is a run \((l_0, \chi_0) \xrightarrow{A^1} (l_1, \chi_1) \xrightarrow{A^2} (l_2, \chi_2) \cdots\) starting in \((l, \chi) = (l_0, \chi_0)\) which satisfies the double until condition: there is a position \((i, 0)\) in the run \((i > 0)\) such that
  - \((M, l_i, \chi_i) \models_\nu \Delta(i, 0) \psi\) and
  - \((M, l_j, \chi_j + \varepsilon) \models_\nu \Delta(j, \varepsilon) \varphi\), for every position \((j, \varepsilon)\) before \((i, 0)\); and
  - \(A^i_j \models \beta\); and
  - either step \(j\) is a time step, or \(A^i_j \models \alpha\), for every \(j\) with \(0 < j < i\).
- \((M, l, \chi) \models_\nu \forall_\X_\alpha \mathcal{U}_\beta \psi\) iff every run starting in \((l, \chi)\) satisfies the double until condition.

\(^3\)Henzinger et al. [13] call \(\nu\) a clock environment.
6 TCTL

TCTL is a temporal logic without action modalities introduced by Alur et al. [1, 13]. It extends CTL [10] by real time. There is a model checker Kronos [2, 22] for TCTL, so if we can reduce ATCTL to TCTL, we can check ATCTL formulae. (Kronos only knows the combined operators $U^{<\infty}$ etc. similar to the abbreviation we have defined for ATCTL.

As mentioned above, this is not a severe restriction, because we could make do with the combined operators.)

We can see TCTL as ATCTL over the empty action symbol set ($A = \emptyset$) and without the operators $X_{a}$ or $\alpha U_{\ell}$. They are omitted because without action labels, there is no unique 'next state' notion. So, TCTL syntax is defined as:

$$\varphi, \psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \land \psi \mid t \leq n \mid t \geq n \mid \text{reset } t \text{ in } \varphi \mid \exists (\varphi U \psi) \mid \forall (\varphi U \psi).$$

The operators have semantics similar to the corresponding ATCTL operators. We may define corresponding abbreviations, e.g. $\exists (\varphi U^{<\infty} \psi) ::= \text{reset } t_{\text{new}} \text{ in } \exists (\varphi U t_{\text{new}} \leq n \land \psi)$.

6.1 TCTL models

TCTL models are timed automata. Roughly speaking, they are CLKSs without action labels. A timed automaton (or clocked Kripke structure) is a quintuple $\mathcal{T} = (L, \rightarrow, I, \hat{i}, \hat{u})$ which has the same elements as a CLKS, except that the transition relation is binary:

- $L$ is a finite set of locations.
- $\rightarrow \subseteq L \times L$ is the transition relation. For any element $(l, l')$, the location $l$ is the source and $l'$ is the destination.
- $I : \mathcal{P} \rightarrow \mathbb{P}(L)$ is the interpretation of proposition symbols.
- $\hat{i} : L \rightarrow CC$ assigns a clock constraint to a location, its location invariant.
- $\hat{u} : (\rightarrow) \rightarrow CC \times \mathbb{P}(\mathcal{C})$ assigns to every transition a clock constraint and a set of clocks to be reset.

States are defined the same way as in ATCTL. Action steps bear no action label. However, we continue to distinguish action steps $(l, \chi) \rightarrow (l', \chi')$ and time steps. Runs are made up of steps as in ATCTL. For more details, see [13].

Here too, we interpret a formula $\varphi$ over a state $(l, \chi)$ of a timed automaton $\mathcal{T}$ and a partial time stamp $v$. The definition of the satisfaction relation $(\mathcal{T}, l, \chi) \models v \varphi$ is a simple adaptation of the corresponding definition for ATCTL. We give one example:

$$(\mathcal{T}, l, \chi) \models v \exists (\varphi U \psi) \text{ iff there is a run } (l_0, \chi_0) \rightarrow (l_1, \chi_1) \rightarrow (l_2, \chi_2) \cdots$$

starting in $(l, \chi) = (l_0, \chi_0)$ which satisfies the until condition: there is a position $(i, \delta)$ in the run $(i \geq 0)$ such that

- $(\mathcal{T}, l_i, \chi_i + \delta) \models v + \Delta(i, \delta) \psi$; and
- $(\mathcal{T}, l_j, \chi_j + \varepsilon) \models v + \Delta(j, \varepsilon) \varphi \lor \psi$, for every position $(j, \varepsilon)$ before $(i, \delta)$.

7 Reduction of ATCTL to TCTL

We now define a reduction of ATCTL formulae and models to TCTL. It extends the reduction defined by De Nicola and Vaandrager [8]: Most of the operators of ATCTL have a direct
Extending CTL with Actions and Real Time

7.1 Action locations

Because TCTL transitions are unlabelled, we introduce an action location for every transition. We add propositions corresponding to the action symbols to the proposition symbols. Further, there is one special proposition \( \text{ACT} \) which holds in the action locations to distinguish them from non-action locations. In every action location, the action propositions hold that correspond to the original transition’s label.

To ensure that actions are instantaneous, we have to add one clock symbol \( t_{\text{new}} \). Clock constraints in the location invariant forbid staying in the action location. Transitions are split into two parts: for every transition \( tr = (l, A, l') \) of \( \mathcal{M} \) with \( i_t(tr) = (c, C) \), we define two new transitions in \( \mathcal{M}^T \), namely \( l \xrightarrow{c} tr \) and \( tr \xrightarrow{[t_{\text{new}}=0]} l' \).

For the rest, the translated model is just a copy of the ATCTL model. More details can be found in the technical report [14].

7.2 Translation of states, steps and runs

Now, continue with states: as the translated model has one new clock, there is a slight difference between states of \( \mathcal{M} \) and the states of \( \mathcal{M}^T \). In the translation of a state, one may choose the value of \( t_{\text{new}} \) nearly arbitrarily.

In the translation of steps, we distinguish action and time steps: action steps are split into two parts, corresponding to the split transitions. Time steps are translated directly. A run, then, is translated stepwise. As the translation of each individual step is one or two step(s) in \( \mathcal{M}^T \), their composition is a run again.

This mapping is injective and essentially surjective. Only for the new clock symbol \( t_{\text{new}} \), not every value is hit. One could say, in a loose manner of writing: the extended mapping \( \{ \text{runs of } \mathcal{M} \} \times \mathbb{R}^+_0 \rightarrow \{ \text{runs of } \mathcal{M}^T \text{ starting in non-action locations} \} \) is a bijection.

7.3 Translation of formulae

We reinterpret action formulae as a special kind of propositional formulae, for we have reinterpreted action symbols as a special kind of proposition symbols. For example, \((\alpha + \beta)^T = \alpha^T \vee \beta^T\).

Propositional formulae are translated straightforward. For the modalities, we show the two difficult cases:

\[
(\exists X_\alpha \varphi)^T = \exists (\neg \text{ACT } U \equiv 0 \exists (\text{ACT } \land \alpha^T \ U \equiv 0 \ \varphi^T)).
\]

\[
(\forall (\varphi \ U_\beta \psi)^T = \forall ((\text{ACT } \land \alpha^T) \lor (\neg \text{ACT } \land \varphi^T) U \exists (\text{ACT } \land \beta^T \ U \equiv 0 \neg \text{ACT } \land \psi^T)).
\]

**Theorem 7.1**

Let \( \mathcal{M} = (L, \rightarrow, I, i_t, i_u) \) be an ATCTL model, \( (l, \lambda, i_u) \) one of its states, \( \varphi \) an ATCTL formula and \( v \) a partial time stamp.

Then, \( (\mathcal{M}, I, \lambda) \models_v \varphi \iff (\mathcal{M}^T, I, \lambda^T) \models_v \varphi^T \).

**Proof.** The proof is given in technical report [14].
8 ACTL

ACTL is a logic with action modalities, but (originally) with $\bot$ as the only atomic proposition and without real time, introduced by De Nicola and Vaandrager [8]. It adapts CTL [10] to action-oriented situations. We have added proposition symbols to the original ACTL, so we are able to describe it as an extension of CTL. Another addition we have made is that the transitions are labelled with a set of concurrent actions rather than a single action. We can see ACTL as ATCTL over the empty clock set: $C = \emptyset$. Then, models, transitions and steps do not bear any real-time information (or, stated differently, every real number is acceptable as the delay of a time step). Time steps in ACTL models are similar to internal steps denoted by $\tau$ in process algebra [18].

De Nicola et al. [7] have constructed a model checker for ACTL which translates ACTL formulae and models to CTL and relies on a CTL model checker to do the rest. So, we may choose to translate ATCTL to ACTL and use this model checker. Although there are ATCTL formulae that Kronos cannot handle, model checking is less efficient this way because the state space is blown up heavily in the reduction.

8.1 ACTL syntax

We repeat the constructs of ATCTL which make up ACTL:

$$\varphi, \psi ::= \bot \mid p \mid \neg \varphi \mid \varphi \land \psi \mid \exists x \alpha \varphi \mid \forall x \alpha \varphi \mid \exists (\varphi \alpha^{U^t} \psi) \mid \forall (\varphi \alpha^{U^t} \psi) \mid \exists (\varphi \alpha^{U^t \beta} \psi) \mid \forall (\varphi \alpha^{U^t \beta} \psi).$$

The operators have semantics similar to the corresponding ATCTL operators.

8.2 ACTL models

Labelled Kripke structures (LKS) are ACTL models. (Originally, ACTL models were called labelled transition systems.) They are similar to ATCTL models without clocks. Assume given a set of proposition symbols $P$ and a finite set of action symbols $A$. A labelled Kripke structure is a triple $L = (L, \rightarrow, I)$ which satisfies the conditions:

- $L$ is a finite set of locations.
- $\rightarrow \subseteq L \times (P(A) \cup \{\top\}) \times L$ is the transition relation. For any element $(l, A^\tau, l')$, the location $l$ is the source, $A^\tau$ is the action set, and $l'$ is the destination.
- $I : P \rightarrow P(L)$ is the interpretation of proposition symbols.

As there is no relevant time stamp in ACTL, states and locations are the same here. Steps $l \xrightarrow{A^\tau} l'$ are defined as in ATCTL. Finally, runs are defined the same way as in ATCTL. For more detailed information, see [7, 8].

8.3 ACTL semantics

Assume given an ACTL-model $L = (L, \rightarrow, I)$. We interpret a formula in a state $l$ of the model. The satisfaction relation $(L, I) \models \varphi$ is similar to the satisfaction relation of ATCTL.

4There is some asymmetry in our definition: in most systems, $\tau$ steps may change the state of the system thoroughly. In our setting, only transitions may become enabled or disabled (because of their clock constraints) after an internal step; no propositional formula changes its truth value.
but doesn’t include time stamps. We give one clause for illustration:

\[(\mathcal{L}, l) \models \exists (\varphi \land L \psi) \text{ iff there is a run } l_0 \xrightarrow{A_1^i} l_1 \xrightarrow{A_2^i} l_2 \rightarrow \cdots \text{ starting in } l = l_0\]

which satisfies the double until condition: there is a position \(i > 0\) in the run such that

- \((\mathcal{L}, l_i) \models \psi; \text{ and}\)
- \((\mathcal{L}, l_j) \models \varphi, \text{ for every position } j \text{ before } i; \text{ and}\)
- \(A_1^j \models \beta; \text{ and}\)
- \(A_2^j \models \tau \text{ or } A_2^j \models \alpha, \text{ for every } j \text{ with } 0 < j < i.\)

9 Reduction of ATCTL to ACTL

The reduction to ACTL, eliminating real time, is a bit more complicated than the first reduction. It is similar to the reduction given by Alur et al. [1]. We do not have space to give the reduction in detail; we give only a general overview. For more details, we refer the reader to report [14].

We construct a region automaton. This is a finite quotient of the ATCTL model (which has an infinite number of states). It is, in general, a very large automaton which may be too large for existing model checkers. The correspondence between an ATCTL model and its region automaton is a bit looser than the correspondence between an ATCTL model and its translated TCTL model: as the clock information in the translated model is lost, the mapping is no more injective.

In our translation, we did not find a satisfactory mapping of the \(\cdot\) in \(\varphi\) operator. It is difficult to translate because the \(\cdot\) in \(\varphi\) operator constructs a relation between states which are completely unrelated in the translated model.\(^5\) This implies that to check real-time properties, one has to add clock resets to the model.

9.1 Equivalent states

Assume given a CLKS. We first define an equivalence relation of time stamps \(\chi\) of the CLKS; that leads us to an equivalence relation of states.

Note that all clock constraints \(t \leq n\) and \(t \geq n\) contain only natural numbers \(n\). In an ATCTL formula, one may express which clock is the first to change its integer part. So, time stamps with the same integer parts and the same ordering of the fractional parts cannot be distinguished by ATCTL formulae and are declared equivalent. Further, assume given an \(N\) such that that in every interesting formula \(t \leq n\) or \(t \geq n\), we have \(n \leq N\). Time stamps whose clocks exceed \(N\) are declared equivalent. The equivalence relation is called \(N\)-similarity.

The equivalence relation is then extended to states by: two states \((l, \chi)\) and \((l', \chi')\) are \(N\)-similar, if \(l = l'\) and the time stamps are equivalent. The equivalence classes of this extended relation are regions, written as \([l, \chi]_N\).

9.2 Translation

The region automaton is the translation of an ATCTL model. Its locations are the regions; its transitions correspond to the transitions in the ATCTL model.

\(^5\)Introducing some kind of special transitions to create this relation is not a solution, as this affects the set of runs starting in a specific state, so the semantics of \(\forall\) formulae would change.
We define mappings of ATCTL states, steps and runs to ACTL. Formulae are mostly translated straightforwardly. For the time constraints \( t \leq n \) and \( t \geq n \), add new proposition symbols \( t^{\leq n} \) and \( t^{\geq n} \) which hold in the corresponding regions and translate time constraints to these proposition symbols.

### 9.3 Equivalence theorem

We can then formulate an equivalence theorem similar to Theorem 7.1.

**Theorem 9.1**

Let \( \mathcal{M} = (L, \rightarrow, I, i_I, i_t) \) be an ATCTL model, \((l, \chi)\) one of its states and \( \varphi \) an ATCTL formula without reset \(-\) in operators.

Then, \( (\mathcal{M}, l, \chi) \models \varphi \) iff \( (\mathcal{M}^A, [(l, \chi)]\cdot) \models \varphi^A \).

**Proof.** The proof is given in technical report [14].

### 10 Comparison of the reductions

We have defined the logics ATCTL and (our variants of) ACTL and TCTL. In our framework, we can give a uniform presentation of ATCTL and the other languages: TCTL is ‘ATCTL without actions’ and ACTL is ‘ATCTL without clocks’.

#### 10.1 Complexity

The translation of formulae is, for both reductions, linear. The reduction to ACTL \(^A\) is somewhat simpler. However, this advantage disappears when we look at the translation of models: The reduction to TCTL \(^T\) (just add action locations) is linear in the number of transitions, but the reduction to ACTL (construct the region automaton) is exponential in the number of clocks and the largest clock constraints. (For exact numbers, see [1].) This is why we decided to implement only the translation to TCTL and let an optimized tool handle the more complicated parts.

#### 10.2 Problems

TCTL lacks some of the operators we have defined for ATCTL: \( X_{\alpha} \) and \( \mathcal{U}_g \). We have found a workaround for the translation. ACTL, on the other hand, lacks the time-related operators. We have not looked for a translation of reset \( t \) in \(-\), because we have abandoned plans to implement this translation. The state explosion problem (because of the exponential translation) needs heavy optimizations; we prefer to reuse work already done by others for TCTL.

### 11 Example

#### 11.1 Case description

As an example to show that our research goal is reachable in principle, we model an internal travel office (ITO) of a large firm. The case is adapted from Verbeek et al. [20].

Imagine an internal travel office in a large firm, e.g. a university. Employees of the firm book business trips via the office. To do this, the office needs a travel permit and a payment allowance.
As an extra service, also private trips can be booked. The cost of a private trip is deducted from the employee’s salary, and no payment allowance is needed.

The office proposes a trip schedule to the employee; as soon as he accepts it, the office tries to book a trip and a hotel for the appropriate period. If this succeeds, the employee is invited to come along and to pick up the necessary documents. If the booking fails, the employee is informed and the trip is cancelled; however, the employee is allowed to restart the procedure. The financial department handles the payments and, if applicable, the deduction from the salary.

We plan to introduce a workflow system that controls this process. We would like to ensure that no payment is made without allowance, and model checking is the method we use.

### 11.2 Lightweight UML

The main elements of LUML are lightweight class diagrams and state–transition diagrams. Class diagrams describe which classes can exist in a specific object-oriented system. They are a simple subset of UML static structure diagrams. A class box may contain a class name, attribute declarations, and signal declarations. Signals formalize messages sent among objects. We allow associations and association classes.

State–transition diagrams (STD) describe the behaviour of each object of a given class. In a STD, a state of the object is depicted by a box. A transition shows how the object reacts to a signal: which response it produces and what will be its next state. For details on the semantics, see [21].

We have modelled the system using our tool TCM (Toolkit for Conceptual Modelling). TCM is available under Gnu Public License from [9]. At this time, we only handle the parallel composition of (finitely many) state–transition diagrams. So, the behaviour of finite object-oriented systems can be modelled.

The model consists of the lightweight class diagram in Figure 1. The two classes’ behaviour is shown in the statecharts of Figures 2 and 3. The system consists of two parts, corresponding to the two departments.

The prototype translator translated these diagrams first to a CLKS, according to the semantics defined by Eshuis and Wieringa [11]: state names are translated to proposition symbols; actions become labels on the transitions.

Then, a second phase of the translator generated a Kronos input file. We have checked the following properties using Kronos:

- ‘The system is non-Zeno’ or, more exactly, ‘in every state, time may pass at least one second’. This can be formalized in ATCTL as:
  \[ \text{init} \rightarrow \forall \square [\exists T, \exists U \geq 1 \ T] \].

- ‘No payment without allowance’. We should state this more precise, saying ‘without allowance, all payments are deducted from the salary.’ This is formalized in ATCTL as:
  \[ \neg \exists [\neg \text{allowed} U / \neg \text{pay} \& \neg \text{deduct} T] \).

- ‘No payment without statement of expenses’.
  \[ \neg \exists [\neg \text{got claim} U / \text{pay} T] \).

Note the different formalization of ‘payment’ in this property and the one before. Kronos reported that the property holds in all three cases.
Figure 1. Lightweight class diagram for the ITO system

Figure 2. The ITO’s process for a trip
12 Conclusion

We have defined an extension ATCTL of CTL with actions and real time, and we have shown that the resulting logic is reducible to TCTL, so that ATCTL formulae can be model checked by, for example, Kronos. An alternative reduction to ACTL allows us to model check all ATCTL formulae not including a clock reset action. However, this second reduction is much less efficient. By means of an example, we have shown that model checking works at least for small cases. All reductions are essentially surjective, so that it is possible to translate counterexamples generated by a model checker back to ATCTL.

ATCTL is a logic that fits LUML well, because it allows the analyst to speak about states, actions and real time. Thanks to the reductions, we can check properties of LUML models by reusing existing model checkers.

We have implemented a prototype translator from statecharts to CLKSs, so that we can undertake larger case studies, where we intend to look at the interplay between diagram-based and textual (formal) specification and the added value that arises by combining these two approaches. For example, we want to look at mapping paths through a CLKS to paths through a collection of object-oriented statecharts. This would help the user understanding the feedback from the model checker. Finally, we want to look at possible semantics of object-oriented statecharts that are closer to the vaguely described UML semantics, by dropping the restrictions imposed by our semantics.

Acknowledgements

This work profited from the helpful comments of Ed Brinksma, Pascal van Eck, Maarten Fokkinga, Joost-Pieter Katoen, Frits Vaandrager, and two anonymous referees.

David N. Jansen is partially supported by the DEIRDRE project, NWO 612-323-419, and the Esprit Working Group ASPIRE, contract number 22704.
References


Received 9 January 2001