Model Checking of Discrete Systems and Theorem Proving of Hybrid Systems

Solofomampionona Fortunat Rajaona (fortunat@aims.ac.za)
African Institute for Mathematical Sciences (AIMS)

Supervised by: Dr. Jeff Sanders
United Nations University, Macao, and
Prof. Ingrid Rewitzky
University of Stellenbosch

19 May 2011

Submitted in partial fulfillment of a postgraduate diploma at AIMS

Abstract

This essay begins with the description of the basis of classical model checking: We define a Kripke structure and a transition system and review the topological characterisation of safety and liveness properties. Temporal logics LTL and CTL are presented with examples. We summarize the automata-based algorithm for LTL model checking and the labelling algorithm for CTL. Then we present the tools for a promising verification technique: The differential dynamic logic d$\mathcal{L}$ and its automated theorem prover KeYmaera. KeYmaera combines deductive proving, process algebra, analysis and logics to verify hybrid systems where model checking alone is limited. As an application, we consider a reachability problem in biology.

Declaration

I, the undersigned, hereby declare that the work contained in this essay is my original work, and that any work done by others or by myself previously has been acknowledged and referenced accordingly.

Solofomampionona Fortunat Rajaona, 19 May 2011

Contents

1. Introduction

Formal verification techniques have reached more and more areas beyond information and communication technologies. Engineers in biological systems, railways, high speed trains and air traffic control all use formal verification tools for checking critical-safety properties of their models

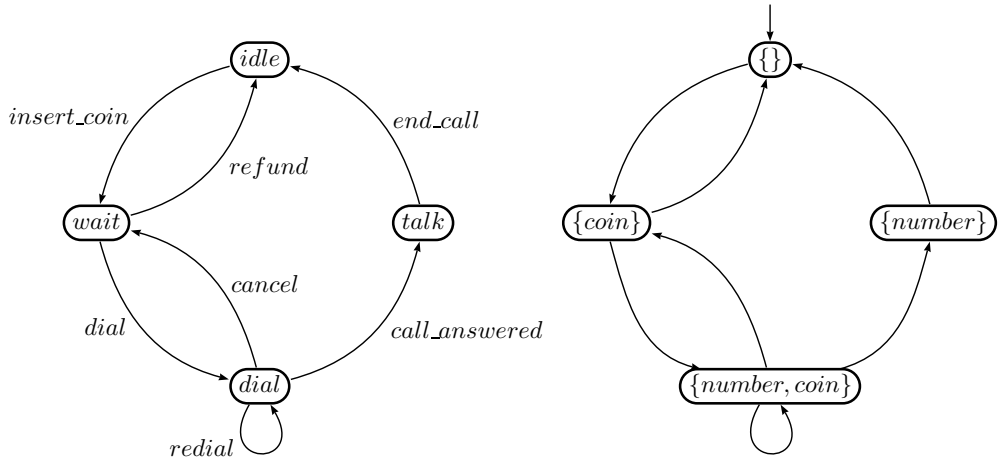
Model checking emerged in the 1980s with the works of E. M. Clarke and E. A. Emerson [CE80], and in parallel J. P. Queille and J. Sifakis [QS82]. Many model checking software have been released such as SPIN, SMV, Java Pathfinder, and they have been used to verify communication protocols, hardware and software products. In 2007, E. M. Clarke, E. A. Emerson and J. Sifakis jointly received the Turing Award for their role in developing the basis of model checking.

Chapter 2 presents the basis model checking techniques. We describe the model of the system to be analysed and give a formal characterisation of its properties. Temporal logics CTL and LTL are defined to specify these properties and associated model checking algorithms are introduced. We also consider extensions and limitations of model checking; for instance the difficulty for handling hybrid systems (systems that display both continuous and discrete behaviour).

In chapter 3, we define hybrid automata to model the complex behaviour of hybrid systems. André Platzer proposed the differential dynamic logic d$\mathcal{L}$ [Pla10] that describes the dynamics of the hybrid systems and their properties. His work has been applied in particular for verifying European Train Control System (ETCS) [PQ09] and aircraft collision avoidance [PC09]. We will present the logic d$\mathcal{L}$, the proof calculus associated, and the method used for an automatic proof. For practical purpose, in chapter 4 we use the d$\mathcal{L}$ theorem prover KeYmaera for a reachability problem in biology.

2. Model checking of reactive systems

Reactive systems are systems which can be controlled by repeated input from the user, perhaps indefinitely, with the result that the system may not terminate. Operating systems, communication protocols, ATM and vending machines are typical examples of reactive systems. Model checking may allow automatic verification of properties of reactive systems. Given a model of the system which is a transition system TS and a property expressible in temporal logic φ, model checking check whether TS satisfies φ ($TS \models \varphi$). We are going to describe the models of these systems, characterise their typical properties and then study the mathematical logics used to express these properties and then present the ideas of some model checking algorithms.



(a) Transition system (b) Kripke structure

Figure 2.1: Transition system and Kripke structure of a simplified public phone

2.1 Modelling the systems

2.1.1 Transition system and Kripke structure. First, we have to define the following notations:

- AP is a set of atomic propositions,
- S is a set of states,
- Act is a the set of actions,
- $\rightarrow \subseteq S \times Act \times S$ a set of transition relations between the states,
- I is a set of initial states,
- L is the labelling function $L : S \rightarrow 2^{AP}$.

A transition system is a tuple $(S, Act, \rightarrow)$ whereas a tuple $(S, \rightarrow, L, I)$ is known as a Kripke structure.

Transition systems are used when modelling reactive systems, for example when modelling a synchronous execution we need to label the transition with the name of the action. The actions in Act are procedures that operates on the state variables and helps describing the model into the program that generates the states.

However, for model checking algorithms only the Kripke structure is relevant, the actions are omitted, and a set of atomic propositions AP is chosen to characterize the states of the system. The labelling function L associates each state to the set of atomic propositions which are true in that state. For example, $AP = \{coin\_is\_inserted, number\_in\_memory\}$ or $AP = \{y > 0, P\_is\_busy, Q\_is\_busy\}$.

Figure 2.1(a) shows the transition system of a simplified public phone and 2.1(b) shows the corresponding Kripke structure when we choose the set of atomic propositions $AP = \{number, coin\}$, where $number$ is true when there is a number in the memory of the machine and $coin$ is true when there is a coin available inside.

In the subsequent sections, we refer to our model as transition system though we may require it to have the properties of a Kripke structure. A transition system can also be used to model terminating systems like standard programs by introducing a set of final states.

2.1.2 Concurrent systems. Most reactive systems are distributed systems: they are composed of many processes which execute in parallel. Some properties of concurrent systems which are simply stated may be very difficult to check due to the large number of possible runs of the systems; model checking originated from this challenge. Some important concepts about concurrency are described as follows.

Interleaving. The order of repeated execution of independent processes P and Q is non-deterministic. All possibilities of order are allowed: for example $PQPQQQPQ\ldots$ or $PQPQPQPQ\ldots$ or even $PPPPPPPP\ldots$. The last execution is considered an unfair execution, such execution can be omitted under *fairness assumptions*.

Consider two processes P and Q with a shared variable n that execute this same program in parallel

```
shared variable n=0
local variable loc = n
loc ++
n = loc
print(proc_name, "n = %d", n)
```

then at the end we can have different results:

P, n = 1	P, n = 2	Q, n = 2	Q, n = 1	Q, n = 1	P, n = 1
Q, n = 2	Q, n = 2	P, n = 1	P, n = 2	P, n = 1	Q, n = 1

Atomicity. Atomic actions cannot overlap with other system's actions. They are expressed in atomic statement are delimited by $\langle\rangle$. If we modify the previous example and put the assign and print statement in an atomic statement,

```
shared variable n=0
local variable loc = n
loc ++,
⟨ n = loc, print(proc_name, "n = %d", n)⟩
```

it will be impossible to get the result

P, n=2
Q, n=2

Synchronous execution. Two processes execute actions simultaneously; to execute synchronous actions the processes must be ready at the same time.

Communication via shared variable. When the processes have shared memory, it is possible to use shared variable.

Communication via channels. Channels are FIFO buffers where messages can be sent and received by the processes. A process ready to send can send only if the channel is not full. A process ready to receive can receive only if the channel is non-empty.

- $c!x$: statement for sending a message x through the channel c.
- $c?x$: statement for receiving a message from channel c and and storing it in the variable x.

When $cap(c) > 0$, we have an asynchronous communication and so there may be a delay between the sending and the receiving actions. When $cap(c) = 0$, we have a synchronous communication: the sending and the receiving actions are done simultaneously.

2.1.3 Paths, traces and reachable states. A *path* π is an infinite sequences of states $s_0, s_1, s_2, \ldots$ where $s_0 \in I$ and for all $i \geq 0$ $s_i \to s_{i+1}$. We associate to a path its trace: $trace(\pi) = L(s_0)L(s_1)L(s_2)\ldots = A_0A_1A_2\ldots$ $trace(\pi)$ is an infinite word over the alphabet AP (i.e $\in (2^{AP})^\omega$). For a transition system which has final states, we can still consider only infinite paths by adding a self-loop in those final state.

A state is said to be reachable iff it can be obtained after some sequence of actions starting from an initial state.

2.2 Linear-time properties

Linear-time properties are property of the execution paths $s_0 \to s_1 \to s_2 \to \ldots$ of a transition system. Classical examples of linear-time properties are:

- *Mutual exclusion.* In every state, at most one process is in the critical section.
- *Absence of deadlock.* A deadlock is a situation where no statement is executable although the processes are not in their final state.
- *Absence of starvation.* In every state, if a process tries to enter its critical section, then eventually it will be enabled to do so.

2.2.1 Invariant. The two first examples above can be expressed as invariants. An invariant is a property that holds for all states in the execution paths, and can be expressed with state formulas.

The model checking of invariant properties reduces to a systematic validation of the property in all reachable states, thus requiring a search algorithm in the state space, which is usually a modification of the depth first first search algorithm [BK08].

2.2.2 Safety property and liveness property. [Lam77] established two fundamental aspects of linear properties: safety and liveness. A safety property is the requirement that "something bad will not happen" like absence of deadlock and liveness is the requirement that "something good will eventually happen" like the absence of starvation. [AS85] showed that any other temporal property is a safety or a liveness or a conjunction of the two.

A safety property is a requirement over paths: the path should not contain any prefix from a given set of *bad prefixes*. Invariants are special cases of safety properties. For example: a red light in the traffic light is always directly preceded by an orange light, this safety property cannot be expressed as an invariant.

A liveness property φ is a requirement over paths: any finite path can be extended to an infinite path which satisfies φ. For example: in a pedestrian traffic light, if a request have been activated, then eventually it will be accepted.

2.2.3 Topological characterisation of safety and liveness. Different methods of classification of linear time properties have been proposed, using the syntax of the temporal formula expressing them or studying the equivalent Büchi automaton of the formula as in [AS86]. We do a characterisation by means of topological properties as in [AS85].

A trace $A_0A_1A_2\dots$ of an execution is an element of the product set $(2^{AP})^\infty = 2^{AP} \times 2^{AP} \times \dots$. We endow each 2^{AP} with the discrete topology, so it is a compact topological space since it is finite. By Tychonoff's theorem the product space $(2^{AP})^\infty$ is also compact. Basic open sets of $(2^{AP})^\infty$ are of the form

$$\{A_0\} \times \dots \times \{A_n\} \times 2^{AP} \times \dots \times 2^{AP} \times \dots \tag{2.2.1}$$

for a finite sequence $\{A_0\}, \dots, \{A_n\}$ of basic open sets of 2^{AP}. But, elements of a set in the form (2.2.1) are exactly the traces that have a common finite prefix $\tilde{\sigma} = A_0 \dots A_n$. An open set V is an union of these sets

$$V = \bigcup_{i \in I} \{\sigma \text{ such that } \sigma \text{ contains the finite prefix } \tilde{\sigma}_i \}$$
$$V = \{\sigma \text{ that have a finite prefix among } \{\tilde{\sigma}_i\}_{i \in I}\}$$

We notice that the complement of the open set V is a safety property whose bad prefixes are in $\tilde{\sigma}_i$ and conversely any safety property is a complement of an open set, thus a closed set.

In the initial definition above, a set P_l is a liveness property if any finite trace σ can be extended to be an element of P_l. This is equivalent to

for all finite prefix $\tilde{\sigma}$, there exists $\sigma_l \in P_l$ which contains $\tilde{\sigma}$
$\Leftrightarrow$ for all basic open $V_{\tilde{\sigma}}$ set associated to $\tilde{\sigma}$, there exists $\sigma_l \in P_l \cap V_{\tilde{\sigma}}$
$\Leftrightarrow$ any basic open set intersects P_l
$\Leftrightarrow$ P_l is a dense subset

Then, we can characterise safety properties as exactly the closed subsets and liveness properties as exactly dense subsets. This yields an important result from [AS85],

2.2.4 Theorem ([AS85])**.** *Every property P is the intersection of a safety property and a liveness property.*

Proof. Let $\overline{P}$ be the closure of P, which from the previous result is also the smallest safety property containing P. Let $P_l = (\overline{P} \cap P^C)^C = \overline{P}^C \cup P$, so $P_l \cap \overline{P} = (\overline{P}^C \cup P) \cap \overline{P} = P \cap \overline{P} = P$. Then, we

have to show that P_l is a liveness property, i.e a dense subset. By contradiction, if P_l is not dense, there exists a non-empty open set $O \subseteq \overline{P} \cap P^C$, because $O \subseteq P^C$ implies $P \subseteq O^C$, we have $P \subseteq \overline{P} \cap O^C$. $\overline{P} \cap O^C$ is closed as intersection of two closed sets, which is a contradiction because $\overline{P}$ is the closure of P. □

2.2.5 Fairness assumptions. The non-determinism in the actions of processes leads to all the possibilities of computation: including unfair computations (unfair paths). These cases can be omitted under fairness conditions, that is often necessary for establishing liveness properties. Fairness conditions are usually divided in three groups, which are described in [BK08] as follows

- *Unconditional fairness.* e.g., every process gets its turn infinitely often.
- *Strong fairness.* e.g., every process that is enabled infinitely often gets its turn infinitely often.
- *Weak fairness.* e.g., every process that is continuously enabled from a certain time instant on gets its turn infinitely often.

As a result of progress in the theory of nondeterministic computation, such properties can now be replaced by stronger probabilistic versions, which quantify the probability of a process getting its turn. The result can be model checked with a probabilistic model checker like PRISM [KNP02].

2.3 Temporal logic

The atomic propositions chosen for a model allow us to express that a property is true or false in a particular state. Temporal logics can express more general properties; for example, the property of a path instead of single state. The notion of time in temporal logic comes from the succession of states; an unit of time can be seen as the jump from one state to its direct successor. However, there are several interpretation of time, and several types of temporal logics resulting from them. We will develop the Linear-time Temporal Logic (LTL), the Computation Tree Logic (CTL) and CTL* which unifies the two.

2.3.1 Linear-time Temporal Logic (LTL). In linear-time logic, we consider the computations paths independent of each other. The actual run of the system will generate one of these paths. A logical formula satisfied by the transition system is therefore a logical formula satisfied by all of the possible paths. Then, an LTL formula is first defined for a path.

An LTL formula over a path is defined by the following syntax

$$\varphi, \psi := \mathbf{true} \mid a \in AP \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \rightarrow \psi \mid \bigcirc \varphi \mid \Diamond\varphi \mid \Box\varphi \mid \varphi \,\mathcal{U}\, \psi$$

The semantic of an LTL formula over a path π is the following where $\sigma = A_0A_1A_2\ldots$ is the trace of π

$$\begin{array}{l}
\pi \models \mathbf{true} \\
\pi \models a \text{ iff } a \in A_0 \\
\pi \models \neg\varphi \text{ iff } \pi \not\models \varphi \\
\pi \models \varphi \wedge \psi \text{ iff } \pi \models \varphi\pi \models \psi \\
\pi \models \varphi \vee \psi \text{ iff } \pi \models \varphi \text{ or } \pi \models \psi \\
\pi \models \bigcirc\varphi \text{ iff } \sigma[1..] = A_1A_2A_3\ldots \models \varphi \\
\pi \models \Diamond\varphi \text{ iff for some } i\ \sigma[i..] \models \varphi \\
\pi \models \Box\varphi \text{ iff for all } i\ \sigma[i..] \models \varphi \\
\pi \models \varphi\,\mathcal{U}\,\psi \text{ iff for some } i\ \sigma[i..] \models \psi \text{ and for } 0 \leq j < i\ \sigma[j..] \models \varphi
\end{array}$$

Then, the satisfaction relation of an LTL formula over a state s is defined as follows

$$s \models \varphi \text{ iff for all path } \pi \text{ starting from } s \text{ we have } \pi \models \varphi$$

and finally for the transition system TS:

$$TS \models \varphi \text{ iff } (\text{ for all paths } \pi \text{ in } TS\ \pi \models \varphi) \Leftrightarrow (\text{ for all } s_0 \text{ in } I\ s_0 \models \varphi)$$

Examples:

- the mutual exclusion requires two processes not to be in the critical section at the same time, thus every state s satisfies $\neg crit\_1 \vee \neg crit\_2$, the LTL formula expressing mutual exclusion for the system is $\Box(crit\_1 \vee \neg crit\_2)$
- The specification that a red light is always directly preceded by an orange light is expressed by $\Box(orange \vee \bigcirc\neg red)$

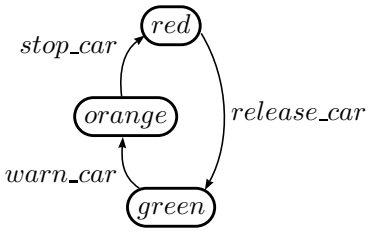


Figure 2.2: Traffic light transition system

2.3.2 Computation tree logic (CTL). In CTL, time is considered to be a branching-tree. A formula is stated about a state, and we allow quantifiers over the paths that pass through that state.

The syntax of CTL state formulas[1] is defined as

$$\Phi, \Psi ::= \mathbf{true} \mid a \mid \neg\Phi \mid \varphi \wedge \Psi \mid \Phi \vee \Psi \mid \Phi \rightarrow \Psi \mid \forall\Diamond\Phi \mid \exists\Diamond\Phi \mid \forall\bigcirc\Phi \mid$$
$$\exists\bigcirc\Phi \mid \forall\Box\Phi \mid \exists\Box\Phi \mid \Phi\,\forall\mathcal{U}\,\Psi \mid \Phi\,\exists\mathcal{U}\,\Psi$$

[1] We are using capital Greek letters to refers to a state formula in contrast of the lower-case Greek letters which refers to path formula like we have seen for LTL.

In CTL the temporal connectives $\Diamond$,$\bigcirc$, $\Box$ and $\mathcal{U}$ are always preceded by a quantifier $\forall$ or $\exists$. $\forall$ means "for all paths" and $\exists$ means "there exists some path" starting from the state.

Examples: the CTL formulas over the set $AP = \{(y > 0), (p = 1), (q = 1)\}$

$\exists \bigcirc \forall \Box (y > 0)$ is syntactically correct
$\forall \bigcirc (p = 1 \ \vee \ q = 1) \rightarrow \neg(y > 0)$ is syntactically correct
$\exists \bigcirc \neg(p = 1) \mathcal{U} (y > 0)$ is syntactically incorrect since $\mathcal{U}$ is not preceded directly by a $\exists$ or a $\forall$

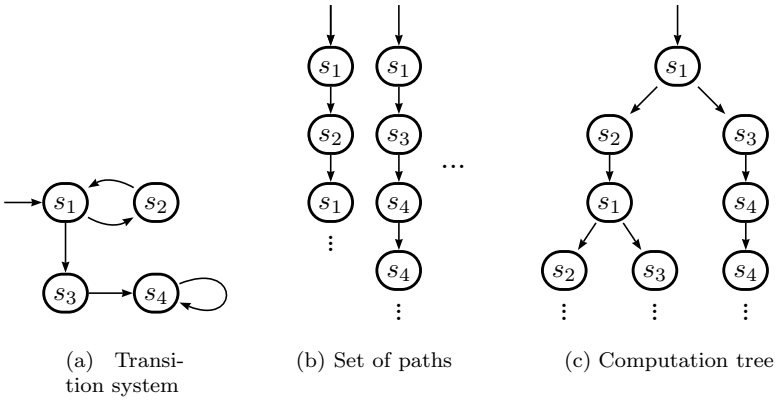
The interpretation of the boolean connectives remains classical. The satisfaction relation is defined inductively by

$s \models a$ iff $a \in L(s)$
$s \models \neg\Phi$ iff $s \not\models \Phi$
$s \models \Phi \ \wedge \ \Psi$ iff $s \models \Phi$ and $s \models \Psi$
(the interpretation of $s \models \Phi \rightarrow \Psi$ and $s \models \Phi \ \vee \ \Psi$ also follow.)
$s \models \exists \bigcirc \varphi$ ($s \models \forall \bigcirc \Phi$) if there is a path π(for all paths π) | $\pi \models \bigcirc \Phi$ i.e $\pi[1] \models \Phi$
$s \models \Phi \exists \mathcal{U} \Psi$ if there is a path π (for all paths π) starting from s, $\pi \models \Phi \mathcal{U} \Psi$
i.e there exists j such that $\pi[j] \models \Psi$ and $\pi[i] \models \Phi$ for $0 \leq i < j$
(the interpretation of formulas involving $\Diamond$ and $\Box$ also follow.)

Examples of specification in CTL:

- The specification "from any state it is always possible to go back to the initial state" is expressed as $\forall \Box (\exists \Diamond s_0)$. ($s_0$ labels the initial state).

2.3.3 Expressiveness of CTL and LTL. We have seen that a LTL formula ϕ is first defined over the paths, and a state s satisfies ϕ if all paths starting from s satisfy ϕ, in that sense LTL logics has an universal quantifier over the set of paths. But LTL logic does not have an existential quantifier over the paths and cannot express properties like "from all future states there exists a path where φ holds". However, this property can be expressed in the CTL formula $\forall \Box (\exists \Diamond \varphi)$. This does not mean that CTL logic is more expressive than LTL. There exists also LTL formulas which are not expressible in CTL, for instance the formula $\Diamond \Box \phi$. The expressive powers of LTL and CTL are therefore incomparable.



(a) Transition system (b) Set of paths (c) Computation tree

Figure 2.3: Example illustrating the points of view of CTL and LTL

The logic CTL* combines the expressiveness of CTL* and LTL, CTL* formulas are defined as CTL but in addition it allows the temporal connectives to be used without a path quantifier. It allows formula such that $\exists(\Box\Diamond\varphi)$, which is expressible neither by LTL nor by CTL [HR99]. We are not going to give the syntax of CTL* but rather give an overview of some CTL and LTL model checking algorithms.

2.4 Basic model checking algorithms

2.4.1 LTL model checking . LTL model checking using automata was first introduced by Vardi and Wolper [SVW87]. An automaton is a tuple $(S, \Sigma, \rightarrow, I, A)$ where S is a set of states Σ is a set of symbols (alphabet), $S \times \Sigma \rightarrow S$ is a set of transition function, I a set of initial states and A is a set of accepting states.

A finite (an infinite) word is a finite (an infinite) sequence of symbols from Σ. An automaton has the property of accepting words.

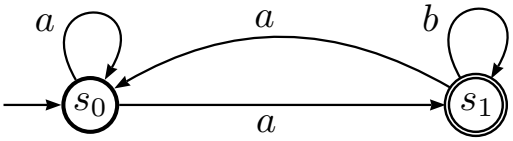


Figure 2.4: Example of automaton

A finite automata accepts finite words that ends in an accepting state. For example the accepted words of the automaton in Figure 2.4 is the language $a(a+b)^*b$.[2]

The definition of Büchi automaton is the same as for a finite automaton, expect that a Büchi automaton accepts infinite words and the accepted states are the states that are visited infinitely often. The accepted words for the automaton in Figure 2.4 considering it as a Büchi automata is the set of words that have infinitely many b.

[2] finite automata accept a regular language and Büchi automata accept ω-regular language

An LTL formula can always be represented by a Büchi automata. Figure 2.4 for example, represents the formula $\Box\Diamond b$. The first stage of automata-based model checking is to construct a Büchi automaton. This construction is done recursively on the subformulas, using for instance complementation and intersection of Büchi automata. [BK08] gives an alternative of this construction.

Algorithm 1 shows the basic idea of an automata-based LTL model checking, the product transition system $TS \otimes \mathcal{A}_{\neg\phi}$ generates paths of TS that are accepted by $\mathcal{A}_{\neg\phi}$.

Algorithm 1 Basic LTL model checking

input: transition system TS, LTL formula Φ
output: $TS \models \Phi$
construct the Büchi automaton $\mathcal{A}_{\neg\phi}$ from the LTL formula ϕ
construct the product transition system $TS \otimes \mathcal{A}_{\neg\phi}$
find the paths of $TS \otimes \mathcal{A}_{\neg\phi}$ (set $path(TS \otimes \mathcal{A}_{\neg\phi})$)
return $path(TS \otimes \mathcal{A}_{\neg\phi}) = \varnothing$

LTL model checking with fairness assumptions can reduce to a standard LTL model checking since fairness assumptions can be expressed in LTL formulas. For example: the weak fairness assumption "every process that request continuously to enter a critical system will do so infinitely often" is expressed by

$$\Diamond\Box\phi \rightarrow \Box\Diamond\psi$$

The SPIN model checker [Hol97] supports LTL formulas, and the specification of the system is written in the Promela language. The following example from [BA08] is a motivating example for model checking. The first example of concurrent processes in Section 2.1.2 is slightly modified, we repeat the actions 10 times, before printing the result, that is written in Promela as follows

```
byte n = 0, finish = 0;

active [2] proctype P() {
 byte register, counter = 0;
 do
 :: counter == 10 -> break
 :: else ->
     register = n;
     register++;
     n = register;
     counter++
 od;
 finish++
}
 active proctype WaitForFinish() {
 finish == 2;
 printf("n = %d\n", n)
}
```

Two processes `P[0]` and `P[1]` are defined simultaneously using `active [2] proctype P()`, the pro-

cess `WaitForFinish()` print the final result. A perfect interleaving of the two processes will result in $n = 10$, so one may expect that the value of n always lies in [10,20] at the end of the program. However one can check using SPIN that this expectation is in fact false, by specifying the LTL formula $\Box(\text{finish}= 2 \rightarrow 10 \leq n \leq 20)$ in SPIN as follows

```
[](finish!=2 | 10 <= n & n <= 20)
```

In fact it is possible to have $n \leq 10$, the smallest possible being 2, after checking by SPIN.

There exists also other LTL model checking not based on automata, for instance tableau-based LTL model checking [LP85].

2.4.2 CTL algorithm. The common algorithm for CTL model checking, is the labelling algorithm (Clarke and Emerson 1981). The concept is to label each state $s \in S$ with the subformulas of Φ which are true in s. This is done iteratively on the set of subformulas $sub(\Phi)$.

Algorithm 2 summarises the basic idea of CTL model checking, $Sat(\Psi)$ denotes the set of states $s \in S$ which satisfy Ψ.

Algorithm 2 Basic CTL model checking

input: transition system TS, CTL formula Φ
output: $TS \models \Phi$
for $i \leq |\Phi|$ **do**
  for $\Psi \in Sub(\Phi)$ of length i **do**
    compute $Sat(\Psi)$ from $Sat(\Psi')$ where Ψ' are subformulas in $sub(\psi)$
  end for
end for
return $I \subseteq Sat(\Phi)$

Examples of computation of the set $Sat(\Psi)$,

- if $\Psi = \Psi_1 \ \wedge \ \Psi_2$, then $Sat(\Psi) = Sat(\Psi_1) \cap Sat(\Psi_2)$
- if $\Psi = \neg\Psi_1$, then $Sat(\Psi) = S \setminus Sat(\Psi_1)$

The algorithms of computation of the set $Sat(\Psi)$ for formulas involving temporal connectives $(\exists\Diamond, \forall\Box, \dots)$ may be more difficult than the previous examples. Their correctness are proved using fixed-point characterisation of CTL [HR99].

Handling fairness conditions for CTL algorithms is not the same as for LTL. The states in a satisfaction set have to be fair states. [BK08] gives a method for dealing with fair CTL model checking.

2.4.3 State space explosion problem. The algorithms of model checking requires exploration of reachable states of the system. However, the space of reachable states can be extremely large. For example for concurrent systems, the number of possibilities of the order of execution grows exponentially with the number of processes in action. Also, for real systems the number of variables as well as the size of their domain multiply considerably the number of states. This is known as the state-space explosion problem.

Progress has been made to solve this problem, and current researches in model checking are directed towards this purpose. Symbolic model checking was proposed by [McM92], it consists of a symbolic representation of state transitions based on Ordered Binary Decision Diagrams (OBDDs), the model checker SMV uses symbolic model checking. Partial Order Reduction is another technique. When the order of independent actions does not influence the property to be checked, one can consider only one of the orders of computation, SPIN is equipped with Partial Order Reduction. Other techniques include Symmetry Reduction, Induction, Parametrised Verification.

2.5 Extension of Model Checking

Model Checking was first designed to handle finite-state systems; this may require the system to consider only finite domain systems. However techniques like data independence, allow us to consider infinite domain problems.

Some systems evolve under probabilistic constraints; also it is sometimes useful to express desirable properties of a system in terms of probability. Probabilistic model checking has been developed for that purpose; PRISM [KNP02] is an example.

Model checking is a successful method for proving properties of discrete systems and is very efficient for verifying properties of concurrent systems. Temporal logics express specification of properties in a clear and formal way. Model checking has also the advantage to be completely automatic and to be able to give counterexamples in case of invalidity of the property.

However, model checking is limited when systems display continuous change in real time and in practice, many systems display both continuous and discrete behaviour; these are called hybrid systems. For these systems, model checking has to do many approximations, for example by considering intervals of time so it cannot handle many properties.

3. Hybrid systems and theorem proving

3.1 Hybrid systems, hybrid automata

3.1.1 Hybrid systems. Hybrid systems are systems whose states can evolve according to a continuous dynamic and a discrete control. The continuous evolution of the state of the system follows *flow conditions* and it is limited by domain restrictions called *invariant conditions*, for example $x' = c$ and $x \leq 90$. The discrete control allows discontinuous evolution of the state called *jump*, for example $c := 5$. Hybrid systems are particularly relevant when there are interactions between the two types of evolution, for example the controller switches when, after a continuous evolution a state variable reaches some critical value.

A classical example of hybrid system is a thermostat that regulates the temperature in a room. The thermostat has two modes *On* and *Off*. When it is *On* the temperature x increases continuously following $x' = K(h - x)$, and when it is *Off* the temperature decreases continuously following $x' = -Kx$. A controller switches the thermostat between the two modes: when it is too hot ($x \geq 30$) it switches the thermostat off, and when it is too cold ($x \leq 20$) it switches the thermostat on. The thermostat may have a timer that counts the time when the thermostat is on, and the time when it is off. A possible requirement is that the thermostat do not run 60 minutes without stopping ($t_{on} \leq 60$ min.)

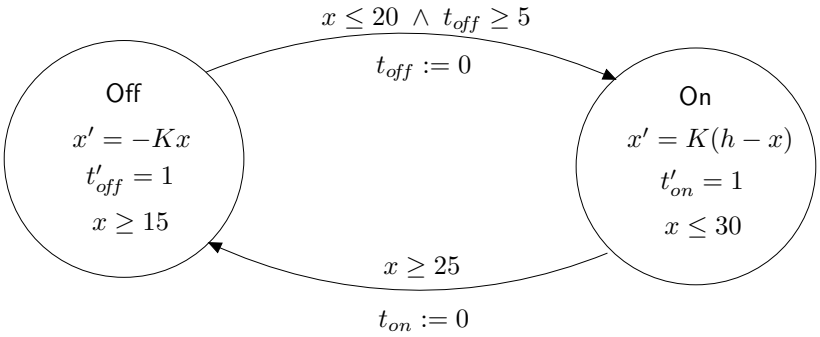


Figure 3.1: Hybrid automata of a Thermostat

3.1.2 Hybrid automata. Hybrid automata are the models to represent hybrid systems. They are defined as a tuple of

- finite set Q of vertices (modes) and a finite set E of edges (control switches)
- continuous state space $\mathbb{R}^n$ (the system has n state variables)
- $flows\ \phi_q \subseteq \mathbb{R}^n \times \mathbb{R}^n$, which are the relations relating the state $x \in \mathbb{R}^n$ to its time derivative x' when the system is in a continuous mode q.
- invariant conditions inv_q subset of $\mathbb{R}^n$ which are the domain restrictions of the state x in a mode q.
- jump relations $jump_e \subseteq R^n \times R^n$ for edges $e \in E$ comprising a transition *guard* which is a predicate on the current state x a reset relation $reset_e : \mathbb{R}^n \times \mathbb{R}^n$ which are the discontinuous change of the state x

3.2 Differential dynamic logic d$\mathcal{L}$

This section is based on the work of André Platzer (Carnegie Mellon University) in his book *Logical Analysis of Hybrid Systems* [Pla10]. "This book describes a truly unique approach to hybrid systems verification with logic and theorem proving" (Edmund M. Clarke). [Pla10] introduces the differential dynamic logic d$\mathcal{L}$ to express properties of hybrid systems. d$\mathcal{L}$ is an extension of dynamic logic[1] that allows continuous modalities expressed by differential equations. d$\mathcal{L}$ makes use of first order logic interpreted over real arithmetic $\text{FOL}_{\mathbb{R}}$. Also, d$\mathcal{L}$ is written with an hybrid program α which characterises the dynamic behaviour of the system; that allows two modal operators $[\alpha]$ and $\langle\alpha\rangle$.

3.2.1 Logical variables, signature. d$\mathcal{L}$ formulas are constructed from a set V of logical variables, and a set Σ of symbols.

- V is a set of logical variables; that means they are the variables used with the quantifiers. Variables in V are interpreted over the reals.
- the signature Σ which is a set of symbols representing functions and predicates including those from real arithmetic.
  - predicates including $=, \leq, \geq, <, >$ (from real arithmetic)
  - real valued functions including $+, -, \times, \cdot, /, 0, 1$, (from real arithmetic)

  One distinguishes between rigid and flexible symbols: flexible symbols are symbols which interpretation can change for each state. So, state variables are flexible, if the interpretation of a function can change from one state to another then it is a flexible symbol also. The arithmetic symbols 0, 1, +, - are rigid symbols, a parameter of the system is introduced as a rigid symbol. [2]
- $\text{Trm}(V, \Sigma)$ is the set of terms, which are well formed functions and predicates, with the appropriate number of arguments. A logical variable $X \in V$ is a term, $x \leq 2$, $2x - 5y$ are also terms, $x+$ and x^y are not terms.

  Also we have to define first order formulas interpreted on real arithmetic $FOL_{\mathbb{R}}$, which are used also in the definition of hybrid program below.
- First order formulas are defined inductively below for ϕ, ψ first order formulas, predicate p including $=, \leq, \geq, <, >$

$$\phi, \psi ::= p(\theta_1, \ldots, \theta_n) \,|\, \neg\phi \,|\, \phi \ \wedge \ \psi \,|\, \phi \ \vee \ \psi \,|\, \phi \rightarrow \psi \,|\, \forall x\, \phi \,|\, \exists x\, \phi$$

3.2.2 Hybrid programs. A hybrid program $\text{HP}(V, \Sigma)$ is defined inductively for state variables $x_i \in \Sigma$, terms $\theta_i \in \text{Trm}(V, \Sigma)$, a quantifier-free first-order formula χ and hybrid programs $\alpha, \beta \in \text{HP}(V, \Sigma)$:

$$x' = \theta \ \wedge \ \chi \,|\, x := \theta \,|\, ?\chi \,|\, \alpha^* \,|\, \alpha \cup \beta \,|\, \alpha; \beta$$

- $x_1 := \theta_1, \ldots, x_n := \theta_n$ is a discrete jump of the system

[1] dynamic logic itself is an extension of modal logic, the dynamic is given by a program structure that allows operators like nondeterministic repetition "*" an sequential composition ";". An example of dynamic logic formula is $\langle(x := x + 1)^*\rangle(x \geq 0)$

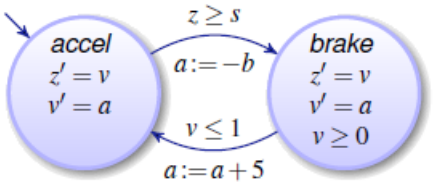
[2] state variables can also be seen as function as well as the symbols 0 and 1, they are function of arity 0. Arity is the number of argument that a function or a predicate takes. + and - are binary function whereas = is a binary predicate

- $x_1' = \theta_1, \ldots, x_n' := \theta_n \ \wedge \ \chi$ is the continuous evolution guarded by the invariant condition χ
- $\alpha \cup \beta$ non-deterministic choice between the two hybrid programs
- α^* non-deterministic (finite) repetition of a hybrid program
- $\alpha; \beta$ sequential composition of the two hybrid programs
- $?\chi$ is a test action, if χ is true in the current state then the state continues to evolve, if not, the state stops.

Other control statements, useful in practice are definable from combinations of the controls and statements above

if χ **then** α **else** β
while χ **do** α
$x := *$ (non-deterministic assignment)
...

This example from [Pla10] represents a simplified train control; on the left is the hybrid automata and on the left the corresponding hybrid program.



$q := accel$; /* initial mode is node accel */
$(\ (?q = accel;\ z' = v, v' = a)$
$\cup\ (?q = accel \wedge z \geq s;\ a := -b;\ q := brake;\ ?v \geq 0)$
$\cup\ (?q = brake;\ z' = v, v' = a \,\&\, v \geq 0)$
$\cup\ (?q = brake \wedge v \leq 1;\ a := a + 5;\ q := accel))^*$

3.2.3 Formulas of differential dynamic logic. Formulas of differential dynamic logic combine $FOL_{\mathbb{R}}$ with modal operators on the hybrid programs.

The syntax is given inductively for differential dynamic logic formulas ϕ and ψ, terms θ_i, a predicate symbol p of n arguments, terms $\theta_i \in \mathsf{Trm}(V, \Sigma)$, a logical variable $x \in V$ and a hybrid program α

$$\phi, \psi ::= p(\theta_1, \ldots, \theta_n) \,|\, \neg\phi \,|\, \phi \ \wedge \ \psi \,|\, \phi \ \vee \ \psi \,|\, \phi \to \psi \,|\, \forall x\, \phi \,|\, \exists x\, \phi \,|\, [\alpha]\phi \,|\, \langle\alpha\rangle\phi$$

The meaning of $[\alpha]\phi$ is that the formula ϕ always holds after any run of the hybrid program α and the meaning of $\langle\alpha\rangle\phi$ is that ϕ holds after some run of α. The symbols $\neg, \ \wedge\ , \ \vee\ , \to, \forall, \exists$ are the classical first order connectives.

3.2.4 Valuation of terms and formulas. To give a meaning to any logical formula, we have to interpret all the symbols. The symbols in d$\mathcal{L}$ have different nature and the interpretation is done at different levels. An interpretation I for rigid symbols in Σ. The arithmetic symbols like $+, \times, 0, =$, keep their standard interpretation. The interpretation of the state variables defines a map *state*: $\nu : \Sigma_{fl} \to \mathbb{R}$. Thus the value of the system variables characterise a state. An assignment η evaluates logical variables $\eta : V \to \mathbb{R}$

The valuation of terms and formulas is then a map $val_{I,\eta}(\nu, .)$ such that for terms we have:

- $val_{I,\eta}(\nu, x) = \eta(x)$ if $x \in V$ is a logical variable
- $val_{I,\eta}(\nu, a) = \nu(a)$ if a is a state variable
- $val_{I,\eta}(v, f(\theta_1, \ldots, \theta_n)) = I(f)(val_{I,\eta}(\nu, \theta_1), \ldots, val_{I,\eta}(\nu, \theta_n))$

d$\mathcal{L}$ formulas are evaluated intuitively, for example

$$val_{I,\eta}(\nu, \phi \wedge \psi) = true \text{ iff } val_{I,\eta}(\nu, \phi) = true val_{I,\eta}(\nu, \psi) = true$$
$$val_{I,\eta}(v, \forall x \phi) = true \text{ iff } val_{I,\eta[x \mapsto d](\nu,\phi)} = true \text{ for all } d \in \mathbb{R}$$
$$val_{\nu,\langle\alpha\rangle} = true \text{ iff } val_{I,\eta}(\omega, \phi) \text{ for some sate } \omega \text{ reachable from } \nu$$

where the valuation $val_{I,\eta[x \mapsto d]}$ agrees with $val_{I,\eta}$ except that x is evaluated to d.

3.2.5 Transition semantics of a hybrid program α. The transition relation $(\nu, \omega) \in \rho_{I,\eta}(\alpha)$ means that the hybrid program can reach the state ω starting from a state ν. The semantics is defined below

- $(\nu, \omega) \in \rho_{I,\eta}(x_1 := \theta_1, \ldots, x_n := \theta_n)$ iff $val_{I,\eta}(\omega, x_i) = val_{I,\eta}(\nu, \theta_i)$ and $val_{I,\eta}(\omega, z) = val_{I,\eta}(\nu, z)$ for variables $z \notin \{x_1, \ldots, x_n\}$.
- $(\nu, \omega) \in \rho_{I,\eta}(x_1' := \theta_1, \ldots, x_n' := \theta_n \,\&\, \chi)$ iff there is a function $f : [0, r] \to \text{Sta}(\Sigma)$ such that
  - $f(0) = \nu$ and $f(r) = \omega$
  - f respects the differential equations.
  - if $z \notin \{x_1, \ldots, x_n\}$ then z remains constant
  - f respects the invariant: $val_{I,\eta}(f(\zeta), \chi) = true$ for each $\zeta \in [0, r]$
- $\rho_{I,\eta}(?\chi) = \{(\nu, \nu) : val_{I,\eta}(\nu, \chi) = true\}$
- $\rho_{I,\eta}(\alpha \cup \beta) = \rho_{I,\eta}(\alpha) \cup \rho_{I,\eta}(\beta)$
- $\rho_{I,\eta}(\alpha; \beta) = \{(\nu, \omega) : (\nu, \mu) \in \rho_{I,\eta}(\alpha) \text{ and } (\mu, \omega) \in \rho_{I,\eta}(\beta) \text{ for a state } \mu\}$
- $(\nu, \omega) \in \rho_{I,\eta}(\alpha^*)$ iff there are states $\nu_0 = \nu, \ldots, \nu_n = \omega$ such that $(\nu_i, \nu_{i+1}) \in \rho_{I,\eta}(\alpha)$ for all $0 \leq i < n$.

3.3 Free variable proof calculus

3.3.1 Substitutions. [Pla10] A substitution σ changes simultaneously variables $y_1, \ldots, y_n$ of V into terms $\theta_1, \ldots, \theta_n$. The application of σ takes effects within a d$\mathcal{L}$ formula.

The substitution σ is admissible if no y_i is quantified ($\exists y_i$ or $\forall y_i$), no y_i is assigned by a discrete jump ($y_i := a$), or occurs in a differential equation ($y' = t$) inside the d$\mathcal{L}$ formula and no θ_i contains a term y_i.

The substitution within the formula ϕ of variable y_i by θ_i (for $1 \leq i \leq m$) is denoted $\phi_{y_1 \ldots y_m}^{\theta_1 \ldots \theta_m}$, for instance the substitution of a single variable y into θ is ϕ_y^θ.

3.3.2 Lemma (Substitution Lemma)**.** [Pla10] Let s be an admissible substitution for the (term or) formula f and let s replace only logical variables. Then for each I, η and ν:

$$\text{for each } I, \eta, \nu : val_{I,\eta}(\nu)(\nu, \sigma(\phi)) = val_{I,\sigma^*(\eta)}(\nu, phi), \tag{3.3.1}$$

where $\sigma^*(\eta)$ is adjoint[3] to σ.

3.3.3 Sequent. A sequent is of the form $\Gamma \vdash \Delta$ where Γ and Δ are set of formulas, called respectively antecedent and succedent.

If Γ and Δ are set of formulas of $d\mathcal{L}$, $\Gamma \vdash \Delta$ is satisfiable for there exists some interpretations I, some states ν and some valuations η of the logical variables such that

$$\bigwedge_{\phi \in \Gamma} val_{I,\eta}(\nu, \phi) \longrightarrow \bigvee_{\psi \in \Delta} val_{I,\eta}(\nu, \psi)$$

$\Gamma \vdash \Delta$ is valid for any interpretation I, any state ν and any valuation η of the logical variables

$$\bigwedge_{\phi \in \Gamma} val_{I,\eta}(\nu, \phi) \longrightarrow \bigvee_{\psi \in \Delta} val_{I,\eta}(\nu, \psi)$$

In other terms, the sequent is valid if one formula in the succedent is evaluated to be true or if one formula in the precedent is evaluated to be false. In sequent calculus, comma separates the formulas in the antecedent and in the succedent so that a comma in the antecedent is synonym of a conjunction and a comma in the succedent is synonym of disjunction.

3.3.4 Proof calculus. The proof calculus for d$\mathcal{L}$ is done as for classical sequent calculus using inference rules. Inference rule is given by premises above the rule bar and a conclusion below the rule bar and labelled by the name of the rule. The rule means that if the premises are valid then the conclusion is valid[4].

Proof rules for propositional logic These rules for propositional logic are essentially for ordering the formulas, eliminating the negations and implications.

$$(\neg r)\ \frac{\phi \vdash}{\vdash \neg\phi} \qquad (\vee r)\ \frac{\vdash \phi, \psi}{\vdash \phi \ \vee \ \psi} \qquad (\wedge r)\ \frac{\vdash \phi \quad \vdash \psi}{\vdash \phi \ \wedge \ \psi} \qquad (\to r)\ \frac{\phi \vdash \psi}{\vdash \phi \to \psi} \qquad (ax)\frac{}{\phi \vdash \phi}$$

$$(\neg l)\ \frac{\vdash \phi}{\neg\phi \vdash} \qquad (\vee l)\frac{\phi \vdash \quad \psi \vdash}{\phi \ \vee \ \psi \vdash} \qquad (\wedge l)\frac{\phi, \psi \vdash}{\phi \ \wedge \ \psi \vdash} \qquad (\to l)\ \frac{\vdash \phi \quad \psi \vdash}{\phi \to \psi \vdash} \qquad (cut)\frac{\vdash \phi \ \ \phi \vdash}{\vdash}$$

When doing a proof calculus, we start from the goal in the bottom and apply the rules backward. The following example prove the propositional formula $\neg(A \to B) \to A \ \wedge \ \neg B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{ax\ \dfrac{*}{A \vdash B, A}}{\to \mathsf{r}\ \dfrac{\neg \mathsf{r}\ \dfrac{ax\ \dfrac{*}{B, A \vdash B}}{A \vdash B, \neg B}}{A \vdash B, A \ \wedge \ \neg B}}{\neg \mathsf{l}\ \dfrac{}{\vdash (A \to B) A \ \wedge \ \neg B}}}{\to \mathsf{r}\ \dfrac{}{\neg(A \to B) \vdash A \ \wedge \ \neg B}}}{\vdash \neg(A \to B) \to A \ \wedge \ \neg B}$$

[3] $\sigma^*(\eta)$ is identical to η except that $\sigma^*(\eta)(x) = val_{I,\eta}(\nu, \sigma(x))$ for all logical variables x

[4] This is different from implication. An implication means that if the formula in the left is true the formula in the right is also true

3.3.5 Quantifier elimination. To remove quantifiers in formulas in a (uninterpreted) first-order logic formula, the proof rules (∀r, ∃r, ∀l, ∃l) can be used. For interpreted first-order logic this is not always true. A first-order logic admits quantifier elimination QE if for each formula ϕ that contains quantifiers $(\forall, \exists)$, there exists an equivalent formula $QE(\phi)$ which is quantifier-free. Tarski [Tar51] showed that first-order logic interpreted over the reals admits a quantifier elimination. For example $QE(\exists x(ax+b=0)) \equiv (a \neq 0 \wedge b \neq 0) \vee (a = 0 \wedge b = 0)$. For a d$\mathcal{L}$ formula like $\exists X[\alpha]\phi$, if the variable X occurs in the modality $[\alpha]$ then the quantifier elimination QE cannot be applied. It is necessary to transform these d$\mathcal{L}$ formula into a first-order logic formula before applying the quantifier elimination. To apply the rules for formulas like $\exists t[\alpha](x=0)$, the method is

- apply the first-order logic quantifier rule, which introduces a Skolem term $s(t)$
- continue the calculus involving modalities, and arrive at an algebraic constraint with the term $s(t)$
- reintroduce the quantified term in the place of $s(t)$ (rule i∀ and i∃).
- apply quantifier elimination on the first-order formula resulting

The correctness of this approach results from the substitution lemma 3.3.2.

First-Order Rules

$$(\forall r)\frac{\vdash \phi(s(X_1,\ldots,X_n))}{\vdash \forall x\phi(x)}{}^{6} \qquad (\exists r)\frac{\vdash \phi(X)}{\vdash \exists x\phi(x)}{}^{8}$$

$$(\exists l)\frac{\phi(s(X_1,\ldots,X_n)) \vdash}{\exists x\phi(x) \vdash}{}^{6} \qquad (\forall l)\frac{\phi(X) \vdash}{\forall x\phi(x) \vdash}{}^{8}$$

$$(i\forall)\frac{QE(\forall X(\Phi(X) \vdash \Psi(X)))}{\Phi(s(X_1,\ldots,X_n)) \vdash \Psi(s(X_1,\ldots,X_n))}{}^{7} \qquad (i\exists)\frac{\vdash QE(\exists X \bigwedge_i(\Phi \vdash \Psi))}{\Phi_1 \vdash \Psi_1 \ldots \Phi_n \vdash \Psi_n}{}^{9}$$

3.3.6 Calculus Rules. [Pla10] The rules schemata of d$\mathcal{L}$ are listed in the propositional logic rules, first-order logic rules, dynamic rules and general rules.

Then the calculus rules applies as follows, if the following is in a rule schemata among the list of rules

$$\frac{\Phi_1 \vdash \Psi_1 \ldots \Psi_n \vdash \Psi_n}{\Phi_0 \vdash \Psi_0}$$

thus we can apply the following calculus

$$\frac{\Gamma, \langle \mathscr{J} \rangle \Phi_1 \vdash \langle \mathscr{J} \rangle \Psi_1, \Delta \ldots \Gamma, \langle \mathscr{J} \rangle \Phi_n \vdash \langle \mathscr{J} \rangle \Psi_n, \Delta}{\Gamma, \langle \mathscr{J} \rangle \Phi_0 \vdash \langle \mathscr{J} \rangle \Psi_0, \Delta}{}^{10}$$

[6] s is a new (Skolem) function symbol and $X_1, \ldots, X_n$ are all free logical variables of $\forall x\phi(x)$

[7] X is a new logical variable. Further QE needs to be defined for the formula in the premise.

[8] X is a new logical variable

[9] Among all open branches, free logical variable X only occurs in the branches $\Phi_i \vdash \Psi_i$. Further, QE needs to be defined for the formula in the premise, especially, no Skolem dependencies on X can occur.

[10] where Γ and Δ are sets of formulas and $\langle \mathscr{J} \rangle$ is a discrete jump set.

If the following is a rule schemata of dynamic logic

$$\frac{\phi_1}{\phi_0}$$

thus we can apply the following as proof calculus.

$$\frac{\Gamma \vdash \langle \mathscr{J} \rangle \phi_0, \Delta}{\Gamma \vdash \langle \mathscr{J} \rangle \phi_1, \Delta}{}_{10} \qquad \frac{\Gamma, \langle \mathscr{J} \rangle \phi_1 \vdash \Delta}{\Gamma, \langle \mathscr{J} \rangle \phi_0 \vdash \Delta}{}_{10}$$

Dynamic rules These rules are written without sequent, they are also called symmetric rules because of the fact that they can be applied on the left or on the right of the sequent as explained above. These rules essentially splits the modalities into elementary parts.

$$(\langle ; \rangle)\frac{\langle \alpha \rangle\ \langle \beta \rangle \phi}{\langle \alpha ; \beta \rangle} \qquad (\langle {}^{*n} \rangle)\frac{\phi \vee \langle \alpha \rangle \langle \alpha^* \rangle \phi}{\langle \alpha^* \rangle \phi} \qquad (\langle := \rangle)\frac{\phi^{\theta_1 \ldots \theta_m}_{x_1 \ldots x_m}}{\langle x_1 := \theta_1, \ldots, x_n := \theta_n \rangle \phi}$$

$$([;])\frac{[\alpha]\ [\beta]\phi}{[\alpha ; \beta]} \qquad ([{}^{*n}])\frac{\phi \wedge [\alpha][\alpha^*]\phi}{[\alpha^*]\phi} \qquad ([:=])\frac{\langle x_1 := \theta_1, \ldots, x_n := \theta_n \rangle \phi}{[x_1 := \theta_1, \ldots, x_n := \theta_n]\phi}$$

$$(\langle \cup \rangle)\frac{\langle \alpha \rangle \phi \vee \langle \beta \rangle \phi}{\langle \alpha \cup \beta \rangle \phi} \qquad (\langle ? \rangle)\frac{\chi\ \wedge\ \psi}{\langle ? \chi \rangle \psi} \qquad (\langle ' \rangle)\frac{\exists t \geq 0((\forall 0 \leq \tilde{t} \leq t \langle \mathscr{S}_{\tilde{t}} \rangle \chi)\ \wedge\ \langle \mathscr{S}_t \rangle \phi)}{\langle x_1' = \theta_1, \ldots, x_n' = \theta_n \& \chi \rangle \phi}{}_{11}$$

$$([\cup])\frac{[\alpha]\phi\ \wedge\ [\beta]\phi}{[\alpha \cup \beta]\phi} \qquad ([?])\frac{\chi \to \psi}{[?\chi]\psi} \qquad (['])\frac{\forall t \geq 0((\forall 0 \leq \tilde{t} \leq t [\mathscr{S}_{\tilde{t}}] \chi) \to [\mathscr{S}_t] \phi)}{[x_1' = \theta_1, \ldots, x_n' = \theta_n \& \chi] \phi}{}_{11}$$

Global Rules

$$([]gen)\frac{\vdash \forall^{\alpha}(\phi \to \psi)}{[\alpha]\phi \vdash [\alpha]\psi} \qquad (\langle\rangle gen)\frac{\vdash \forall^{\alpha}(\phi \to \psi)}{\langle \alpha \rangle \phi \vdash \langle \alpha \rangle \psi}$$

$$(ind)\frac{\vdash \forall^{\alpha}(\phi \to [\alpha]\phi)}{\phi \vdash [\alpha^*]\phi} \qquad (con)\frac{\vdash \forall^{\alpha} \forall \nu > 0(\varphi(\nu) \to \langle \alpha \rangle \varphi(\nu - 1))}{\exists \nu \varphi(\nu) \vdash \langle \alpha^* \rangle \exists \nu \leq 0 \varphi(\nu)}{}_{12}$$

3.3.7 On soundness, completeness and decidability. First order logic is complete and real arithmetic is decidable, thus d$\mathcal{L}$ proof calculus is complete for closed first-order formula. However the occurrence of modalities makes the undecidable

3.3.8 Theorem (Soundness). *d$\mathcal{L}$ calculus is sound. All provable formulas in d$\mathcal{L}$ are valid.([Pla10])*

3.3.9 Theorem (Incompleteness). *d$\mathcal{L}$ calculus is not complete. Not all valid formulas in d$\mathcal{L}$ are provable.([Pla10])*

[10] where Γ and Δ are sets of formulas and $\langle \mathscr{J} \rangle$ is a discrete jump set.

[11] t and $\tilde{t}$ are logical variables and $\langle \mathscr{S}_t \rangle$ is the jump set $\langle x_1 := y_1(t), \ldots, x_n := y_n(t) \rangle$ with simultaneous solutions $y_1, \ldots, y_n$ of the respective differential equations with constant symbols x_i as symbolic initial values

[12] Logical variable ν does not occur in α

3.4 Automated verification techniques

The automated d$\mathcal{L}$ prover includes a main prover that does the deductive proof, it handles propositional, global rules and dynamic rules except for $\langle' \rangle$ and $[']$. The two latter are processed by algebraic computations. Quantifier elimination tool is needed to perform algebraic computations. Mathematica is preferred for its strength on doing symbolic calculations.

[Pla10] proposed also differential-algebraic logic (DAL) and differential temporal logic (dTL). The former allow disturbances in the continuous evolution and the use of quantifiers in the jumps.

4. Hybrid system verification example: bacterial chemotaxis

4.1 Problem statement

The motion of the bacteria Escherichia Coli (E. coli) is composed by two modes: run and tumble. The bacteria can sense the presence of an attractant (food) or a repellent(predator) by receptors on its body and acts accordingly. If it senses something good, it runs; if it senses nothing good or something bad, it tumbles. E. coli moves does not depend directly on the concentration of attractant (or repellent) but rather on its gradient. It has a short memory[1], giving more importance to more recent event (loss of memory).

Run If the concentration increases, the cell is in a "good direction", so it adopts the mode run: it keeps its direction and move forward during an average time of 1 second [NAM09].

Tumble If the concentration decreases or does not change, it adopts the mode tumble, a reorientation: it chooses a direction randomly and move for an average duration of 0.1 second before doing another comparison [NAM09].

We assume that our bacteria is localised in a two dimensional space. We consider a frame of reference (O, x, y). An attractant is placed at the origin; we suppose that there is a concentration c_{max} at that point and that the concentration $c(r)$ decreases as the radius r is increasing. For example, they are related by $c(r) = c_{max} - kr$ for a coefficient k.[2]

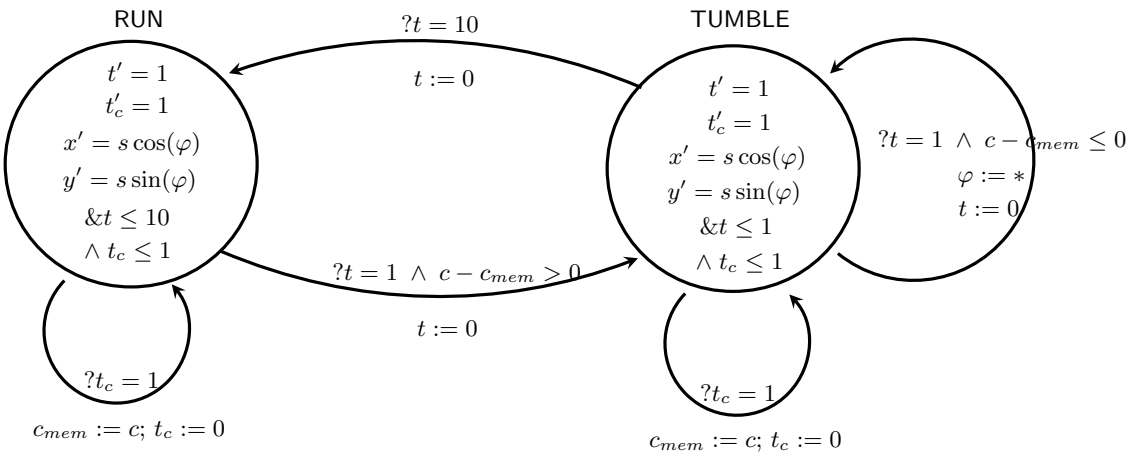


Figure 4.1: Automata used as a model of Escherichia Coli chemotaxis (unit of time: 0.1 second)

The hybrid automata in Figure 4.1 captures the motion of the bacteria. It has two modes run and tumble.

[1] many models assume the bacteria to have a memory up to 4s, like in [NAM09].

[2] a more realistic model would have used a Gaussian distribution $c(r) = c_{max} e^{-\frac{x^2+y^2}{2\sigma^2}}$

- x, y which are the coordinates of the bacteria
- t the time, which measure the duration when the bacteria is in mode tumble or in mode run
- φ is the angle that makes the current direction of the bacteria with the horizontal
- $r = \sqrt{x^2 + y^2}$
- $c = c_{max} - k * r$ is the concentration of attractant measured at a distance r from the source.
- The state variable t_c is a clock that allows to measure the concentration c each 0.1 second and compute a new concentration in memory (c_m) from it by

  $$c_{mem} := 0.75c(x, y) - 0.25c_{mem}$$

  This expresses the fact that the cell looses memory and gives more importance to the recent past. We take 25% of the old value of c_{mem} and 75% of the value of the current concentration.

Most mathematical models of E. coli chemotaxis involve probabilistic choice in the transition between the mode run an tumble. Our model is less flexible, if the concentration has increased then obligatory the bacteria will run. And if the concentration has decreased, the bacteria will tumble.

4.2 Presentation of the tool: KeYmaera

KeYmaera is the software dedicated for verifying correctness of hybrid system properties expressed in d$\mathcal{L}$. It is an interactive software, it can do a fully automatic proof and also allow the user to decide how the proof shall be done KeYmaera is based on the KeY theorem prover and can incorporate other software, like Mathematica or Redlog. For our purpose we have used KeYmaera assisted with the quantifier elimination software QEPCAD. With QEPCAD only, we cannot deal with complicated system in KeYmaera, however we still can run codes and verify properties of small systems. The important thing for us is to be able to use d$\mathcal{L}$ for describing a non-trivial system.

KeYmaera follows the syntax of d$\mathcal{L}$, in particular we have the following notations[3]

`a;b`	sequential composition	`a ++ b`	nondeterministic choice
`x:=t`	discrete jump	`{x' = t, y' = s, F}`	continuous flow
`a*`	nondeterministic repetition	`?F`	test statement
`R x`	variable declaration	`if(F) then a else b`	if condition
`x:=*`	nondeterministic assignment		

4.3 Implementation

We have made many simplifications of the model since our theorem prover lacks of arithmetic solver. Thus in our code the unit of time is 0.1 second. We assume that the bacteria has a constant velocity $s = 1$

[3]Of course there are other statement notations that we don't have here, for instance those using DAL

u, v represents the $\cos(\varphi)$ and the $\sin(\varphi)$, because $\cos$ and $\sin$ are not terms of d$\mathcal{L}$. The following statement allows a random choice of u lying between -1 and 1 and a random v such that $u^2 + v^2 = 1$

$$u := *; ?(-1 \leq u \,\&\, u \leq 1); v := *; ?(u^2 + v^2 = 1);$$

In this program, we take $k = 1$ and try to verify the trivial requirement [*bacteria*]$(t \leq 10)$ where *bacteria* is the hybrid program describing our system. Then, KeYmaera could prove that the property is true.

```
\problem {
  /* state variable declarations */
 \[ R mod; R t, tc, x, y; R u, v, cmem, s, Ux, Uy, r, c; R k , cmax\](
  /* initial state characterization */
  ( 1 >= t & t >= 0 & mod = 1 & u^2 + v^2 = 1 & tc = 0 & s = 1 & cmax =
       100 & -100 <= x&x <= 100 & -100 <= y&y <= 100  & k = 1)
  ->
  \[
  (
    (? (mod = 1 & tc < 1); /* mod 1: tumble*/
       (?(t >= 1);
          r := *; ?(r^2 = x^2 + y^2 & r >= 0);
          if (cmax - k*r >= 0) then /* take the current concentration
                                       and compare it with cmem    */
            c := cmax - k*r
          else
            c := 0
          fi;
          if (c-cmem <= 0) then        /* decrease of the concentration -->
                                    choose a random direction and tumble */
            u:=*; ?(1 >= u & u >= -1); v:=*; ?(v^2 + u^2 = 1) ; t := 0; mod := 1
          else                         /* increase of the concentration -->
                                          keep direction and run */
            t := 0; mod := 2
          fi)
     ++(?(t < 1); Ux := u*s; Uy := v*s; {x'= Ux, y'= Uy,
                                          t'= 1, tc'= 1, t <= 1 & tc <= 1 })
    )
  ++(? (mod = 2 & tc < 1); /* mod 2: run */
       (?(t >= 10);  mod := 1)
     ++(?(t < 10); Ux := u*s; Uy := v*s; {t'= 1, tc' = 1, x'= Ux, y'= Uy,
                                          t <= 10 & tc <= 1})
    )
  ++(? (tc = 1);  /* take the concentration memory every second */
      r := *; ?(r^2 = x^2 + y^2 & r >= 0);
      if (cmax - k*r >= 0) then
         cmem := .75*(cmax - k*r) + .25*cmem  /* loss of memory */
      else
         cmem := 0
      fi;
```

```
      tc := 0
    )
  )*
  \](tc <= 10)
  )
}
```

5. Conclusion

The challenge of verifying critical-safety properties of large systems has led to the development to powerful formal methods of verification. Often, the properties to be checked can be easily stated; like the absence of deadlock or the mutual exclusion. But the complexity of the systems makes the verification task extremely difficult. We have explained why distributed systems are difficult to analyse. As a solution we have proposed model checking techniques. CTL and LTL logics were defined and compared with examples illustrating their differences. We have also presented two important model checking algorithms using respectively CTL and LTL. The state-space explosion problem is a major problem of model checking, and systems displaying continuous behaviour cannot be model-checked without approximations.

Hybrid systems display continuous evolution interacting with discrete controls. They are encountered in many physical problems and their verification requires more tools than model checking. The differential dynamic logic d$\mathcal{L}$ was presented to describe the behaviour of hybrid systems. d$\mathcal{L}$ provides a sound proof calculus, even though it cannot check all properties, it has demonstrated his ability to check complex problems like air traffic collision avoidance. We have presented the basics of this technique. Although, as we didn't have a powerful algebraic solver, we could only present a very small example which does not represent the actual power of d$\mathcal{L}$ theorem prover.

Acknowledgements

My first thanks come to Dr. Jeff Sanders, who helped me a lot for achieving this essay. I also thank the founder of AIMS Pr. Neil Turok and the Director Barry Green for giving me the opportunity to write this paper. Thanks to Jan for his technical supports, Frances and AIMS tutors for their advises. I appreciate all my colleagues at AIMS for their friendship especially Muzamil, Tovo, Kazage and Sophie.

Dia misaotra ny ao an-trano rehetra koa naharitra sy nanatitra. Tsetsatsetsa tsy aritra.

References

[AS85] Bowen Alpern and Fred B. Schneider, *Defining liveness*, Information Processing Letters **21** (1985), no. 4, 181–185.

[AS86] ———, *Recognizing safety and liveness*, Distributed Computing **2** (1986), 117–126.

[BA08] Mordechai Ben-Ari, *Principles of the spin model checker*, Springer, 2008.

[BK08] Christel Baier and Joost-Pieter Katoen, *Principles of model checking*, MIT Press, 2008.

[CE80] E. M. Clarke and E. A. Emerson, *Characterizing correctness properties of parallel programs using fixpoints*, Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 85, Springer - Berlin Heidelberg, 1980, pp. 169–181.

[Hol97] Gerard J. Holzmann, *The model checker spin*, IEEE Transactions On Software Engineering **23** (1997), no. 5, 279–295.

[HR99] Michael Huth and Mark Ryan, *Logic in computer science: Modelling and reasoning about systems*, 1999.

[KNP02] Marta Kwiatkowska, Gethin Norman, and David Parker, *Prism: Probabilistic symbolic model checker*, Springer, 2002, pp. 200–204.

[Lam77] L. Lamport, *Proving the correctness of multiprocess programs*, IEEE Transactions On Software Engineering **3** (1977), 125–143.

[LP85] Orna Lichtenstein and Amir Pnueli, *Checking that finite state concurrent programs satisfy their linear specification*, Symposium on Principles of Programming Languages, 1985, pp. 97–107.

[McM92] Kenneth Lauchlin McMillan, *Symbolic model checking: an approach to the state explosion problem*, Ph.D. thesis, Pittsburgh, PA, USA, 1992, UMI Order No. GAX92-24209.

[NAM09] Dan V. Nicolau, Jr., Judith P. Armitage, and Philip K. Maini, *Research article: Directional persistence and the optimality of run-and-tumble chemotaxis*, Comput. Biol. Chem. **33** (2009), 269–274.

[PC09] André Platzer and Edmund M. Clarke, *Formal verification of curved flight collision avoidance maneuvers: A case study*, FM (Ana Cavalcanti and Dennis Dams, eds.), LNCS, vol. 5850, Springer, 2009, pp. 547–562.

[Pla10] André Platzer, *Logical analysis of hybrid systems: Proving theorems for complex dynamics*, Springer, Heidelberg, 2010.

[PQ09] André Platzer and Jan-David Quesel, *European Train Control System: A case study in formal verification*, ICFEM (Karin Breitman and Ana Cavalcanti, eds.), LNCS, vol. 5885, Springer, 2009, pp. 246–265.

[QS82] J. P. Queille and J. Sifakis, *Specification and verification of concurrent systems in cesar*, International Symposium on Programming, Lecture Notes in Computer Science, vol. 137, Springer Berlin - Heidelberg, 1982, pp. 337–351.

[SVW87] A. Prasad Sistla, Moshe Y. Vardi, and Pierre Wolper, *The complementation problem for bchi automata with applications to temporal logic*, Theoretical Computer Science **49** (1987), no. 2-3, 217 – 237.

[Tar51] Alfred Tarski, *A decision method for elementary algebra and geometry*, University of California Press, 1951.