Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS

Petr Mandys

# Combining temporal logic and behavior protocols

Department of Software Engineering

Supervisor: RNDr. Jiří Adámek
Field of study: Computer Science

I would like to thank my supervisor Jiří Adámek for his valuable comments and corrections. Next, I would like to thank my wife Lucie for her linguistic corrections and important moral support.

I declare that I completed my master thesis myself and with use of cited literature only. I agree with using this thesis.

Prague, April 19, 2006 Petr Mandys

# Table of contents

**Title:** Combining temporal logic and behavior protocols
**Author:** Petr Mandys
**Department:** Department of Software Engineering
**Supervisor:** RNDr. Jiří Adámek
**Supervisor's e-mail address:** adamek@nenya.ms.mff.cuni.cz
**Abstract:** In this thesis we consider one of the weaknesses of temporal logic – the fact that the temporal formulas specifying complex properties are hard to read. We introduce new temporal logic "BP-CTL", that originate from Computational Tree Logic (CTL) extended with operators partly taken from Behavior Protocols (BP) and partly newly defined. Text of the thesis is divided into several parts. First we introduce reader to the context of the issue. Next we describe new operators and show their usage on small examples. Then we formally define the resulting language (BP-CTL). In the next part we demonstrate the usability of BP-CTL and introduce the tool – called bpctl – for checking properties written in BP-CTL. Finally we evaluate and conclude our work. The text is extended with appendixes including detailed description of used formalisms, mapping tables of patterns collected in Property Specification Patterns project for BP-CTL and bpctl user manual.
**Keywords:** model checking, temporal logic, behavior protocols, BP-CTL, formula readability

**Název práce:** Propojení temporální logiky a behaviorálních protokolů
**Autor:** Petr Mandys
**Katedra:** Katedra softwarového inženýrství
**Vedoucí diplomové práce:** RNDr. Jiří Adámek
**E-mail vedoucího:** adamek@nenya.ms.mff.cuni.cz
**Abstrakt:** V této diplomové práci se zabýváme jednou ze slabin temporální logiky – obtížnou čitelností temporálních formulí popisujících komplexní vlastnosti. Představíme novou temporální logiku „BP-CTL", která vychází z Computational Tree Logic (CTL) a rozšiřuje ji o operátory zčásti převzaté z behaviorálních protokolů (BP) a zčásti nově definované. Text práce je rozdělen do několika částí. Nejprve seznámíme čtenáře s širším kontextem problému, kterým se zabýváme. Dále popíšeme nové operátory a ukážeme jejich použití na malých příkladech. Poté formálně zadefinujeme výsledný jazyk (BP-CTL). V další části dokážeme použitelnost BP-CTL v praxi a představíme nástroj, nazvaný bpctl, pro ověřování vlastností zapsaných pomocí BP-CTL. Na závěr zhodnotíme a shrneme naši práci. Text je dále doplněn o dodatky obsahující detailní popis použitých formalizmů, mapovací tabulky paternů shromážděných v rámci projektu Property Specification Patterns pro BP-CTL a uživatelský manuál bpctl.
**Klíčová slova:** model checking, temporální logika, behaviorální protokoly, BP-CTL, čitelnost formule

Chapter 1

# Introduction

## 1.1 Problem context

### 1.1.1 Model checking

Model checking [1] is a technique for verifying finite state concurrent systems such as sequential circuit designs and communication protocols. It has a number of advantages over traditional approaches that are based on simulation, testing, and deductive reasoning. In particular, model checking is automatic and usually relatively fast. Also, if the design contains an error, model checking produce a counterexample that can be used to pinpoint the source of the error. The method, which was awarded the 1998 ACM Paris Kanellakis Award for Theory and Practice, has been used successfully in practice to verify real industrial designs, and companies are beginning to market commercial model checkers.

The verifying of the finite state systems is achieved by verifying if the model, often derived from hardware or software design, satisfies a logical specification. The model can be written separately as a prototype of some algorithm or hardware component (for example PCI bus). Such a way is used in projects like NuSMV [8] or Spin [10]. The next possibility is to generate the model from the source code of real programs. This is currently a modern way to check the correctness of a program. There are applications, such as Bandera [11] or Java PathFinder [12], that construct the model of the program from its sources (Java sources or bytecode in these cases) and translate it to an input language of an existing (or their own) checker. They are called model-builders.

The model is usually expressed as a state transition system, for example directed graph consisting of nodes (or vertices) and edges. With each node a set of atomic propositions is associated. The nodes represent states of a system, the edges represent possible executions which modify the state, while the atomic propositions represent the basic properties that hold at a point of execution.

The main challenge in model checking is dealing with the state space explosion problem. This problem occurs in systems with many components that can interact with each other or systems with data structures that can assume many different values. In such cases the number of global states can be enormous. Researchers have made considerable progress on this problem over the last fifteen years.

### 1.1.2 Properties description

There exist many different formalisms for description of properties that we would like to check. One category of such languages consists of temporal logic systems. Concrete types of temporal logics differ in many aspects, such as used operators (logical, temporal, ...), expressive power and accompanying efficiency.

A compromise between these two aspects is for example Computational Tree Logic, which we use in this thesis. One of the problems of properties described by temporal logic formulas – which holds also for CTL – is its readability. The formulas describing complex properties are always very hard to understand.

### 1.1.3 Behavior Protocols and software components

Component programming has drawn a lot of attention, mainly because components provide a higher level of abstraction than objects. There exist languages for describing the behavior of software components, its communication and substitutability conditions. One of these languages are behavior protocols (BP) [3] – the formalism with notation similar to regular expressions.

Expressions written in behavior protocols are quite straight-lined and therefore easy to comprehend. The similarity to regular expressions also improves the readability because they are commonly used in the world of computers and software.

### 1.1.4 Specification patterns

There exist repositories of commonly occurring property specification patterns such as Specification Patterns project [5] at Kansas State University. These repositories simplify the formulation of complex properties. Users can just adjust prepared and well documented pattern expressing the wanted property.

## 1.2 Problem statement

Let us have a look on a small example demonstrating the hard readability of CTL formulas. There is a washing machine firmware and we would like to express a property: "Between water filling (`W`) and washing powder adding (`P`) the washing machine drum must not start to rotate (transition from the state where `!R` holds to state where `R` holds) more than two times". This property can be also expressed as "there must not happen: after water filling the drum starts to rotate three times until the washing powder is added". The CTL transcription of this property is:

```
AG(W -> !E[!P U (!R & !P & EX(R &
          E[!P U (!R & !P & EX(R &
          E[!P U (!R & !P & EX(R & !P))])))])))]).
```

As we can see, this formula is really hard to understand. On the contrary in the new temporal logic introduced in this thesis the same property can be expressed as:

```
!EF(W, !R; R, !R; R, !R; R &= !P).
```

The improved readability of property specifying expressions is the main contribution of this thesis.

## 1.3 Goals

This thesis aims at targeting the following goals:

- To propose a new formalism (BP-CTL) for temporal property specification, based on a combination of temporal logic and behavior protocols.
  - To write the formal definition of its syntax and semantics.
  - To demonstrate the usability of newly defined language in comparison to the concrete temporal logic.
- Prototype implementation of a model checker using BP-CTL for specification of temporal properties.

## 1.4 Structure of the thesis

The Chapter 1 introduces the reader to the context of the issue and sets the expected goals of the thesis. In the Chapter 2 we describe how to combine temporal logic (CTL in the concrete) and behavior protocols. We define new operators and show their meaning on the examples. The Chapter 3 formally describes the resulting language (BP-CTL). In the Chapter 4 we introduce the tool for specifying model properties via BP-CTL. Input for the tool is extended input language of symbolic model checker NuSMV. Next we demonstrate the usability of the BP-CTL on a complex example and we introduce the Specification Patterns project [5]. In the Chapter 5 we evaluate our work and compare achieved results with other similar projects. Moreover we describe an inconsistency found in the Specification Patterns project and propose the solution of the mistake. Finally in the Chapter 6 we conclude the results and outline possible improvements and future research.

The text of the thesis is extended with three appendixes. Appendix A includes formal description of used formalisms. In Appendix B there are all Property Specification Patterns mapping tables for BP-CTL and a comparison with the CTL mappings. Finally Appendix C includes user manual for the tool – NuSMV preprocessor – called bpctl, which allows to check properties written in BP-CTL with existing model checker NuSMV.

Chapter 2

# Combining temporal logic and BP

## 2.1 Introduction

In this chapter, we describe details of the merge of CTL and BP. At the beginning we introduce important keywords. We consecutively focus on the problem how to combine temporal logic (with emphasis on CTL) and BP in order to properties specification. Next we describe the evolution of BP-CTL. On some examples we show the purpose of the changes and then describe newly defined operators and demonstrate their usage.

With combining of the temporal logic and behavior protocols we can achieve different objects. This may be for example improvement of the expressive power, improvement of the efficiency of checking algorithms or – which is also our goal – better readability of formulas specifying complex properties.

## 2.2 Used formalisms

### 2.2.1 Temporal logic

The term temporal logic is used to describe a formal system for representing and reasoning on propositions qualified in the terms of time. Term is sometimes also used to refer to tense logic, a particular modal logic-based system introduced by Arthur Prior in the 1960s. Subsequently it has been developed further by computer scientists, notably Amir Pnueli, and logicians.

Temporal logic can be applied to clarifying philosophical issues regarding time, as a framework within which the semantics of temporal expressions in natural language is defined, as a language for encoding temporal knowledge in artificial intelligence, and as a tool for handling the temporal aspects of the execution of computer programs.

There exist two main approaches to temporal logic, predicate-logic and modal-logic approach.

The modal style of temporal logic has found extensive application in the area of computer science concerned with the specification and verification of programs, especially concurrent programs in which the computation is performed by two or more processors working in parallel. In order to ensure correct behavior of such a program it is necessary to specify the way in which the actions of various processors are interrelated. The relative timing of the actions must be carefully coordinated so as to ensure that integrity of the information shared amongst the processors is maintained. Amongst the key notions here is the distinction between "liveness" properties of the tense-logical form, which ensure that desirable states obtain in the course of the computation, and "safety" properties of the form, which ensure that undesirable states never obtains.

Gradually several formal systems of temporal logic such as interval temporal logic (ITL), linear temporal logic (LTL), computational tree logic (CTL), or more expressive system CTL* was developed.

Since the differences amongst these particular systems are very important, it is necessary for the further work to choose one concrete member of temporal logic and combine it with the behavior protocols. We decided to use CTL. It is ranked amongst the most used systems (both for theoretical reasoning and practical implementation). It has sufficient expressive power and in addition there are numbers of model checkers for CTL.

Formulas in CTL are composed of atomic propositions and logical and temporal operators. Atomic propositions are the building blocks for making statements about the states of system. The *logical operators* are the usual ones, known from most of logics: *negation* (sign as `!`), *and* (`&`), *or* (`|`), *implication* (`->`) and *equivalence* (`<->`). Model checker which we are using for checking CTL formulas (NuSMV) also accepts the advanced logical operators: *exclusive or* (`XOR`) and *exclusive nor* (`XNOR`).

*Temporal operators* are the couple – a path quantifier followed by one of the forward-time operators. Path quantifier is either `A` for all computation paths or `E` which means some computation path (or there exist a path). Forward-time operators are: globally (`G`), in the future (`F`), next time (`X`) and until (`U`).

More detailed description of the meaning of temporal operators and formal semantics of CTL can be found in Appendix A.

### 2.2.2 Behavior protocols

Behavior protocols (BP) [3] are a formalism originally intended for description of component behavior inside a component-based software application. They describe the communication possibilities for whole system and also for its particular components. They define the rules for components substitutability.

Behavior protocols address these aims: to be an easy-to-read notation for behavior specification; clear support for behavior specification refinement in ADL (Architecture Description Language); support for formal reasoning about the adherence of a component's implementation to the behavior specification of the component, as well as about the correctness of behavior specification refinement; dealing with the potential decidability and state explosion problem.

Behavior protocol consists of combination of operators and action events. BP operators are: *sequencing* (;), *alternative* (+), *repetition* (*), *and-parallel* (|), *or-parallel* (||), *restriction* (/), *composition* ($\sqcap_X$), *adjustment* ($|T|$) and *consent* ($\nabla_X$). A BP event has the following syntax:

<event prefix><connection name>.<local event name><event suffix>.

The event prefix is one of the symbols *!* (emit), *?* (receive) and τ (internal). The event suffix can be ↑ (request) or ↓ (response).

Let us imagine a component based system like in Figure 2. It has two components called *A* and *B* which communicate together via connections *a* and *b*.

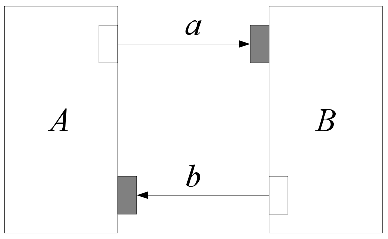


***Figure 2**: Component based system schema*

Now we can describe the behavior protocols for both components:

$$Prot_A = !a\uparrow;\ ?a\downarrow;\ ?b\uparrow;\ !b\downarrow\ +\ ?b\uparrow;\ !b\downarrow;\ !a\uparrow;\ ?a\downarrow$$
$$Prot_B = ?a\uparrow;\ !a\downarrow;\ !b\uparrow;\ ?b\downarrow\ +\ !b\uparrow;\ ?b\downarrow;\ ?a\uparrow;\ !a\downarrow$$

We refer the reader to Appendix A, [3] and [4] for the details about BP.

## 2.3 Possible solutions of the problem

Temporal logic and behavior protocols were designed for different purposes. While temporal logic (and also the members such as CTL or CTL*) is a formalism for transforming temporal expressions in natural language into the mathematic accurate expressions, BP are focused on description of the behavior of software components in the system.

There are at least two main ways how to reason about combining CTL and BP.

1) Independent coexistence
2) Extension of the logical expressions with protocol based expressions

### 2.3.1 Independent coexistence
This means that for specification we can use both CTL and BP but we cannot use any combination of them. This solution does not allow usage of temporal operators and path quantifiers inside protocol based requests, and also CTL formulas cannot use any BP expressions.

Such an extension does not fully use possibilities of combining CTL and BP. There is no change in the CTL and its hard-readability and also BP itself is not suitable for checking general properties.

### 2.3.2 Extension of the logical expressions with protocols
Via combining BP and CTL we can get the powerful tool for specifying property formulas. CTL offers path quantifying and also usage of logical and temporal operators, while BP allows well-arranged notation of state sequences and repetitions.

CTL formula has the following schema:

*path quantifier   temporal operator   logical formula*

The way that we have chosen to extend CTL formulas with BP expressions is to enrich logical formulas. In the next section we describe the details of including BP into CTL's logical formulas.

The extension is designed as shortcuts for common CTL expressions. This is possible because we do not need to improve expressive power of CTL and at the same time it is really desirable for practical use. We do not need to write a new model checker, but we can use existing efficient solution. For the use of BP-CTL with existing model checker is sufficient to implement preprocessor which transforms expressions written in BP-CTL into CTL formulas accepted by the model checker.

## 2.4 Including BP into CTL

In this section we at first describe the resulting language in general and then more precisely. We consecutively show how to understand BP expressions in the properties specifications, the modifications and extensions of BP and CTL that are necessary or desirable to do, so that the resulting language would be more useful and its expressions much more readable.

### 2.4.1 Resulting language – BP-CTL
We call the new language BP-CTL. The base is CTL which is extended with operators partly taken from BP and partly newly defined. These extensions do not change the expressive power of CTL, because they are only shortcuts for often used CTL formulas.

There is a set of similar operators in BP-CTL – we call them *state separators*, because they describe the relationships amongst contiguous states – and further two *path-oriented operators*.

**State separators**
The state separating operators are: *sequencing* (`;`), *repetition* (`*`) (both known from BP), *state separating logical and* (`&&`), *advanced sequencing* (`;+`), *special sequencing* (`,`), *advanced repetition* (`+`) and *infinite repetition* (`@`) (all newly defined).

The state separators allow to express properties describing sequences of states. Let us show their benefit on an example. Let us imagine some multi-threaded system and we would like to check such a property, that every thread after its initialization (`i`), request acceptation (`r`) and giving response (`s`) should be correctly terminated (`t`). In other words, that in the system does not remain any inactive threads. This property written in BP-CTL looks like this:

```
!(!i*; i, r;+ s & !t@)
```

and equivalent CTL formula is:

```
!E[!i U (i & EF(r & EX(EF(s & EG(!t)))))].
```

**Path-oriented operators**
The first of path-operators is operator called *explicit always* (`A(…)`). It allows to change the implicit existence in state separators interpretation (which means that they usually express that there **exists** a descendant state or path which satisfy some property).

The last of newly defined operators is called *path-oriented logical and* (`&=`). It is designed for expressing: “In every state of the given path some other property also hold.”

This is all about BP-CTL in general for now. In the rest of this chapter we focus on the detailed description of particular operators.

## 2.4.2 BP expressions interpretation

Now we describe how to understand requests for checking the state space given by BP. Every event name becomes a boolean variable which holds or not. We ignore all event prefixes and suffixes used in BP expressions. Single operators are construed as described further.

**Sequencing (`;`)**
Sequencing is the basic operator of the extension. In the property specification it separates properties which hold in two contiguous states. For better insight we show the meaning on a small example.

The expression containing sequence `a; b; c` is satisfied when there are three contiguous states where in the first property `a`, in the second `b` and in the third property `c` holds. Formula with the same meaning written in the pure CTL looks like this: `a & EX(b & EX(c))`.

Note that we implicitly use sequencing and other state-separating operators as existence operators. This means that the expression `a; b` says that there holds a property `a` in the current state and also **exists** at least one next state where `b` holds. Further an operator which replaces this implicit existence by explicit always is also defined.

**Repetition (`*`)**
This operator means that given property should hold in any finite sequence of states. We again use an illustration for better understanding.

The BP-CTL expression `a; b*; c` holds if there is a path (sequence of states), where property `a` holds in the first state of the path, then there is **finite** number of states where property `b` holds and there follows a state on the path where property `c` holds. Note that the number of states, where property `b` holds can also be zero. This means, that expression `a; b*; c` holds trivially when expression `a; c` holds. The same property can be expressed in CTL as formula `a & EX(E[b U c])`.

This operator can be also used as the last member in the expression. Although it has no impact on the expression validity (this subexpression holds at any case) we can use it by reason of formal preciseness of property specification. For example expression `a; b*` has the same meaning as CTL formula `a & EX(E[b U TRUE])`.

**Other BP operators**
It is necessary to modify BP to be usable for formulating property specifications. Thereat we omit all other BP operators (except for preceding sequencing and repetition).

So-called composed operators (composition, adjustment and consent) are designed for realizing parallelism and synchronization of software components. For our purpose they are useless as the same as restriction operator, which we also omit.

Alternative operator can be replaced by CTL logical operator or.

Finally and-parallel and from this follows that also or-parallel operator cannot be efficiently expressed in pure CTL. In general case the only way how to do this are alternatives of all possible state sequences. The length of the output CTL formula is in general case factorial multiple of length of the input expression.

### 2.4.3 Newly defined state separating operators
For formulation of efficient request we have to add some new operators. These operators were proposed (similar to path-oriented operators defined in Section 2.4.4) following the requirements during formulating common property specifications.

**Advanced sequencing (`;+`)**
The first of the new operators extends potential of original sequencing operator (`;`). While operator `;` describes two proximate consequential states operator `;+` express two states which belong to conjunctive path. Let us show the difference on an example.

The expression `a; b` means that there are two contiguous states, where in the first one property `a` holds and in the second one property `b` holds. While expression `a;+ b` holds for every pair of states, where in the first one (let us denote it $s_1$) holds property `a`, in the second one ($s_2$) holds property `b` and there exists a finite and non-empty path from $s_1$ to $s_2$.

We illustrate the meaning of this operator on an example. The following expression including extended sequencing operator `a; b;+ c; d` holds in a state-space where exists a sequence of states such that in the first state property `a` holds, in the second `b` holds and moreover in the penultimate state property c holds and finally in the last state d holds. It has the same meaning as the formula written in CTL `a & EX(b & EX(EF(c & EX(d))))`.

Here we can see that even such a simple formula – which can be part of much more complex expression – is quite hard to read when written in pure CTL.

**Special sequencing (`,`)**
This operator is very similar to the previously defined advanced sequencing operator. The difference is that special sequencing operator can be in one special case satisfied by one state. Let us show the meaning on an example.

As already mentioned expression `a;+ b` holds for every pair of states, where in the first one (let us denote it $s_1$) property `a` holds, in the second one ($s_2$) property `b` holds and there exists a finite and non-empty path from $s_1$ to $s_2$. While `a, b` holds also when states $s_1$ and $s_2$ are the same state, which means that the path from $s_1$ to $s_2$ can be empty.

We can see the difference between these two operators also on their equivalent CTL formulas. While `a;+ b` should be rewritten as `a & EX(EF(b))`, expression `a, b` has the same meaning as CTL formula `a & EF(b)`.

**Infinite repetition (`@`)**
This is an unary operator, which comes from common repetition operator (`*`). Let us show the difference between two expressions, where in the first one is used common repetition operator (`a; b*; c`) and in the second one is replaced with new infinite repetition operator (`a; b@; c`). The expression `a; b@; c` holds if holds expression `a; b*; c` and moreover also when there exists an **infinite** path, where property `a` holds in the first state of the path and property `b` holds in every following state on the path.

Now we show how to rewrite infinite repetition into the CTL. The following expression `a; b@; c` has the same meaning as CTL formula `a & EX(EG(b) | E[b U c])`.

Note that the example in the previous paragraph can be smartly expressed when we use weak until operator which is often defined as a member of CTL. The resulting formula looks like this: `a & EX[b W c]`.

There is one more difference between repetition and infinite repetition operators. The difference appears when we use these operators as the last operator in the expression. As it was mentioned while describing repetition operator, expression `a; b*` means the same as CTL formula `a & EX(E[b U TRUE])`. Because of this someone could expect that expression `a; b@` is mapped to CTL as `a & EX(EG(b) | E[b U TRUE])`.

The subformula `EX(EG(b) | E[b U TRUE])` has no impact on whole formula validity, similar to subformula `EX(E[b U TRUE])` in the case of common repetition operator. But in praxis there are commonly occurred formulas ending with subformula like `a & EX(EG(b))`.

On the basis of the previous results we establish the convention that when infinite operator occurs at the end of expression we say that it becomes strongly infinite operator. This means that expression `a; b@` is equivalent to the CTL formula `a & EX(EG(b))`.

Note, that this has the same meaning as when we use expression `a; b@; FALSE` but the newly established convention makes the expressions much more readable than tricky usage of `FALSE` expression.

**Advanced repetition (`+`)**
This is also an unary operator, that modifies meaning of repetition operator used in BP. The distinctness from repetition operator is, that advanced repetition operator requires at least one occurrence of repeated expression. Let us show the meaning of advanced repetition operator on an example. The expression `a; b+; c` holds, for every path where property `a` holds in the first state of the path, after it follows at least one (but only **finite** number) state where property `b` holds and finally there comes the state where property `c` holds.

Described expression can be also written in the pure CTL. Resulting formula looks like this: `a & EX(b & EX(E[b U c]))`.

**State separating logical and (`&&`)**
This operator is state separating equivalent to the logical and operator (`&`) in CTL. In general case it can be used in the same way as logical and operator and it also has the same meaning. The practical usage of this operator is in combination with path-oriented logical and (`&=`) or explicit always. These two operators are defined in the next section.

Example demonstrating the difference can be expression `a && b+; c &= d` which is equivalent to the CTL formula `((a & d) & (b & d)) & EX(E[(b & d) U (c & d)])`. When we write an expression `a & b+; c &= d` it has the same meaning as CTL formula `a & b & EX(E[b U c]) & d`. The difference is in the logical interpretation of the left operand. While expression `a && b+; c` describes **a path** where in the first state hold properties `a` and `b` and property b holds also in all following states until property `c` becomes true, the expression `a & b+; c` describes **a state** where holds property `a` and also here starts a path where holds property `b` until property `c` becomes true. For better understanding we show the difference in Figures 3 and 4.

We call these operators (`&&`, `;`, `*`, `;+`, `,`, `@` and `+`) state separators. As we can see all of them express relationship amongst two or more states. None of them – except one special case of special sequencing operator and of course state separating logical and – can be used in the framework of one state.

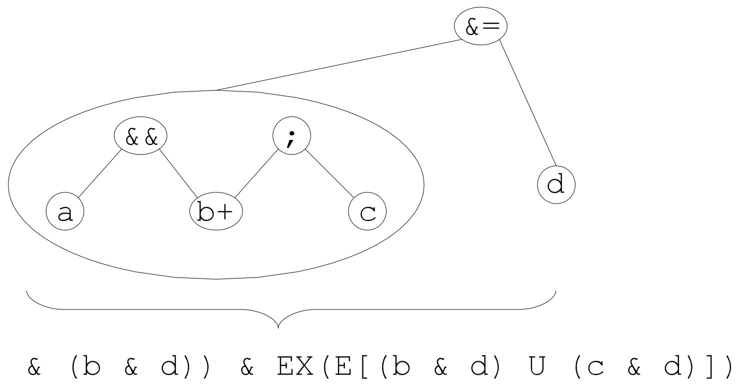


***Figure 3****: State separating logical and combined with path-oriented logical and*

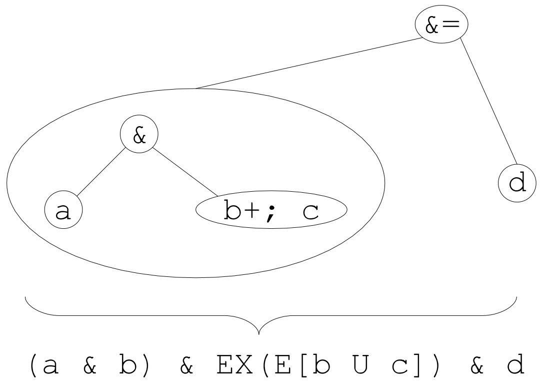


***Figure 4****: Logical and combined with path-oriented logical and*

### 2.4.4 Path-oriented operators

In this section we define two new operators. We call them path-oriented operators, because they change the meaning of the whole path. Although their usage breaks the linearity of the property transcription, they allow to efficiently express some specific properties for each state on the path.

**Path-oriented logical and (`&=`)**

This operator is modification of existing CTL operator `&` for formulas describing a number of consequent states. If we use it within the framework of one state, it has the same meaning as ordinary CTL operator.

Most common way of usage of this operator is:

*path expression* `&=` *state expression*

Where the *path expression* means the sequence of states.

The example formula means that in each state of *path expression* holds also formula given by *state expression*. Let us show the meaning of this operator on a small example.

Expression `a; b; c &= d` describes three consequent states. In the first one should hold property `a` and also property `d`, in the second state `b` and `d` and finally in the third state should hold properties `c` and `d`. Mentioned expression is in practice an abbreviation for expression `a & d; b & d; c & d`.

There are two possibilities how to explicate advanced sequencing operator when it is used in combination with path-oriented logical and. Let us remind the meaning of advanced sequencing operator. It says in general that there should hold some property in the current state, then there can be couple of states with no specified properties and after that should be a state satisfying some other property.

Let us focus on states with no specified properties. Should these states satisfy property given on the left side of path-oriented logical and? According to the definition of the advanced sequencing operator we can say no, they should not. This means that expression `a;+ b &= c` is equivalent to the CTL formula `(a & c) & EX(EF(b & c))`. In the praxis it is much more usable the interpretation where states with no specified properties also satisfy left side property of path-oriented logical and. We demonstrate the meaning of this case on the example. Expression `a;+ b &= c` is now equivalent to the following CTL formula `(a & c) & EX(E[c U b & c])`.

Similar to the combining of path-oriented logical and with advanced sequencing there are two possibilities of how to explicate combination of special sequencing operator and path-oriented logical and. Also for this operator we use the case where states with no specified properties should satisfy left side property of path-oriented and. Again we show how to rewrite expression including special sequencing operator to the CTL formula. For example expression `a, b &= c` has the same meaning as formula `a & E[c U b & c]`.

Note that we can also use a path expression instead of state expression. This means that there should start a path satisfying this expression in every state. We demonstrate the meaning on an example. When we write `a; b; c &= d; e; f` it is the same as `a & (d; e; f); b & (d; e; f); c & (d; e; f)` and the state-space in which holds such property looks like that in Figure 5.

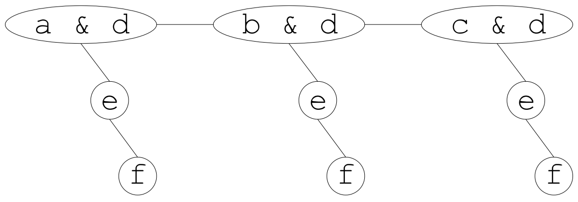


***Figure 5**: Path-oriented logical and – state space example*

In the same way as path-oriented logical and we can also define extension to every logical operator used in CTL. Such operators would be used very seldom and readability of resulting expressions would fall away.

**Explicit always (`A(…)`)**
As it was already mentioned all state separating operators say in general that there **exists** a path beginning in the current state which satisfy some property. We call this property as implicit existence. This implicit behavior can be changed via usage of newly defined operator explicit always. We show the usage and exact meaning of the operator on the following example.

The expression `a; A(b; c); d` which includes usage of explicit always operator means that there should hold property `a` in the current state, next that there should hold property `b` in all next states of the current state, after this should hold property `c` in all descendants of these states and finally all these descendant should have at least one next state in which holds property `d`. Verbal description of this expression is quite hard to understand so we show the equivalent formula in CTL and also the picture (Figure 6) which may clears up the meaning.

Equivalent formula in CTL looks like this: `a & AX(b & AX(c & EX(d)))`.

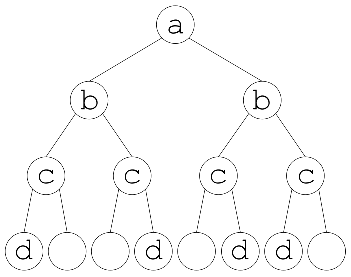


*Figure 6: Explicit always*

Chapter 3

# BP-CTL – formal description

## 3.1 Formal description of a temporal logic

There exist two ways how to formally describe a temporal logic. The first one is to formulate its own formal semantics and the second is to propose the exact transformation to another already defined logic with equivalent or stronger expressive power.

In this chapter we describe formal details of BP-CTL via both of this approaches. First we show how to rewrite general BP-CTL expression to the equivalent CTL formula. Then we introduce the BP-CTL formal semantics.

## 3.2 BP-CTL to CTL transformation

### 3.2.1 BP-CTL as CTL shortcuts

As it was described in details in the previous text, BP-CTL comes from temporal logic CTL and extends its expressions with a set of new operators. These are in fact the shortcuts for some (basic) composite CTL formulas. In this part we show how to rewrite newly defined operators to the pure CTL.

Let us remember in short BP-CTL operators. There are seven state separators and two specific path-oriented operators that modify the meaning of the given path.

### 3.2.2 State separators

First we describe the equivalent CTL formulas for state separating operators.

Let *S* be a set of BP-CTL expressions, which do not include any state separating operators, or more precisely expressions in which all included state separators are used inside subformulas of CTL operators. In such a case we at first recursively rewrite these subformulas – that include state separating operators – to the pure CTL so the resulting expression is without state separators. Because of the finiteness of CTL formulas and BP-CTL expressions, such a recursion is also finite.

Such expressions describe in BP-CTL properties which should hold in one state and hence we call them state expressions.

Next let $G$ denotes a set of general BP-CTL expressions, which means state expressions and also expressions which arise by combining state expressions and state separating operators. Because every BP-CTL expression must be finite, we can say, that $G = \{p, p_1 \circ_1 p_2, ..., p_1 \circ_1 p_2 \circ_2 \cdots \circ_{n-1} p_n\}$, where $\circ_i$ denotes one of the state separators, $p, p_1, ..., p_n \in S$ and $n \in \mathbb{N}$.

Let `p`, `q` and `r` be expressions from $S$, `P`, `Q` and `R` be expressions from $G$ (such that $\mathtt{P} = \mathtt{p}_1 \circ_1 \cdots \circ_{n-1} \mathtt{p}_n$, $\mathtt{Q} = \mathtt{q}_1 \circ_1 \cdots \circ_{n-1} \mathtt{q}_n$ and $\mathtt{R} = \mathtt{r}_1 \circ_1 \cdots \circ_{n-1} \mathtt{r}_n$) and expression `TRUE` denotes (state) expression which holds in any case.

Note that we omit the indexes by state separators ($\circ$) because they are usually obvious from the context.

State separators are right associative operators, which means that `p ∘ q ∘ r` should be interpreted as expression `p ∘ Q`, where `Q` denotes subexpression `q ∘ r`.

| **State separating operator** | **BP-CTL expression** | **Equivalent CTL formula** |
|---|---|---|
| state separating logical and | `p && q` | `p & q` |
| sequencing | `p; q` | `p & EX(q)` |
| special sequencing | `p, q` | `p & EF(q)` |
| advanced sequencing | `p;+ q` | `p & EX(EF(q))` |
| repetition (as unary op.) | `p*` | `E[p U TRUE]` |
| repetition | `p*; q` | `E[p U q]` |
| advanced repetition (unary) | `p+` | `p & EX(E[q U TRUE])` |
| advanced repetition | `p+; q` | `p & EX(E[p U q])` |
| infinite repetition (unary) | `p@` | `EG(p)` |
| infinite repetition | `p@; q` | `EG(p) \| E[p U q]` |

On the basis of the previous table and the right associativity of state separators we are able to rewrite any $G$ expression `P` to the pure CTL. Let us denote this basic transformation as function $f$(`P`). Then holds the relationship:

$$f(\mathtt{P}) = f(\mathtt{p}_1 \circ f(\mathtt{p}_2 \circ \cdots \circ f(\mathtt{p}_{n-1} \circ \mathtt{p}_n)...)).$$

### 3.2.3 Explicit always

The following table shows how to rewrite expression with state separators, when it is included in explicit always operator, to the CTL formula. Let us remember the interpretation of this operator:

Implicit existence of state separators (expressed as "...and there exist next state ...") is changed to explicit always ("...and for each of next states ...").

| State separators inside always operator expression | |
|---|---|
| **BP-CTL expression** | **Equivalent CTL formula** |
| `A(p && q)` | `p & q` |
| `A(p; q)` | `p & AX(q)` |
| `A(p, q)` | `p & AF(q)` |
| `A(p;+ q)` | `p & AX(AF(q))` |
| `A(p*)` | `A[p U TRUE]` |
| `A(p*; q)` | `A[p U q]` |
| `A(p+)` | `p & AX(A[p U TRUE])` |
| `A(p+; q)` | `p & AX(A[p U q])` |
| `A(p@)` | `AG(p)` |
| `A(p@; q)` | `AG(p) \| A[p U q]` |

Let us denote transformation of *G* expression `P` in the scope of explicit always operator as function $f_A$(`P`). Similar to the function *f* holds the following relationship:

$$f_A(\mathtt{P}) = f_A(\mathtt{p}_1 \circ f_A(\mathtt{p}_2 \circ \cdots \circ f_A(\mathtt{p}_{n-1} \circ \mathtt{p}_n)\ldots)).$$

In the next two tables we show how to interpret BP-CTL expressions which includes a *G* expression `Q` in the scope of explicit always operator as a subexpression (this means the type of expression described in the previous table).

At first, we focus on expressions including explicit always operator preceded by any state separator.

| State separators preceding always operator expression | |
|---|---|
| **BP-CTL expression** | **Equivalent CTL transformation** |
| `p && A(Q)` | `p &` $f_A$(`Q`) |
| `p; A(Q)` | `p & AX(`$f_A$(`Q`)`)` |
| `p, A(Q)` | `p & AF(`$f_A$(`Q`)`)` |
| `p;+ A(Q)` | `p & AX(AF(`$f_A$(`Q`)`))` |
| `p*; A(Q)` | `A[p U `$f_A$(`Q`)`]` |

| | |
|---|---|
| p+; A(Q) | p & AX(A[p U $f_A$(Q)]) |
| p@; A(Q) | AG(p) $\mid$ A[p U $f_A$(Q)] |

In general case holds that expression p $\circ$ A(Q) has to be interpreted via the following relationship:

$$f_A(p \circ Q).$$

Now we show how to rewrite expressions containing explicit always followed by some state separating operator.

| **Explicit always followed by state separators** | |
|---|---|
| **BP-CTL expression** | **Equivalent CTL transformation** |
| A(P) && Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (p_n \,\&\, f(Q)))\ldots)$ |
| A(P); Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (p_n \,\&\, EX(f(Q))))\ldots)$ |
| A(P), Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (p_n \,\&\, EF(f(Q))))\ldots)$ |
| A(P);+ Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (p_n \,\&\, EX(EF(f(Q)))))\ldots)$ |
| A(P)*; Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (E[p_n\, U\, f(Q)]))\ldots)$ |
| A(P)+; Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (EX(E[p_n\, U\, f(Q)])))\ldots)$ |
| A(P)@; Q | $f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ (EG(Q) \mid E[p_n\, U\, f(Q)]))\ldots)$ |

In general case holds that expression A(P) $\circ$ Q has to be rewritten to the pure CTL according to the following relationship:

$$f_A(p_1 \circ \cdots \circ f_A(p_{n-1} \circ f(p_n \circ Q))\ldots).$$

### 3.2.4 Path-oriented logical and

Path-oriented logical and is the last operator defined in BP-CTL. It says that in every state of the given path also a given property holds.

The following table shows how to rewrite expression containing state separators when they are used in the framework of path-oriented logical and to the equivalent CTL formula.

| **State separators inside path-oriented logical and** | |
|---|---|
| **BP-CTL expression** | **Equivalent CTL transformation** |
| p && q &= r | (p & r) & (q & r) |
| p; q &= r | (p & r) & EX(q & r) |
| p, q &= r | (p & r) & E[r U q & r] |

| | |
|---|---|
| p;+ q &= r | (p & r) & EX(E[r U q & r]) |
| p* &= r | E[p & r U r] |
| p*; q &= r | E[p & r U q & r] |
| p+ &= r | (p & r) & EX(E[p & r U r]) |
| p+; q &= r | (p & r) & EX(E[p & r U q & r]) |
| p@ &= r | EG(p & r) |
| p@; q &= r | EG(p & r) \| E[p & r U q & r] |

Let us denote state separators transformations in the framework of path-oriented logical and as function $f_{\&=}(\mathtt{P},\ \mathtt{r})$. Similar to the functions $f$ and $f_A$ holds the following relationship:

$$f_{\&=}(\mathtt{P},\ \mathtt{r}) = f_{\&=}(\mathtt{p}_1 \circ f_{\&=}(\mathtt{p}_2 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ \mathtt{p}_n,\ \mathtt{r})\ldots,\ \mathtt{r}),\ \mathtt{r}).$$

In the next two tables we show how to interpret expressions with path-oriented logical and operator in the context of state separating operators.

At first, we focus on expressions where state separator precedes the subexpression including path-oriented logical and.

| **State separators preceding path-oriented logical and** | |
|---|---|
| **BP-CTL expression** | **Equivalent CTL transformation** |
| p && (Q &= r) | p & $f_{\&=}$(Q, r) |
| p; (Q &= r) | p & EX($f_{\&=}$(Q, r)) |
| p, (Q &= r) | p & EF($f_{\&=}$(Q, r)) |
| p;+ (Q &= r) | p & EX(EF($f_{\&=}$(Q, r))) |
| p*; (Q &= r) | E[p U $f_{\&=}$(Q, r)] |
| p+; (Q &= r) | p & EX(E[p U $f_{\&=}$(Q, r)]) |
| p@; (Q &= r) | EG(p) \| E[p U $f_{\&=}$(Q, r)] |

In general holds that expression p ∘ (Q &= r) has to be interpreted according to the following relationship:

$$f(\mathtt{p} \circ f_{\&=}(\mathtt{Q},\ \mathtt{r})).$$

Now we show how to map expressions containing path-oriented logical and followed by some state separating operator.

| **Path-oriented logical and followed by state separators** | |
|---|---|
| **BP-CTL expression** | **Equivalent CTL transformation** |
| `(P &= q) && R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ (\mathtt{p}_n \ \&\ f(\mathtt{R})), \mathtt{q})\ldots, \mathtt{q})$ |
| `(P &= q); R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ (\mathtt{p}_n \ \&\ \mathtt{EX}(f(\mathtt{R}))), \mathtt{q})\ldots, \mathtt{q})$ |
| `(P &= q), R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ (\mathtt{p}_n \ \&\ \mathtt{E[q\ U}\ f(\mathtt{R})\mathtt{]}), \mathtt{q})\ldots, \mathtt{q})$ |
| `(P &= q);+ R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ (\mathtt{p}_n \ \&\ \mathtt{EX(E[q\ U}\ f(\mathtt{R})\mathtt{])}), \mathtt{q})\ldots, \mathtt{q})$ |
| `(P &= q)*; R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-2} \circ f(\mathtt{p}_{n-1} \circ (\mathtt{E[p}_n \ \&\ \mathtt{q\ U}\ f(\mathtt{R})\mathtt{]})), \mathtt{q})\ldots, \mathtt{q})$ |
| `(P &= q)+; R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ (\mathtt{p}_n \ \&\ \mathtt{EX(E[p}_n \ \&\ \mathtt{q\ U}\ f(\mathtt{R})\mathtt{])}), \mathtt{q})\ldots, \mathtt{q})$ |
| `(P &= q)@; R` | $f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-2} \circ f(\mathtt{p}_{n-1} \circ (\mathtt{EG(p}_n \ \&\ \mathtt{q)} \mid \mathtt{E[p}_n \ \&\ \mathtt{q\ U} f(\mathtt{R})\mathtt{]})), \mathtt{q})\ldots, \mathtt{q})$ |

For this type of expression does not generally hold that expression `(P &= q) ∘ R` should be interpreted via relationship:

$$f_{\&=}(\mathtt{p}_1 \circ \cdots \circ f_{\&=}(\mathtt{p}_{n-1} \circ f(\mathtt{p}_n \circ \mathtt{R}), \mathtt{q})\ldots, \mathtt{q}).$$

It does not hold for expressions `(P &= q)*; R` and `(P &= q)@; R`, where the last *S* subexpression of *G* expression `P` ($\mathtt{p}_n$) together with expression `q` may not be included in state space satisfying whole expression.

## 3.3 Formal semantics of BP-CTL

### 3.3.1 BP-CTL elements

BP-CTL expression consists of:

- atomic propositions,
- original CTL operators
  logical `!` (not), `&` (and), `|` (or),
  path quantifiers `E` (exists) and `A` (always) followed by
  temporal operator `X` (next), `F` (future), `G` (global) or `U` (until),
- a set of newly defined operators together denoted as state separators `&&` (state separating logical and), `;` (sequencing), `;+` (advanced sequencing), `,` (special sequencing), `*` (repetition), `+` (advanced repetition), `@` (infinite repetition) and
- two path-oriented operators `A(…)` (explicit always) and `&=` (path-oriented logical and).

### 3.3.2 Expressions

There are two types of expressions in BP-CTL: *state expressions* (which hold in a specific state) and *path expressions* (which hold along a specific path). Moreover state expressions have three subtypes: standard (denoted simply as state expressions), extended (e-state expressions) and path-like (p-state expressions). The difference between state expressions and e-state expressions and also between path expressions and p-state expressions is, simply said, in the fact whether they include explicit always or path-oriented logical and operators or not. The precise description follows.

Let *AP* be the set of atomic proposition names. The syntax of state and e-state expressions is given by the following rules:

- if $p \in AP$, then $p$ is a state expression,
- if $g$ and $h$ are path expressions, then
  `!`$g$, $g$ `&` $h$, $g$ `|` $h$,
  `EX(`$g$`)`, `EF(`$g$`)`, `EG(`$g$`)`, `E[`$g$ `U` $h$`]`,
  `AX(`$g$`)`, `AF(`$g$`)`, `AG(`$g$`)`, `A[`$g$ `U` $h$`]` are state expressions,
- if $f$ is a state expression, then $f$ is also e-state expression,
- if $g$ and $h$ are p-state expressions, then
  `!`$g$, $g$ `&` $h$, $g$ `|` $h$,
  `EX(`$g$`)`, `EF(`$g$`)`, `EG(`$g$`)`, `E[`$g$ `U` $h$`]`,
  `AX(`$g$`)`, `AF(`$g$`)`, `AG(`$g$`)`, `A[`$g$ `U` $h$`]` are e-state expressions.

The syntax of path expressions and p-state expressions is given by the following rules:

- if $f$ is a state expression, then $f$ is also path expression,
- if $f$ is state expression and $g$ is path expression, then
  $f$ `&&` $g$, $f$`;` $g$, $f$`,` $g$, $f$`;+` $g$,
  $f$`*`, $f$`*;` $g$,
  $f$`+`, $f$`+;` $g$,
  $f$`@` and $f$`@;` $g$ are path expressions,
- if $f$ is a e-state expression and $g$ is a path expression, then both $f$ and $g$ are also p-state expressions,
- if $f$ is e-state expression, $g$ is p-state expression and $h$ and $i$ are path expressions, then
  $f$ `&&` $g$, $f$`;` $g$, $f$`,` $g$, $f$`;+` $g$,
  $f$`*`, $f$`*;` $g$,
  $f$`+`, $f$`+;` $g$,
  $f$`@`, $f$`@;` $g$,
  `A(`$h$`)`, `A(`$h$`) &&` $g$, `A(`$h$`);` $g$, `A(`$h$`),` $g$, `A(`$h$`);+` $g$,
  `A(`$h$`)*;` $g$, `A(`$h$`)+;` $g$, `A(`$h$`)@;` $g$,
  $h$ `&=` $i$, `(`$h$ `&=` $i$`) &&` $g$, `(`$h$ `&=` $i$`);` $g$, `(`$h$ `&=` $i$`),` $g$, `(`$h$ `&=` $i$`);+` $g$,
  `(`$h$ `&=` $i$`)*;` $g$, `(`$h$ `&=` $i$`)+;` $g$ and `(`$h$ `&=` $i$`)@;` $g$ are p-state expressions.

Note that each path expression $g$ has the following schema:

$$f_0^g \circ_0^g f_1^g \circ_1^g \cdots \circ_{n-1}^g f_n^g,$$

where $n$ is a finite number which denotes the *length of path expression* $g$, $f_i^g$ are state expressions and $\circ_i^g$ denote state separating operators. Remember that $f$`*` – when used as the last element in the expression – is interpreted as $f$`*; TRUE`. Similar $f$`+` is interpreted as $f$`+; TRUE` and $f$`@` as $f$`@; FALSE`.

## 3.3.3 Semantics of the expressions

A *Kripke structure* is a 5-tuple $(S, I, P, L, T)$ with $S$ a finite set of states, $I \subseteq S$ a set of initial states, $P$ a finite set of atomic propositions, $L$: $S \rightarrow 2^P$ a labeling function, and $T \subseteq S \times S$ a transition relation. Let $M(S, I, P, L, T)$ be a Kripke structure. A path in $M$ is a (finite or infinite) sequence of states $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ such that $\forall i \geq 0$: $(\pi_i, \pi_{i+1}) \in T$. We use the notation $\pi^n$ for the suffix of $\pi$ which begins at state $\pi_n$. Similarly, we use the notation ${}^n\pi$ for the prefix of $\pi$ which ends at state $\pi_n$ (for example ${}^0\pi$ denotes only the first state of the path – $\pi_0$). Note that ${}^\infty\pi$ denotes an infinite path. Finally ${}^n\pi^m$ denotes the sequence of states $\pi_m \rightarrow \pi_{m+1} \rightarrow \ldots \rightarrow \pi_n$.

If $f$ is a state expression, the notation $M, s \vDash f$ means, that $f$ holds at state $s$ in the Kripke structure $M$. Similarly, if $f$ is a path expression, $M, \pi \vDash f$ means that $f$ holds along path $\pi$ in Kripke structure $M$. The relation $\vDash$ is defined inductively as follows (assuming, that $p$ is atomic proposition and $g_1$ and $g_2$ are path expressions):

| | |
|---|---|
| $M, s \vDash p$ | $\Leftrightarrow$ $p \in L(s)$ |
| $M, s \vDash \texttt{!}\,g_1$ | $\Leftrightarrow$ not $M, s \vDash g_1$ |
| $M, s \vDash g_1 \;\texttt{\&}\; g_2$ | $\Leftrightarrow$ $M, s \vDash g_1$ and $M, s \vDash g_2$ |
| $M, s \vDash g_1 \;\texttt{\|}\; g_2$ | $\Leftrightarrow$ $M, s \vDash g_1$ or $M, s \vDash g_2$ |
| $M, s \vDash \texttt{EX(}g_1\texttt{)}$ | $\Leftrightarrow$ there exist a state $t$ and a transition $s \rightarrow t$ in $M$ such that $M, t \vDash g_1$ |
| $M, s \vDash \texttt{EF(}g_1\texttt{)}$ | $\Leftrightarrow$ there exist a state $t$ and a path from $s$ to $t$ in $M$ such that $M, t \vDash g_1$ |
| $M, s \vDash \texttt{EG(}g_1\texttt{)}$ | $\Leftrightarrow$ there exists an infinite path $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ in $M$ such that $\pi_0 = s$ and $\forall i \geq 0$: $M, \pi_i \vDash g_1$ |
| $M, s \vDash \texttt{E[}g_1 \texttt{ U } g_2\texttt{]}$ | $\Leftrightarrow$ there exists a path $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ in $M$ such that $\pi_0 = s$ and $\exists i \geq 0$ such that $M, \pi_i \vDash g_2$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \vDash g_1$ |
| $M, s \vDash \texttt{AX(}g_1\texttt{)}$ | $\Leftrightarrow$ for every state $t$ in $M$ such that $s \rightarrow t$, $M, t \vDash g_1$ holds |
| $M, s \vDash \texttt{AF(}g_1\texttt{)}$ | $\Leftrightarrow$ on every infinite path in $M$ beginning in $s$ there is a state $t$ such that $M, t \vDash g_1$ |
| $M, s \vDash \texttt{AG(}g_1\texttt{)}$ | $\Leftrightarrow$ for every infinite path $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ in $M$ such that $\pi_0 = s$: $\forall i \geq 0$: $M, \pi_i \vDash g_1$ |
| $M, s \vDash \texttt{A[}g_1 \texttt{ U } g_2\texttt{]}$ | $\Leftrightarrow$ for every infinite path $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ in $M$ such that $\pi_0 = s$: $\exists i \geq 0$ such that $M, \pi_i \vDash g_2$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \vDash g_1$ |

For the semantics of e-state expressions hold exactly the same rules as for the semantics of state expressions (with the difference that the first rule which defines the validity of atomic proposition is included in the fact that each state expression is also e-state expression and $g_1$ and $g_2$ denotes p-state expressions instead of path-expressions). Note, that the semantics of state expressions is in fact the semantics of pure CTL.

The validity of path expressions is defined as follows (assuming, that $f$ is a state expression, $g$ is a path expression and $n \geq 0$):

| | |
|---|---|
| $M, \pi \vDash g$ | $\Leftrightarrow$ there exists $n$ such that $M, {}^n\pi \vDash g$ |
| $M, {}^n\pi \vDash f$ | $\Leftrightarrow$ $n = 0$, $\pi_0 = s$ and $M, s \vDash f$ |
| $M, {}^n\pi \vDash f\ \texttt{\&\&}\ g$ | $\Leftrightarrow$ $M, {}^0\pi \vDash f$ and $M, {}^n\pi \vDash g$ |
| $M, {}^n\pi \vDash f\texttt{;}\ g$ | $\Leftrightarrow$ $M, {}^0\pi \vDash f$ and $M, {}^n\pi^1 \vDash g$ |
| $M, {}^n\pi \vDash f\texttt{,}\ g$ | $\Leftrightarrow$ $M, {}^0\pi \vDash f$ and $\exists i \geq 0$: $M, {}^n\pi^i \vDash g$ |
| $M, {}^n\pi \vDash f\texttt{;+}\ g$ | $\Leftrightarrow$ $M, {}^0\pi \vDash f$ and $\exists i \geq 1$: $M, {}^n\pi^i \vDash g$ |
| $M, {}^n\pi \vDash f\texttt{*}$ | $\Leftrightarrow$ $n \geq -1$ and $\forall i$, $0 \leq i \leq n$: $M, \pi_i \vDash f$ (${}^{-1}\pi$ denotes empty path) |
| $M, {}^n\pi \vDash f\texttt{*;}\ g$ | $\Leftrightarrow$ $\exists i \geq 0$ such that $M, {}^n\pi^i \vDash g$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \vDash f$ |
| $M, {}^n\pi \vDash f\texttt{+}$ | $\Leftrightarrow$ $\forall i$, $0 \leq i \leq n$: $M, \pi_i \vDash f$ |
| $M, {}^n\pi \vDash f\texttt{+;}\ g$ | $\Leftrightarrow$ $M, {}^0\pi \vDash f$ and $M, {}^n\pi^1 \vDash f\texttt{*;}\ g$ |
| $M, {}^n\pi \vDash f\texttt{@}$ | $\Leftrightarrow$ $n = \infty$ and $\forall i \geq 0$: $M, \pi_i \vDash f$ |
| $M, {}^n\pi \vDash f\texttt{@;}\ g$ | $\Leftrightarrow$ $M, {}^n\pi \vDash f\texttt{@}$ or $M, {}^n\pi \vDash f\texttt{*;}\ g$ |

The p-state expressions are specific expressions. Although they describe paths (this is the reason for their "path-like" name) their meaning has to be defined for states because they can describe more than one path simultaneously. Let $f$ be a e-state expression, $g_1$ and $g_2$ be path expressions and $h$ be a p-state expression, then:

| | |
|---|---|
| $M, s \vDash g_1$ | $\Leftrightarrow$ there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and $M, \pi \vDash g_1$ |
| $M, s \vDash f\ \texttt{\&\&}\ h$ | $\Leftrightarrow$ $M, s \vDash f$ and $M, s \vDash h$ |
| $M, s \vDash f\texttt{;}\ h$ | $\Leftrightarrow$ $M, s \vDash f$ and there exist a state $t$ and a transition $s \to t$ in $M$ such that $M, t \vDash h$ |
| $M, s \vDash f\texttt{,}\ h$ | $\Leftrightarrow$ $M, s \vDash f$ and there exist a state $t$ and a path from $s$ to $t$ in $M$ such that $M, t \vDash h$ |
| $M, s \vDash f\texttt{;+}\ h$ | $\Leftrightarrow$ $M, s \vDash f$ and there exist a state $t \neq s$ and a path from $s$ to $t$ in $M$ such that $M, t \vDash h$ |
| $M, s \vDash f\texttt{*}$ | $\Leftrightarrow$ there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and $\exists i \geq 0$ such that $\forall j$, $0 \leq j < i$: $M, \pi_j \vDash f$ |
| $M, s \vDash f\texttt{*;}\ h$ | $\Leftrightarrow$ there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and $\exists i \geq 0$ such that $M, \pi_i \vDash h$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \vDash f$ |
| $M, s \vDash f\texttt{+}$ | $\Leftrightarrow$ $M, s \vDash f$ and there exist a state $t$ and a transition $s \to t$ in $M$ such that $M, t \vDash f\texttt{*}$ |
| $M, s \vDash f\texttt{+;}\ h$ | $\Leftrightarrow$ $M, s \vDash f$ and there exist a state $t$ and a transition $s \to t$ in $M$ such that $M, t \vDash f\texttt{*;}\ h$ |
| $M, s \vDash f\texttt{@}$ | $\Leftrightarrow$ there exists an infinite path $\pi$ in $M$ such that $\pi_0 = s$ and $\forall i \geq 0$: $M, \pi_i \vDash f$ |
| $M, s \vDash f\texttt{@;}\ h$ | $\Leftrightarrow$ $M, s \vDash f\texttt{@}$ or $M, s \vDash f\texttt{*;}\ h$ |
| $M, s \vDash \texttt{A(}g_1\texttt{)}$ | $\Leftrightarrow$ for every infinite path $\pi$ in $M$ such that $\pi_0 = s$: $M, \pi \vDash g_1$ |
| $M, s \vDash f\texttt{;}\ \texttt{A(}g_1\texttt{)}$ | $\Leftrightarrow$ $M, s \vDash f$ and for every $t$ in $M$ such that $s \to t$: $M, t \vDash \texttt{A(}g_1\texttt{)}$ |
| $M, s \vDash f\texttt{,}\ \texttt{A(}g_1\texttt{)}$ | $\Leftrightarrow$ $M, s \vDash f$ and on every infinite path in $M$ beginning in $s$ there is a state $t$ such that $M, t \vDash \texttt{A(}g_1\texttt{)}$ |
| $M, s \vDash f\texttt{;+}\ \texttt{A(}g_1\texttt{)}$ | $\Leftrightarrow$ $M, s \vDash f$ and on every infinite path in $M$ beginning in $s$ there is a state $t \neq s$ such that $M, t \vDash \texttt{A(}g_1\texttt{)}$ |
| $M, s \vDash f\texttt{*;}\ \texttt{A(}g_1\texttt{)}$ | $\Leftrightarrow$ on every infinite path $\pi$ in $M$ such that $\pi_0 = s$: $\exists i \geq 0$ such that $M, \pi_i \vDash \texttt{A(}g_1\texttt{)}$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \vDash f$ |
| $M, s \vDash f\texttt{+;}\ \texttt{A(}g_1\texttt{)}$ | $\Leftrightarrow$ $M, s \vDash f$ and for every $t$ in $M$ such that $s \to t$: $M, t \vDash f\texttt{*;}\ \texttt{A(}g_1\texttt{)}$ |

| | | |
|---|---|---|
| $M, s \models \texttt{A(}g_1\texttt{) \&\&}\ h$ | ⇔ | for every infinite path $\pi$ in $M$ such that $\pi_0 = s$ either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and $M, \pi_n \models h$ or $M, {}^\infty\pi \models g_1$ |
| $M, s \models \texttt{A(}g_1\texttt{) ;}\ h$ | ⇔ | for every infinite path $\pi$ in $M$ such that $\pi_0 = s$ either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and there exist a state $t$ and a transition $\pi_n \rightarrow t$ in $M$ such that $M, t \models h$ or $M, {}^\infty\pi \models g_1$ |
| $M, s \models \texttt{A(}g_1\texttt{) ,}\ h$ | ⇔ | for every infinite path $\pi$ in $M$ such that $\pi_0 = s$ either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and there exist a state $t$ and a path from $\pi_n$ to $t$ in $M$ such that $M, t \models h$ or $M, {}^\infty\pi \models g_1$ |
| $M, s \models \texttt{A(}g_1\texttt{) ;+}\ h$ | ⇔ | for every infinite path $\pi$ in $M$ such that $\pi_0 = s$ either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and there exist a state $t \neq \pi_n$ and a path from $\pi_n$ to $t$ in $M$ such that $M, t \models h$ or $M, {}^\infty\pi \models g_1$ |
| $M, s \models \texttt{A(}g_1\texttt{)+;}\ h$ | ⇔ | for every infinite path $\pi$ in $M$ such that $\pi_0 = s$ either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and there exists a path $\rho$ in $M$ such that $\rho_0 = \pi_n$ and $\exists i > 0$ such that $M, \rho_i \models h$ and $\forall j$, $0 < j < i$: $M, \rho_j \models f_m^{g_1}$ ($m$ is length of the expression $g_1$) or $M, {}^\infty\pi \models g_1$ |
| $M, s \models \texttt{A(}g_1\texttt{)*;}\ h$ | ⇔ | $M, s \models \texttt{A(}g_1\texttt{)+;}\ h$ or $M, s \models \texttt{A(}\ f_0^{g_1} \circ_0^{g_1} f_1^{g_1} \circ_1^{g_1} \cdots \circ_{m-2}^{g_1} f_{m-1}^{g_1} \circ_{m-1}^{g_1}\ \texttt{TRUE) \&\&}\ h$ ($m$ is length of the expression $g_1$) |
| $M, s \models \texttt{A(}g_1\texttt{)@;}\ h$ | ⇔ | $M, s \models \texttt{A(}g_1\texttt{)*;}\ h$ or for every infinite path $\pi$ in $M$ such that $\pi_0 = s$ there exists finite $n$ such that $M, {}^n\pi \models g_1$ and there exists a path $\rho$ in $M$ such that $\rho_0 = \pi_n$ and $\forall i$, $i \geq 0$: $M, \rho_i \models f_m^{g_1}$ ($m$ is length of the expression $g_1$) |
| $M, s \models g_1\ \texttt{\&=}\ g_2$ | ⇔ | there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and there exists $n$ such that $M, {}^n\pi \models g_1$ and $\forall i$, $0 \leq i \leq n$: $M, \pi_i \models g_2$ |
| $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{) \&\&}\ h$ | ⇔ | there exist a path $\pi$ in $M$ such that $\pi_0 = s$ and either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and $\forall i$, $0 \leq i \leq n$: $M, \pi_i \models g_2$ and $M, \pi_n \models h$ or $M, {}^\infty\pi \models g_1$ and $\forall i \geq 0$: $M, \pi_i \models g_2$ |
| $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{) ;}\ h$ | ⇔ | there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and $\forall i$, $0 \leq i \leq n$: $M, \pi_i \models g_2$ and $M, \pi_{n+1} \models h$ or $M, {}^\infty\pi \models g_1$ and $\forall i \geq 0$: $M, \pi_i \models g_2$ |
| $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{) ,}\ h$ | ⇔ | there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and $\exists i \geq n$: $M, \pi_i \models h$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \models g_2$ or $M, {}^\infty\pi \models g_1$ and $\forall i \geq 0$: $M, \pi_i \models g_2$ |
| $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{) ;+}\ h$ | ⇔ | there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and $\exists i > n$: $M, \pi_i \models h$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \models g_2$ or $M, {}^\infty\pi \models g_1$ and $\forall i \geq 0$: $M, \pi_i \models g_2$ |
| $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{)+;}\ h$ | ⇔ | there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and either there exists finite $n$ such that $M, {}^n\pi \models g_1$ and $\exists i > n$: $M, \pi_i \models h$ and $\forall j$, $n < j < i$: $M, \pi_j \models f_m^{g_1}$ ($m$ is length of the expression $g_1$) and $\forall k$, $0 \leq k < i$: $M, \pi_k \models g_2$ or $M, {}^\infty\pi \models g_1$ and $\forall i \geq 0$: $M, \pi_i \models g_2$ |
| $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{)*;}\ h$ | ⇔ | $M, s \models \texttt{(}g_1\ \texttt{\&=}\ g_2\texttt{)+;}\ h$ or $M, s \models \texttt{(}\ f_0^{g_1} \circ_0^{g_1} f_1^{g_1} \circ_1^{g_1} \cdots \circ_{m-2}^{g_1} f_{m-1}^{g_1}\ \texttt{\&=}\ g_2\texttt{)}\ \circ_{m-1}^{g_1}\ h$ ($m$ is length of the expression $g_1$) |

$M, s \vDash$ `(`$g_1$ `&=` $g_2$`)@;` $h$ $\Leftrightarrow$ $M, s \vDash$ `(`$g_1$ `&=` $g_2$`)*;` $h$ or there exists a path $\pi$ in $M$ such that $\pi_0 = s$ and there exists finite $n$ such that $M, {}^n\pi \vDash g_1$ and $\forall i, i > n$: $M, \pi_i \vDash f_m^{g_1}$ ($m$ is length of the expression $g_1$) and $\forall k \geq 0$: $M, \pi_k \vDash g_2$

For better insight we show the difference amongst the meaning of $M, s \vDash$ `A(`$g_1$`)+;` $h$, $M, s \vDash$ `A(`$g_1$`)*;` $h$ and $M, s \vDash$ `A(`$g_1$`)+;` $h$ on the following pictures.

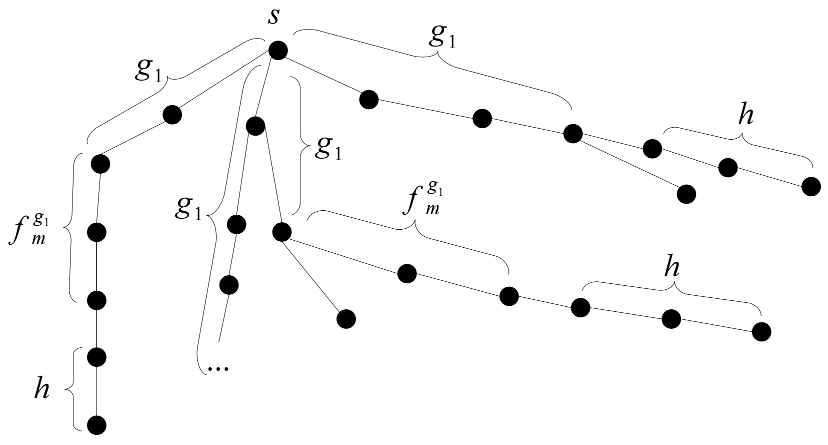
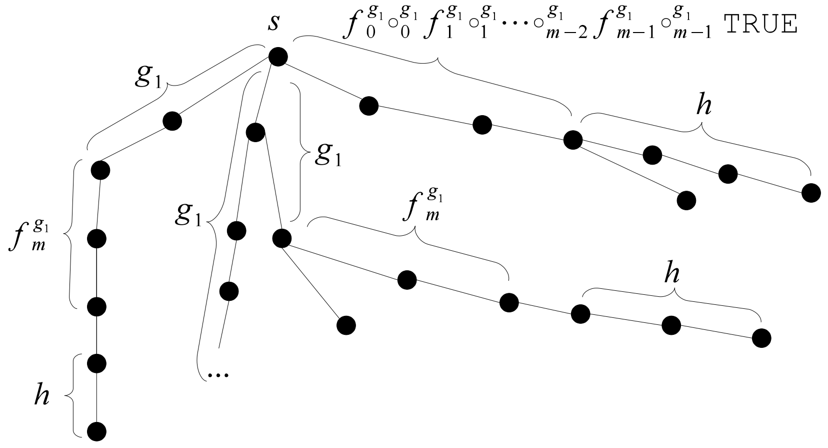


***Figure** 7: Example of the state space, which corresponds to the Kripke structure M such that M, s* $\vDash$ `A(`$g_1$`)+;` *h*



$s$
$f_0^{g_1} \circ_0^{g_1} f_1^{g_1} \circ_1^{g_1} \cdots \circ_{m-2}^{g_1} f_{m-1}^{g_1} \circ_{m-1}^{g_1}$ TRUE
$g_1$
$g_1$
$g_1$
$h$
$f_m^{g_1}$
$f_m^{g_1}$
$h$
$h$
...

***Figure** 8: Example of the state space, which corresponds to the Kripke structure M such that M, s* $\vDash$ `A(`$g_1$`)*;` *h*

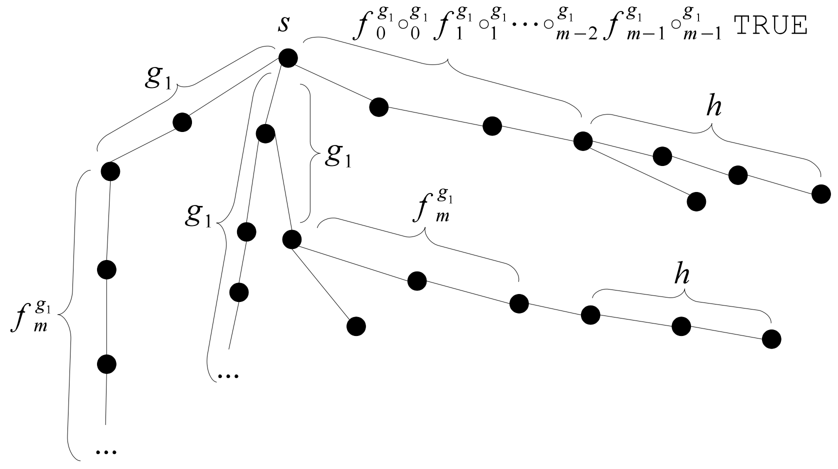

$s$
$f_0^{g_1} \circ_0^{g_1} f_1^{g_1} \circ_1^{g_1} \cdots \circ_{m-2}^{g_1} f_{m-1}^{g_1} \circ_{m-1}^{g_1}$ TRUE
$g_1$
$h$
$g_1$
$g_1$
$f_m^{g_1}$
$h$
$f_m^{g_1}$
...
...

***Figure 9****: Example of the state space, which corresponds to the Kripke structure M such that M, s* $\vDash$ `A(`$g_1$`)@;` *h*

Chapter 4

# Usability demonstration

## 4.1 NuSMV preprocessor

As a part of this thesis was developed a preprocessor for NuSMV symbolic model checker [8], called bpctl, which accepts extended NuSMV source files and transforms them to common NuSMV source files. The extension of the input language is new `BPSPEC` section, that enables to use BP-CTL as an input language for specifying checking properties as a part of NuSMV source file. The bpctl reads the NuSMV source file extended with properties written in BP-CTL and transforms all included `BPSPEC` sections to `SPEC` sections with equivalent CTL formulas (according to the transcription rules described in Appendix A).

The preprocessor combined with NuSMV model checker is functional solution for using BP-CTL in praxis. The preprocessor does not depend on any concrete version of NuSMV. It appears from the basic syntax of NuSMV source files, which is settled down since SMV – the NuSMV ancestor. Thus, bpctl will be probably compatible also with future – more optimized – versions of the model checker.

For more information about the NuSMV preprocessor see the user manual in Appendix C.

## 4.2 Crossroads simulation

In this section we show the usage of BP-CTL on the completely processed example demonstrating the advantages of BP-CTL with comparison to the CTL. It clearly shows that the properties written in BP-CTL are much more readable the CTL equivalents. The example includes word description of a real situation and formulation of the properties, that we want to check. Next we show the model representing the situation and BP-CTL transcription of the properties. For comparison we show also the equivalent CTL formulas.

Let us imagine a crossroads where the traffic is regulated by semaphores. The first road goes in the east-west direction and the second road goes in the north-south direction as shown in Figure 10.

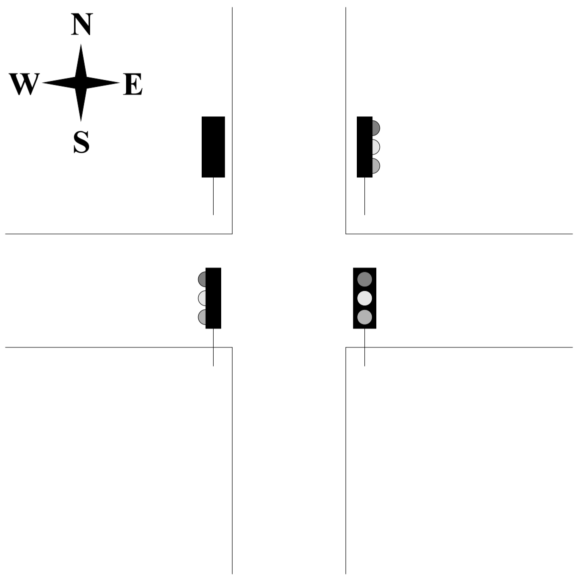


***Figure 10**: Crossroads schema*

Each semaphore on the crossroads has three colors (phases) with specific meaning: red (stop), yellow (attention, the phase is changing) and green (free). The possible color changes are: red → red and yellow together → green → yellow → red. The opposite semaphores (semaphores for one direction) are always in the same phase.

There is a software which should service the semaphores on the crossroads and we want to check some properties to be sure of the software correctness.

### 4.2.1 The model

Now we show the purpose of the model implementation in NuSMV input language. The model is represented via instances of five modules.

The `main` module representing the crossroads:

```
MODULE main
VAR
  ew: direction;  -- East-West road
  ns: direction;  -- North-South road

  EW_ToG: process ToGreen(ew);
  NS_ToG: process ToGreen(ns);

  EW_ToY: process ToYellow(ew, ns);
  NS_ToY: process ToYellow(ns, ew);

  EW_ToR: process ToRed(ew);
  NS_ToR: process ToRed(ns);
```

```
ASSIGN
  init(ew.turn) := {TRUE, FALSE};
  init(ns.turn) := !ew.turn;

FAIRNESS
  running;
```

The module `direction` represents two opposite semaphores in one direction – they are always in the same phase (stop, attention, free):

```
MODULE direction
VAR
  red:    boolean;        -- red light
  yellow: boolean;        -- yellow light
  green:  boolean;        -- green light
  turn:   boolean;        -- will be this direction next free
ASSIGN
  init(red)    := TRUE;   -- initial color is red
  init(yellow) := FALSE;
  init(green)  := FALSE;
```

The instances of the following three modules represent simultaneous processes changing the current color on the semaphores. The module `ToGreen` represents process that changes the color from “red and yellow” to “green”.

```
MODULE ToGreen(dir)
ASSIGN
  next(dir.red) :=
  case
    dir.red & dir.yellow: FALSE;
    TRUE:                 dir.red;
  esac;
  next(dir.yellow) :=
  case
    dir.red & dir.yellow: FALSE;
    TRUE:                 dir.yellow;
  esac;
  next(dir.green) :=
  case
    dir.red & dir.yellow: TRUE;
    TRUE:                 dir.green;
  esac;
FAIRNESS
  running
```

The module `ToYellow` changes the color of the semaphore from “red” to “red and yellow” and from “green” to “yellow” depending on the current value of other variables.

```
MODULE ToYellow(dir1, dir2)
ASSIGN
  next(dir1.yellow) :=
  case
    dir2.red & dir1.turn: TRUE;
    TRUE:                 dir1.yellow;
  esac;
  next(dir1.green) :=     FALSE;
  next(dir1.turn) :=
  case
    dir1.green:           FALSE;
    TRUE:                 dir1.turn;
  esac;
  next(dir2.turn) :=
  case
    dir1.green:           TRUE;
    TRUE:                 dir2.turn;
  esac;
FAIRNESS
  running
```

The module `ToRed` changes the color on the semaphore from “yellow” to “red”.

```
MODULE ToRed(dir)
ASSIGN
  next(dir.red) :=
  case
    dir.yellow:           TRUE;
    TRUE:                 dir.red;
  esac;
  next(dir.yellow) :=
  case
    !dir.red:             FALSE;
    TRUE:                 dir.yellow;
  esac;
FAIRNESS
  running
```

We refer the reader to [8] and [9] for more information about the format of NuSMV input language.

### 4.2.2 Checked properties

The first property we want to check is fundamental. Any time the red color has to be on the semaphores in at least one direction which means that simultaneous cars can go at most in one direction (EW or NS). Such property can be expressed in CTL as

```
SPEC  -- Safety property
  AG(ew.red | ns.red)
```

or

```
SPEC  -- Safety property
  !EF(!ew.red && !ns.red).
```

These formulas are too simple to have a specific (which means different from pure CTL) transcription in BP-CTL.

The next property we want to check is whether the colors on the semaphore change in the right order. This means from “red” to “red and yellow” and “green” and back from “green” to “yellow” and further from “yellow” to “red”. This property is much more complicated than the previous one. It can be expressed in CTL as

```
SPEC  -- Semaphore lights cycle is correct
  AG(AF(red &                -- red
    AX(A[red U               -- still red until...
      (red & yellow) &       -- red and yellow
      AX(A[(red & yellow) U  -- still red and yellow until...
        green &              -- green
        AX(A[green U         -- still green until...
          yellow &           -- yellow
          AX(A[yellow U      -- still yellow until...
            red])])])])      -- red
  )).
```

Equivalent BP-CTL expression is

```
BPSPEC  -- Semaphore lights cycle is correct
  AG(AF(A(red+; (red & yellow)+; green+; yellow+; red))).
```

We can see that the BP-CTL transcription of the same property is significantly more readable than in CTL.

The last property we want to check is whether the direction with green color on the semaphores is regularly changing. Which means once is green for north–south direction and once for east–west direction. Such a property can be in CTL expressed as

```
SPEC  -- The ns lights does not cycle twice with no ew green
  !EF(ns.green &     -- ns is green
    E[ew.red U       -- ew is red until...
      ns.red &       -- ns becomes red
      E[ew.red U     -- ew is still red until...
        ns.green]]   -- ns becomes green
  ).
```

The equivalent BP-CTL expression is

```
BPSPEC  -- The ns lights does not cycle twice with no ew green
  !EF(ns.green, ns.red, ns.green &= ew.red).
```

Again we can see that BP-CTL expression is much more direct and more readable.

Note, that the implementation of this example can be found also on the CD enclosed to this thesis.

## 4.3 Common specification patterns

There are some commonly occurring patterns in the property specifications of concurrent and reactive systems. On the basis of this fact Specification Patterns project [5, 6, 7] originates in the department of Computing and Information Sciences at Kansas State University. Authors of this on-line repository collect such patterns and rewrite their imprecise word description into the precise statements in common formal specification languages. Thanks to this, people do not have to reason about notation of these basic properties and may use error free records of this repository. Currently they supply patterns mapping for five formalism (LTL, CTL, GIL, QRE, INCA). There exist also mappings for additional formalisms developed by other researches (ACTL, RAFMC).

To demonstrate the usability of BP-CTL we decided to map all the patterns from the repository to BP-CTL. The mapping tables can be found in Appendix B.

The information in the patterns can be presented in a variety of ways. One organization, illustrated in Figure 11, is based on classifying the patterns in terms of the kinds of system behaviors they describe.

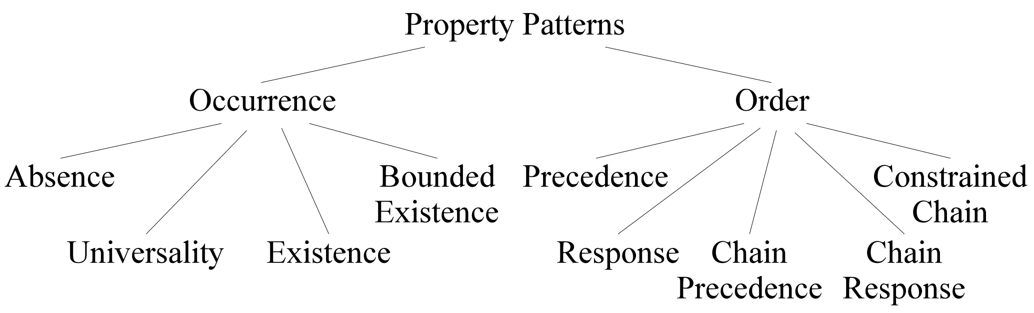


***Figure 11**: Property Patterns organization*

**Occurrence Patterns** describe the occurrence of a given event/state during system execution.

**Order Patterns** describe relative order in which multiple events/states occur during system execution.

The mappings to most of formalisms are in the patterns repository extended with scopes. The scope of a pattern is the extent of the program execution over which the pattern must hold. There are five basic kinds of scopes: global (the entire program execution), before (the execution up to a given state/event), after (the execution after given state/event), between (any part of the execution from one given state/event to another given state/event) and after–until (like between but designated part of the execution continues even if the second state/event does not occur). The scope is determined by specifying a starting and an ending state/event for the pattern.

For state-delimited scopes, the interval in which the property is evaluated is closed at the left and open at the right end. Thus, the scope consists of all states beginning with the starting state and up to but not including the ending state. Authors of the repository choose closed-left open–right scopes because they are relatively easy to encode in specifications and they work for the real property specifications they studied. The experiences indicate that most informal requirements are specified as properties of program executions or segments of program executions. Thus a pattern system for properties mirrors this view by scopes to enhance usability.

Now we show one example pattern from the repository. It is "Bounded Existence Property Pattern" in the scope "after–until".

To describe a portion of a system's execution that contains at most a specified number of instances of designated state transition or event. Bounded overtaking properties can be naturally expressed using instances of this pattern. For example, if we wish to say that process 1 can enter its critical region at most twice while process 2 is waiting to enter its region we would use a between scope (delimited by process 2 entering and exiting its waiting region) with 2-bounded existence for process 1 entering its critical region.

In these mapping we illustrate one instance of the bounded existence pattern, where the bound is at most 2 designated states. Other bounds can be specified by variations on this mapping.

Word description of the pattern is "Transitions to P-states occur at most 2 times".

| | |
|---|---|
| After Q until R | `AG(Q -> !E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & !R))]))]))])` |
| | `!EF(Q, !P; P, !P; P, !P; P &= !R)` |

The top cell includes the pure CTL transcription of such pattern and in the bottom one is the BP-CTL equivalent.

Note, that this pattern in this scope corresponds to the "washing machine" example in the Section 1.2.

Chapter 5

# Evaluation and related work

## 5.1 BP-CTL

The result of combining temporal logic and behavior protocols – BP-CTL – is a new temporal logic. The expressive power of BP-CTL is the same as the expressive power of CTL. The proof of this claim is very simple: Each expression written in BP-CTL can be expressed as an equivalent CTL formula as it is formally described in Appendix A, while each CTL formula is trivially (by definition) BP-CTL expression.

The greatest contribution of BP-CTL is the readability of its expressions (in comparison with CTL). While a CTL formula describing sequence of states and their properties usually results in hard readable expression containing many parenthesis and until operators, BP-CTL provides set of simple operators for description of relationships amongst states and specified properties. Thus the expressions in BP-CTL are significantly shorter (for example expressions in Specification Patterns mapping tables for BP-CTL in Appendix B are on average 33 % shorter than formulas in mapping tables for CTL) and much more readable. In addition expressions written in BP-CTL are on principle similar to basic regular expressions and hence easily understandable for most of the programmers.

## 5.2 Specification Patterns inconsistency

Good readability and directness of BP-CTL expressions enables to find an inconsistency in the scope “before R” interpretation across different patterns in property pattern mappings for CTL in the Specification Patterns project [5]. The inconsistency is in fact whether any pattern within the scope before R should trivially hold if no R state occurs. Transcripted CTL formulas for most of the patterns trivially hold in that case. But there are also three transcription pattern formulas which do not hold in such case. The problem was discussed with the repository maintainers. They agreed that this is an inconsistency and said that are planning to make several modifications to patterns and take into account this comments.

There are in general two logical approaches to the issue how to map patterns to CTL. First one is to describe the property which has to hold and the second possibility is negation of the property which must not occur. For example formula `AG(P)` says that there should hold property `P` in every state and the equivalent "negative" form is `!EF(!P)` which means there must not occur non-P-state in the future.

In property pattern mappings to CTL are used both of these approaches according to which of the resulting transcription is more efficient. For mapping of patterns to newly defined BP-CTL we are using just the negative form, because it mostly results in unbranched sequence of states which must not occur. The BP-CTL is designed for describing such state sequences. This uniform approach to the mapping was also one of the important factor for finding out the inconsistency in the interpretation of the scope before R.

Now let us have a look at the problematic CTL transcription in property mappings. At first we show the correct mapping to get better insight to the issue. For example the absence pattern within the scope before R is in the repository as CTL formula `A[(!P | AG(!R)) W R]` which is equivalent to CTL formula in the negative form `!E[!R U (P & !R` **`& EF(R)`**`)]` according to the rule for interpretation of weak until operator `A[x W y] =!E[!y U (!x & !y)]`. As we can see this formula trivially holds when there is no R-state. Now the first of incorrect mappings. Pattern bounded existence within the scope before R is in the repository interpreted as CTL formula

```
!E[!R U (!P & !R & EX(P &
   E[!R U (!P & !R & EX(P &
     E[!R U (!P & !R & EX(P & !R))])))])))].
```

This means that there can be an infinite non R sequence of states which does not satisfy the previous formula. Such a path can be expressed in CTL as

```
E[!R U (!P & !R & EX(P &
  E[!R U (!P & !R & EX(P &
    E[!R U (!P & !R & EX(P & EG(!R)))])))])))].
```

The solution of the problem is to map bounded existence pattern within the scope before R to CTL as

```
!E[!R U (!P & !R & EX(P &
   E[!R U (!P & !R & EX(P &
     E[!R U (!P & !R & EX(P & !R & EF(R)))])))])))].
```

Similar solution can be used also for both of others incorrectly mapped patterns. For the mapping of pattern 1 cause-2 effect precedence chain within the scope before R should be used CTL formula

```
!E[(!P & !R) U (S & !P & !R & EX(E[!R U (T & !R & EF(R))]))]
```

instead of the current

```
!E[(!P & !R) U (S & !P & !R & EX(E[!R U (T & !R)]))].
```

And finally as the mapping for the pattern 2 cause-1 effect precedence chain within the scope before R to CTL should be used formula

```
!E[(!S & !R) U (!P & !R & EF(R))] &
!E[(!P & !R) U (S & !P & !R &
                EX(E[(!T & !R) U (P & !T & !R & EF(R))]))]
```

instead of the current

```
!E[(!S & !R) U (!P & !R)] &
!E[(!P & !R) U (S & !P & !R &
                EX(E[(!T & !R) U (P & !T & !R)]))].
```

Note that the difference between the meaning of interpretations of before R scope used in the patterns repository is the same as the difference between the meaning of between Q and R scope and after Q until R scope.

## 5.3 Related work

### 5.3.1 PSL/Sugar

PSL/Sugar [14] is the specification language used by engineers to specify the functional properties of logic designs. These properties, in turn, serve as input to property-checking tools, which are key to modern-day functional verification. Sugar [13], developed by IBM, is simple, concise, and expressive. The Sugar language was submitted to the Accelera EDA standards organization, who selected Sugar as the basis for an IEEE international standard and renamed it to PSL (short for Property Specification Language). PSL is commonly referred to as PSL/Sugar.

**Sugar**

Sugar [13] was developed in IBM Haifa Research Laboratory in 1994 as Syntactic sugaring of CTL for RuleBase model checker (version 1.0) and further improved by addition of regular expressions – Sugar Extended Regular Expressions (SERE) (1995) and automatic generation of simulation monitors (1997). In 2001 it moves to linear (LTL-based) semantics (version 2.0). In 2002 Sugar was selected by Accellera to serve as a basis for IEEE standard.

**PSL**

Property Specification Language (PSL) [14] is a language for the formal specification of hardware developed by The Accellera Standards Organization. It is used to describe properties that are required to hold in the design under verification. PSL provides a means to write specifications that are both easy to read and mathematically precise. It is intended to be used for functional specification and as input to functional verification tools. Thus, a PSL specification is an executable documentation of a hardware design.

The resulting language combines together LTL, ITL, CTL and expressions – SERE.

**Example**
"Whenever $Req_1$ and $Req_2$ are false, they remain false until *Start* becomes true with $Req_2$ still false."

CTL: $AG(\neg Req_1 \wedge \neg Req_2 \Rightarrow A[\neg Req_1 \wedge \neg Req_2\ U\ (Start \wedge \neg Req_2)])$
PSL: `always(!`$Req_1$` & !`$Req_2$` -> ((!`$Req_1$` & !`$Req_2$`) until! (`*Start*` & !`$Req_2$`)))`

**Comparison**
The PSL/Sugar was developed for hardware verification. Today the project includes a few verification and description languages and many implementation of model checkers. By contrast to this BP-CTL extends the transcription possibilities of well known temporal logic CTL using principles similar to regular expressions and should be therefore easier to learn and understand. Moreover, it is designed as a set of shortcuts for composed formulas and can be rewritten to the pure CTL. This fact is very useful for the praxis because there exist really efficient implementations of model checkers, accepting CTL as a property specification language, which can be used for checking BP-CTL expressions.

## 5.3.2 PROSPER

PROSPER [15] (Proof and Specification Assisted Design Environments) is a collaboration involving the University of Glasgow (UK), the University of Cambridge (UK), the University of Edinburgh (UK), the Universities of Karlsruhe and Tübingen (D), IFAD (DK), and Prover Technology (S). The University of Glasgow acts as the project coordinator. The project is developing an extensible, open proof tool architecture for incorporating formal verification into industrial CAD/CASE tool flows and design methodologies. The tools include novel user-friendly interfaces, and is tested on two major example systems.

As a part of this project, there exists a system which allows the formal verification of digital circuits using specifications expressed in English [16]. The system can turn English sentences into CTL formulas, allowing natural language specifications to be used. A parser for English, returning general-purpose semantic representations, is allied with a convertor from these representations to CTL. The convertor is integrated with the SMV model checker, so that inferential information may be used during semantic interpretation.

**Example**
"After $Sig_1$ is active, $Sig_2$ is active for three cycles."

Alvey semantics:
```
(DECL
 (AFTER (uqe (some (e1) e1))
  (BE (uqe (some (e2) (PRES e2)))
   (ACTIVE (name (the (x1) (and (sg x1) (named x1 sig1))))
    (degree unknown)))
  (and
   (BE #1=(uqe (some (e3) (PRES e3)))
    (ACTIVE (name (the (x2) (and (sg x2) (named x2 sig2))))
     (degree unknown)))
   (FOR #1# (uq ((NN \3) (x3) (and (pl x3) (CYCLE x3))))))
  (timespan unknown)))
```

Resulting CTL formula: $\mathrm{AG}(Sig_1 \Rightarrow \mathrm{AX}(Sig_2 \wedge \mathrm{AX}\ Sig_2 \wedge \mathrm{AX}\ \mathrm{AX}\ Sig_2))$

**Comparison**

This system accepts the most readable formulas at all – natural language sentences. The problem is in the precise interpretation and in the fact, that verbal description is often quite ungainly for formal expressions.

# Chapter 6

# Conclusion and future work

In this thesis we dealt with the combining of temporal logic and behavior protocols with respect to hard readability of temporal logics. The main goals of our work were:

- To propose a new formalism (BP-CTL) for temporal property specification, based on a combination of temporal logic and behavior protocols.
  - To write the formal definition of its syntax and semantics.
  - To demonstrate the usability of newly defined language in comparison to the concrete temporal logic.
- Prototype implementation of a model checker using BP-CTL for specification of temporal properties.

The goals of the thesis were satisfied. We formulated new temporal logic called BP-CTL which enrich the CTL with operators inspired by BP (Chapter 2). We formally defined the BP-CTL in two ways – the set of rules for rewriting a general BP-CTL expression to the equivalent CTL formula (Section 3.2) and the formal semantics (Section 3.3). The usability of BP-CTL was demonstrated on couple of small examples, one complex example modeling a crossroad (Section 4.2) and implementation of Property Specification Patterns (Section 4.3 and Appendix B). During the transcription of specification patterns we even found an inconsistency in existing patterns mappings (Section 5.1). Finally we developed a tool that accepts property specified via BP-CTL and converts them to their CTL equivalents. This tool is proposed as preprocessor for NuSMV symbolic model checker.

The BP-CTL has the same expressive power as CTL (proved in Section 5.1). The possible improvement of the BP-CTL concept is to use CTL* instead of CTL and enrich it with BP operators. The resulting logic – BP-CTL* – would have the same expressive power as CTL*, but it would be much more complicated to work with CTL* and expressions written in BP-CTL* would be probably harder to read than in the case of BP-CTL.

Chapter 7

# References

[1] Edmund M. Clarke, Orna Grumberg, Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 2000. ISBN 0-262-03270-8.

[2] Edmund M. Clarke, E. Allen Emerson, A. Prasad Sistla. *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*. ACM Transactions on Programming Languages and Systems, Vol. 8, No. 2, April 1986.

[3] František Plášil, Stanislav Višňovský. *Behavior Protocols for Software Components*, IEEE Transactions on Software Engineering, Vol. 28, No. 11, November 2002.

[4] Jiří Adámek, František Plášil. *Component Composition Errors and Update Atomicity: Static Analysis*. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 17, No. 5, September 2005.

[5] Hamid Alavi, George S. Avrunin, James C. Corbett, Laura Dillon, Matthew B. Dwyer, Corina Pasareanu. *Specification Patterns project*. An online repository for information about property specification for finite-state verification. Project home page http://patterns.projects.cis.ksu.edu/.

[6] Matthew B. Dwyer, George S. Avrunin, James C. Corbett. *Property Specification Patterns for Finite-state Verification*. Proceedings of the Second Workshop on Formal Methods in Software Practice, pages 7–15, March 1998.

[7] Matthew B. Dwyer, George S. Avrunin, James C. Corbett. *Patterns in Property Specifications for Finite-state Verification*. Proceedings of the 21st International Conference on Software Engeneering, pages 411–420, May 1999.

[8] *NuSMV symbolic model checker*. Project home page http://nusmv.irst.itc.it/.

[9] Roberto Cavada, Alessandro Cimatti, Emanuele Olivetti, Gavin Keighren, Marco Pistore, Marco Roveri. *NuSMV 2.2 User Manual*. CMU and ITC-IRST, 2004.

[10] *Spin*. Project home page http://www.spinroot.com/.

[11] *Bandera*. Project home page http://bandera.projects.cis.ksu.edu/.

[12] *Java PathFinder*. Project home page http://javapathfinder.sourceforge.net/.

[13] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, Yoav Rodeh. *The Temporal Logic Sugar*. Proceedings of 13th International Conference on Computer-Aided Verification (CAV), LNCS 2102, pages 363–367. Springer-Verlag, July 2001.

[14] Accellera. *Property Specification Language Reference Manual*. June 2004.

[15] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, Mike Gordon, and Tom Melham. *The Prosper Toolkit*. Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785, pages 78–92. Springer-Verlag, April 2000.

[16] Alexander Holt, Ewan Klein and Claire Grover. *Natural language for hardware verification: semantic interpretation and model checking*. Proceedings of ICoS-1 workshop: Inference in Computational Semantics, Institute for Logic, Language and Computation (ILLC), Amsterdam, August 1999, pages 133-137. ILLC, University of Amsterdam, September 1999.

## Appendix A

# Used formalisms in details

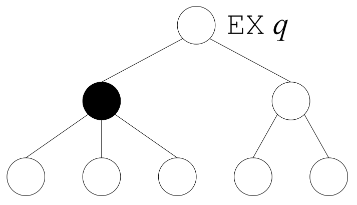
## A.1 Computation Tree Logic (CTL)

The CTL is a representative of temporal logic. It is based on propositional logic of branching time, that is, a logic where time may split into more than one possible future using a discrete model of time.

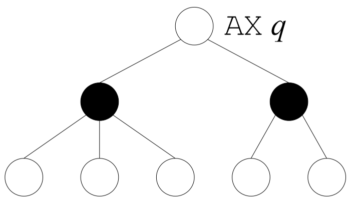
The formulas in CTL are composed of atomic propositions and logical and temporal operators. Atomic propositions are its building blocks for making statements about the states of system. The logical operators are: *negation* ( `!` ), *and* ( `&` ) and *or* ( `|` ). All others common logical operators can be expressed as combination of these three.

Temporal operators consist of forward-time operators globally (`G`), in the future (`F`), next time (`X`) and until (`U`) preceded by a path quantifier which decides where should hold the given property. There are two possibilities `A` for all computation paths and `E` which means some computation path (or there exist a path).

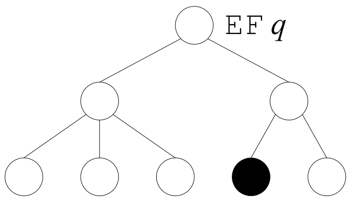
We show the meaning of the temporal operators on simple diagrams and complete them by verbal description. The computation tree represents an unfolded state graph where the nodes are the possible states that the system may reach. The grayed nodes are those states in which property *p* holds and black nodes these where property *q* holds. Thus it is possible to express properties that hold in the root node (initial state) using CTL temporal operators.



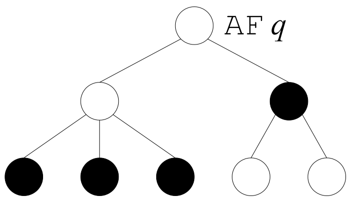
The formula `EX` *q* holds in the initial state if there exists at least one next state in which property *q* is satisfied.



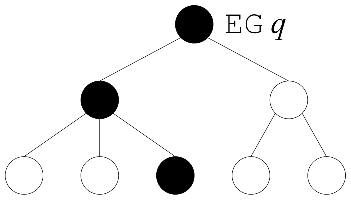
The formula AX *q* holds in the initial state if in every next state of the initial state property *q* is satisfied.



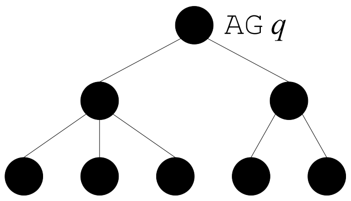
The formula EF *q* holds in the initial state if there exists a path, starting from the initial state, such that there exists at least one state on the path (including the initial state) in which property *q* is satisfied.



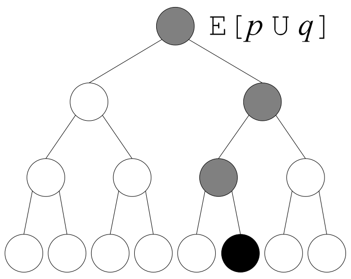
The formula AF *q* holds in the initial state if for every possible path, starting from the initial state, there exists at least one state (including the initial state) in which property *q* is satisfied, that is, *q* eventually happen.



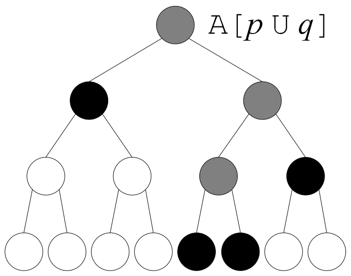
The formula EG *q* holds in the initial state if there exists a path, starting in the initial state, such that in every state on the path property *q* is satisfied.



The formula AG *q* holds in the initial state if property *q* is satisfied in the initial state and every following state. This means *q* holds in the whole subtree.



The formula `E [`$p$ `U` $q$`]` holds in the initial state if there exists a path, starting in the initial state, such that in every state on the path (including the initial state) property *p* is satisfied until comes the state where property *q* is satisfied. This means *p* holds until *q* becomes true.



The formula `A [`$p$ `U` $q$`]` holds in the initial state if for every possible path, starting in the initial state, in every state property p is satisfied until comes the state where property q is satisfied. In other words on every path, stating in the initial state, *p* holds until *q* becomes true.

### A.1.1 Formal semantics of CTL

A **Kripke structure** is a 5-tuple $(S, I, P, L, T)$ with $S$ a finite set of states, $I \subseteq S$ a set of initial states, $P$ a finite set of atomic propositions, $L: S \to 2^P$ a labeling function, and $T \subseteq S \times S$ a transition relation.

Let $M$ be a Kripke structure, $s, t \in S$, $p \in P$ and $\varphi$ be a CTL formula. $M, s \models \varphi$ denotes that formula $\varphi$ holds in the state $s$ of Kripke structure $M$. Transition $s \to t$ means that $(s, t) \in T$. Now we can inductively define satisfaction relation:

| | | |
|---|---|---|
| $M, s \models p$ | $\Leftrightarrow$ | $p \in L(s)$ |
| $M, s \models \neg\varphi$ | $\Leftrightarrow$ | not $M, s \models \varphi$ |
| $M, s \models \varphi_1 \vee \varphi_2$ | $\Leftrightarrow$ | $M, s \models \varphi_1$ or $M, s \models \varphi_2$ |
| $M, s \models \varphi_1 \wedge \varphi_2$ | $\Leftrightarrow$ | $M, s \models \varphi_1$ and $M, s \models \varphi_2$ |
| $M, s \models \mathtt{EX}\varphi$ | $\Leftrightarrow$ | there is a state $t$ and a transition $s \to t$ in $M$ such that $M, t \models \varphi$ |
| $M, s \models \mathtt{AX}\varphi$ | $\Leftrightarrow$ | for every state $t$ in $M$ such that $s \to t$, $M, t \models \varphi$ holds |
| $M, s \models \mathtt{EF}\varphi$ | $\Leftrightarrow$ | there exists a state $t$ and a path from $s$ to $t$ in $M$ such that $M, t \models \varphi$ |
| $M, s \models \mathtt{AF}\varphi$ | $\Leftrightarrow$ | on every infinite path in $M$ beginning in $s$ there is a state $t$ such that $M, t \models \varphi$ |
| $M, s \models \mathtt{EG}\varphi$ | $\Leftrightarrow$ | there exists an infinite path $\pi = \pi_0 \to \pi_1 \to \pi_2 \to \ldots$ in $M$ such that $\pi_0 = s$ and $\forall i \geq 0$: $M, \pi_i \models \varphi$ |
| $M, s \models \mathtt{AG}\varphi$ | $\Leftrightarrow$ | for every infinite path $\pi = \pi_0 \to \pi_1 \to \pi_2 \to \ldots$ in $M$ such that $\pi_0 = s$: $\forall i \geq 0$ $M, \pi_i \models \varphi$ |

|  |  |  |
|---|---|---|
| $M, s \models \mathtt{E}[\varphi_1 \ \mathtt{U}\ \varphi_2]$ | $\Leftrightarrow$ | there exists an infinite path $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ in $M$ such that $\pi_0 = s$ and $\exists i \geq 0$ such that $M, \pi_i \models \varphi_2$ and $\forall\ 0 \leq j < i$: $M, \pi_j \models \varphi_1$ |
| $\mathrm{M}, s \models \mathtt{A}[\varphi_1 \ \mathtt{U}\ \varphi_2]$ | $\Leftrightarrow$ | for all infinite path $\pi = \pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \ldots$ in $M$ such that $\pi_0 = s$: $\exists i \geq 0$ such that $M, \pi_i \models \varphi_2$ and $\forall j$, $0 \leq j < i$: $M, \pi_j \models \varphi_1$ |

## A.2 Behavior protocols

Behavior protocol (or protocol for short) *Prot* is in principle an expression which describes a component's behavior – all possible actions that the component can perform (call and receive methods). Protocol generates a set of traces over an alphabet S – the language L(Prot). By convention S holds for $S \subseteq \{!, ?, \tau\} \times \mathrm{GNS} \times \mathrm{LNS}_{cn} \times \{\uparrow, \downarrow\}$, where GNS stands for the global name space (names of connections of the component) and $\mathrm{LNS}_{cn}$ stands for the local name space (names of events of the connection cn). Therefore every action token (an event) is syntactically written as <event prefix> <connection_name>.<local_event_name><event suffix>. The event prefix (one of the symbols *!*, *?*, resp. τ), expresses whether an event is emitted, received, resp. internal. The event suffix denotes a request (↑) and a response (↓) part of a remote method call. The traces of a component are words over its alphabet S (from S*).

The simplest behavior protocol is an event token or the NULL symbol (empty trace). A behavior protocol is constructed in a way similar to a regular expression and can use the operators and abbreviations listed bellow. The basic operators are those of regular expressions. The enhanced operators provide a notation for describing concurrency resp. communication hiding and represent well-known operations of shuffle resp. restriction of a language. Finally the composed operators: the semantics of the adjustment operator |T| is inspired by the generalized parallel operator defined in CSP (Communicating Sequential Processes), while the semantics of the composition operator $\sqcap_X$ by the parallel composition in CCS (Calculus of Communicating Systems).

**Operators** (A, B denotes a protocol; m and e an event name)

*Basic operators* (defined in classical regular expressions)

|  |  |
|---|---|
| A; B | *sequencing*; the set of traces formed by concatenation of a trace generated by A and a trace generated by B, |
| A + B | *alternative*; the set of traces which are generated either by A or by B, |
| A* | *repetition*; equivalent to NULL + A + (A; A) + (A; A; A) + ..., where A is repeated any finite number of times. |

*Enhanced operators*

|  |  |
|---|---|
| A \| B | *and-parallel*; an arbitrary interleaving of event tokens of traces generated by A and B, |
| A \|\| B | *or-parallel*; stands for A + B + (A \| B), |
| A / G | *restriction*; the event tokens not in a set G are omitted from the traces of L(A). |

*Composed operators*

|  |  |
|---|---|
| $A \sqcap_X B$ | *composition*; the set of traces – each formed as an arbitrary interleaving of event tokens from a pair of traces (α, β), (where α, resp. β, is generated by A, resp. B), such that, for every event x from X, if x is prefixed by ? in α and by ! in β (or vice |

| | |
|---|---|
| | versa), any appearance of ?x, !x resp. !x, ?x as a result of the interleaving is merged into τx in the resulting trace (the pair of events becomes an internal event), |
| A \|T\| B | *adjustment*; the set of traces – each formed as an arbitrary interleaving of event tokens from a pair of traces (α, β), (where α, resp. β, is generated by A, resp. B), with the exception of event tokens from T which have to appear in α and β in the same order (representing "synchronization points"). If the interleaving produces ...x, x... for an x from T (a set of event tokens), then x, x is merged into ...x... in the resulting trace (the pair becomes a single event), |
| A $\nabla_X$ B | *consent*; the set containing all the traces from A $\sqcap_X$ B and all the erroneous traces induced by the composition of A and B. |

**Abbreviations**

| | |
|---|---|
| ?m{α} | *nested incoming call*; stands for ?m↑; α; !m↓ |
| ?m | *simple incoming call*; stands for ?m↑; !m↓ |
| !m | *simple outgoing call*; stands for !m↑; ?m↓ |

**Precedence of operators**

1. (highest) repetition (*), restriction (/)
2. sequencing (;)
3. and-parallel (|), or-parallel (||)
4. alternative (+)
5. (lowest) composition ($\sqcap_X$), adjustment (|T|), consent ($\nabla_X$)

**Example protocols**

To demonstrate behavior protocols as a tool for language generation, let us consider the protocol *?a; (!p + !q); !b*. It contains event tokens *?a*, *!b*, *!p*, *!q* and the operators *;* and +. (In the examples here, we omit event suffixes ↑ and ↓ and connection names for simplicity.) The protocol generates traces, which start with *?a*, followed by *!p* or *!q* and finish with *!b* and therefore the generated language is {<*?a, !q, !b*>, <*?a, !p, !b*>}.

Consider the protocol *?a; (!p + !q); !b* || *?x*. The event *x* occurs in parallel with the behavior on the left hand side of ||. The generated language includes for instance the traces <*?a, !p, !b*> and <*?a, ?x, !q, !b*> (notice that *x* does not have to be necessarily present). Now, we enhance the protocol by adding the composition operator, e.g., *?a; (!p + !q); !b* || *?x* $\sqcap_S$ *(?q; ?r)*, where S contains *p* and *q*. In the language generated, there cannot be a trace including a *p* event, since the right operand of $\sqcap_S$ requires *q* (and no *p*) to appear. However, for traces of the left operand of composition where *q* is present, in the resulting traces is *q* expressed via an internal event as in <*?a, ?x, τq, !b, ?r*> or <*?a, ?x, τq, ?r, !b*> (*!b*, *?r* can be arbitrary interleaved).

For more information about BP we refer the reader to [3] and [4].

Appendix B

# Property Specification Patterns

In this section we demonstrate the usability of BP-CTL. We compare formulas written in pure CTL (the first row) and their equivalent formulas written in BP-CTL (the second row). We use the patterns from the repository of Specification Patterns project.

Note that many of the mappings use the weak until operator (`W`) which is related to the strong until operator (`U`) by the following equivalences:

```
A[x W y] = !E[!y U (!x & !y)]
E[x U y] = !A[!y W (!x & !y)].
```

## B.1 Absence Property Pattern

To describe a portion of a system's execution that is free of certain events or states. Also called as **Never**. The most common example is mutual exclusion. In a state-based model, the scope would be global and `P` would be a state formula that is true if more than one process is in its critical section. For an event-based model, the scope would be a segment of the execution in which some process is in its critical section (i.e., between an enter section event and a leave section event), and `P` would be the event that some other process enters its critical section.

P is false:

| | |
|---|---|
| Globally | `AG(!P)` |
| | `!EF(P)` |
| Before R | `A[(!P \| AG(!R)) W R]` |
| | `!((TRUE, P &= !R), R)` or `!(!R*; P & !R, R)` |
| After Q | `AG(Q -> AG(!P))` |
| | `!EF(Q, P)` |

| | |
|---|---|
| Between Q and R | AG(Q & !R -> A[(!P \| AG(!R)) W R]) |
| | !EF((Q, P &= !R), R) |
| After Q until R | AG(Q & !R -> A[!P W R]) |
| | !EF(Q, P &= !R) |

## B.2 Existence Property Pattern

To describe a portion of a system's execution that contains an instance of certain events or states. Also called as **Eventually**. The classic example of existence is specifying termination, e.g., on all executions we do eventually reach a terminal state.

This pattern is the dual of the Absence pattern. In fact, in many specification formalism negation and explicit queries for existence is used to formulate an instance of the Absence pattern.

P becomes true:

| | |
|---|---|
| Globally | AF(P) |
| | !(!P@) |
| Before R | A[!R W (P & !R)] |
| | !(!P*; R) or !((!P &= !R)*; R) |
| After Q | A[!Q W (Q & AF(P))] |
| | !(!Q*; Q && !P@) |
| Between Q and R | AG(Q & !R -> A[!R W (P & !R)]) |
| | !(Q & !R && !P*; R) |
| After Q until R | AG(Q & !R -> A[!R U (P & !R)]) |
| | !(Q & !R && !P@; R) |

## B.3 Bounded Existence Property Pattern

To describe a portion of a system's execution that contains at most a specified number of instances of designated state transition or event. Bounded overtaking properties can be naturally expressed using instances of this pattern. For example, if we wish to say that process 1 can enter its critical region at most twice while process 2 is waiting to enter its region we would use a between scope (delimited by process 2 entering and exiting its waiting region) with 2-bounded existence for process 1 entering its critical region.

In these mapping we illustrate one instance of the bounded existence pattern, where the bound is at most 2 designated states. Other bounds can be specified by variations on this mapping.

Transitions to P-states occur at most 2 times:

| | |
|---|---|
| Globally | `!EF(!P & EX(P & EF(!P & EX(P & EF(!P & EX(P))))))` |
| | `!EF(!P; P, !P; P, !P; P)` |
| Before R | `!E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & !R & EF(R)))])))])))]` |
| | `!((TRUE, !P; P, !P; P, !P; P &= !R), R)` |
| After Q | `!E[!Q U (Q & EF(!P & EX(P & EF(!P & EX(P & EF(!P & EX(P)))))))]` |
| | `!EF(Q, !P; P, !P; P, !P; P)` |
| Between Q and R | `AG(Q -> !E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & !R & EF(R)))])))])))])` |
| | `!EF((Q, !P; P, !P; P, !P; P &= !R), R)` |
| After Q until R | `AG(Q -> !E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & E[!R U (!P & !R & EX(P & !R))])))])))])` |
| | `!EF(Q, !P; P, !P; P, !P; P &= !R)` |

## B.4 Universality Property Pattern

To describe a portion of system's execution which contains only states that have a desired property. Also called as **Henceforth** and **Always**. This pattern can be applied in most situations where the absence pattern can be applied. This is especially true for state-based formalism, e.g., where mutual exclusion can be formulated as absence or universality with a between scope.

P is true:

| | |
|---|---|
| Globally | `AG(P)` |
| | `!EF(!P)` |

| | |
|---|---|
| Before R | `A[(P \| AG(!R)) W R]` |
| | `!((TRUE, !P &= !R), R)` or `!(!R*; !P & !R, R)` |
| After Q | `AG(Q -> AG(P))` |
| | `!EF(Q, !P)` |
| Between Q and R | `AG(Q & !R -> A[(P \| AG(!R)) W R])` |
| | `!EF((Q, !P &= !R), R)` |
| After Q until R | `AG(Q & !R -> A[P W R])` |
| | `!EF(Q, !P &= !R)` |

## B.5 Precedence Property Pattern

To describe relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first. Precedence properties occur quite commonly in specifications of concurrent systems. One common example is in describing a requirement that a resource is only granted in response to a request.

S precedes P:

| | |
|---|---|
| Globally | `A[!P W S]` |
| | `!(!S*; P & !S)` or `A(!P@; S)` |
| Before R | `A[(!P \| AG(!R)) W (S \| R)]` |
| | `!((!S*; P & !S &= !R), R)` |
| After Q | `A[!Q W (Q & A[!P W S])]` |
| | `!(!Q*; Q && !S*; P & !S)` |
| Between Q and R | `AG(Q & !R -> A[(!P \| AG(!R)) W (S \| R)])` |
| | `!EF((Q && !S*; P & !S &= !R), R)` |
| After Q until R | `AG(Q & !R -> A[!P W (S \| R)])` |
| | `!EF(Q && !S*; P & !S &= !R)` |

## B.6 Response Property Pattern

To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also called as **Follows** and **Leads-to**. Response properties occur quite commonly in specifications of concurrent systems. Perhaps the most common example is in describing a requirement that a resource must be granted after it is requested.

S responds to P:

| | |
|---|---|
| Globally | `AG(P -> AF(S))` |
| | `!EF(P && !S@)` |
| Before R | `A[((P -> A[!R U (S & !R)]) \| AG(!R)) W R]` |
| | `!((TRUE, P &= !R) && !S*; R)` |
| After Q | `A[!Q W (Q & AG(P -> AF(S))]` |
| | `!EF(Q, P && !S@)` |
| Between Q and R | `AG(Q & !R -> A[((P -> A[!R U (S & !R)]) \| AG(!R)) W R])` |
| | `!EF((Q, P &= !R) && !S*; R)` |
| After Q until R | `AG(Q & !R -> A[(P -> A[!R U (S & !R)]) W R])` |
| | `!EF((Q, P &= !R) && !S@; R)` |

## B.7 Precedence Chain

This is a scalable pattern. We describe the 1 cause-2 effect version here. To describe a relationship between an event/state P and a sequence of events/states (S, T) in which the occurrence of S followed by T within the scope must be preceded by an occurrence of the sequence P within the same scope. In state-based formalisms, the beginning of the enabled sequence (S, T) may be satisfied by the same state as the enabling condition (i.e., P and S may be true in the same state).

An example of this pattern, assuming reliable communication between client and server, is that: “If a client makes a request and there is no response, then the server must have crashed”. This would be expressed by parameterizing the constrained variant of the 1-2 precedence chain pattern.

This illustrates the 1 cause-2 effect precedence chain.

P precedes S, T:

| | |
|---|---|
| Globally | !E[!P U (S & !P & EX(EF(T)))] |
| | !(!P*; S && !P;+ T) |
| Before R | !E[(!P & !R) U (S & !P & !R & EX(E[!R U (T & !R & EF(R))]))] |
| | !((!P*; S && !P;+ T &= !R), R) |
| After Q | !E[!Q U (Q & E[!P U (S & !P & EX(EF(T)))])] |
| | !(!Q*; Q && !P*; S && !P;+ T) |
| Between Q and R | AG(Q -> !E[(!P & !R) U (S & !P & !R & EX(E[!R U (T & !R & EF(R))]))]) |
| | !EF((Q && !P*; S && !P;+ T &= !R), R) |
| After Q until R | AG(Q -> !E[(!P & !R) U (S & !P & !R & EX(E[!R U (T & !R)]))]) |
| | !EF(Q && !P*; S && !P;+ T &= !R) |

This illustrates the 2 cause-1 effect precedence chain.

S, T precedes P:

| | |
|---|---|
| Globally | !E[!S U P] & !E[!P U (S & !P & EX(E[!T U (P & !T)]))] |
| | !(!S*; P) & !(!P*; S & !P; !T*; P & !T) |
| Before R | !E[(!S & !R) U (P & !R & EF(R))] & !E[(!P & !R) U (S & !P & !R & EX(E[(!T & !R) U (P & !T & !R & EF(R))]))] |
| | !((!S*; P &= !R), R) & !((!P*; S & !P; !T*; P & !T &= !R), R) |
| After Q | !E[!Q U (Q & (E[!S U P] \| E[!P U (S & !P & EX(E[!T U (P & !T)]))]))] |
| | !(!Q*; Q && ((!S*; P) \| (!P*; S & !P; !T*; P & !T))) |
| Between Q and R | AG(Q -> !E[(!S & !R) U (P & !R & EF(R))] & !E[(!P & !R) U (S & !P & !R & EX(E[(!T & !R) U (P & !T & !R & EF(R))]))]) |
| | !EF(Q; (((!S*; P &= !R), R) \| ((!P*; S & !P; !T*; P & !T &= !R), R))) |

| | |
|---|---|
| After Q until R | AG(Q -> !E[(!S & !R) U (P & !R)] &<br>        !E[(!P & !R) U (S & !P & !R &<br>         EX(E[(!T & !R) U (P & !T & !R)]))]) |
| | !EF(Q; ((!S*; P &= !R) \|<br>        (!P*; S & !P; !T*; P & !T &= !R))) |

## B.8 Response Chain

This is a scalable pattern. We describe the intent of the 1 stimulus-2 response version here. To describe a relationship between a stimulus event (P) and a sequence of two response events (S, T) in which the occurrence of the stimulus event must be followed by an occurrence of the sequence of response events within the scope. In state-based formalisms, the states satisfying the response must be distinct (i.e., S and T must be true in different states to count as a response), but the response may be satisfied by the same state as the stimulus (i.e., P and S may be true in the same state).

If a resource allocator grants a process access to a resource (GrantRes), the process start using the resource (BeginRes) and finish using the resource (EndRes).

This illustrates the 1 stimulus-2 response chain.

S, T responds to P:

| | |
|---|---|
| Globally | AG(P -> AF(S & AX(AF(T)))) |
| | !EF(P && !S@) \|<br>!EF(P, S; !T@) |
| Before R | !E[!R U (P & !R & (E[!S U R] \|<br>                 E[!R U (S & !R & EX(E[!T U R]))]))] |
| | !(!R*; P && ((!R && !S*; R) \|<br>             (!R*; S & !R; !T*; R))) |
| After Q | !E[!Q U (Q & EF(P & (EG(!S) \|<br>                     EF(S & EX(EG(!T))))))] |
| | !(!Q*; Q, P && !S@) \|<br>!(!Q*; Q, P, S; !T@) |
| Between Q and R | AG(Q -><br>      !E[!R U (P & !R &<br>               (E[!S U R] \|<br>                E[!R U (S & !R & EX(E[!T U R]))]))]) |
| | !EF(Q && !R*; P && ((!R && !S*; R) \|<br>                    (!R*; S & !R; T*; R))) |

| | |
|---|---|
| After Q until R | `AG(Q -> !E[!R U (P & !R & (E[!S U R] \| EG(!S & !R) \| E[!R U (S & !R & EX(E[!T U R] \| EG(!T & !R)))])))])` |
| | `!EF(Q && !R*; P && ((!R && !S@; R) \| (!R*; S & !R; T@; R)))` |

This illustrates the 2 stimulus-1 response chain.

P responds to S, T:

| | |
|---|---|
| Globally | `!EF(S & EX(EF(T & EG(!P))))` |
| | `!EF(S;+ T && !P@)` |
| Before R | `!E[!R U (S & !R & EX(E[!R U (T & !R & E[!P U R])]))]` |
| | `!((TRUE, S;+ T & !P &= !R); !P*; R)` |
| After Q | `!E[!Q U (Q & EF(S & EX(EF(T & EG(!P)))))]` |
| | `!(!Q*; Q, S;+ T && !P@)` |
| Between Q and R | `AG(Q -> !E[!R U (S & !R & EX(E[!R U (T & !R & E[!P U R])]))])` |
| | `!EF((Q, S;+ T & !P &= !R); !P*; R)` |
| After Q until R | `AG(Q -> !E[!R U (S & !R & EX(E[!R U (T & !R & (E[!P U R] \| EG(!P & !R)))]))])` |
| | `!EF((Q, S;+ T & !P &= !R); !P@; R)` |

## B.9 Constrained Chain Property Patterns

To describe a variant of response and precedence chain patterns that restrict user specified events from occurring between pairs of states/events in the chain sequences. Consecutive pairs of states/events in chain sequences are refered to as links. This pattern allows specification of the absence of states/events from individual links.

Introducing constraints into mappings for chain links depend on the formalism and scope of the mapping. In CTL with global and after scopes, the future operator (`AF(X)`) can be expanded to its until equivalent (`A[TRUE U X]`), then the `TRUE` can be replaced by the negation of the constraining formula. With before, between and after–until scopes, until is used for expressing each link and this can be selectively constrained by conjoining the negation of the constraining formula on the left of the until operator.

This is a 1-2 response chain constrained by a single proposition.

S, T without Z responds to P:

| | |
|---|---|
| Globally | AG(P -> AF(S & !Z & AX(A[!Z U T]))) |
| | !EF((P && !S@) \| (P, S && ((!T*; Z) \| EX(!T@)))) |
| Before R | !E[!R U (P & !R & (E[!S U R] \| E[!R U (S & !R & (E[!T U Z] \| EX(E[!T U R])))])))] |
| | !(!R*; P & !R && ((!S*; R) \| (!R*; S & !R && ((!T*; Z) \| EX(!T*; R))))) |
| After Q | !E[!Q U (Q & EF(P & (EG(!S) \| EF(S & (E[!T U Z] \| EX(EG(!T))))))))] |
| | !(!Q*; Q, ((P && !S@) \| (P, S && ((!T*; Z) \| EX(!T@))))) |
| Between Q and R | AG(Q -> !E[!R U (P & !R & (E[!S U R] \| E[!R U (S & !R & (E[!T U Z] \| EX(E[!T U R])))])))]) |
| | !EF(Q && !R*; P & !R && ((!S*; R) \| (!R*; S & !R && ((!T*; Z) \| EX(!T*; R))))) |
| After Q until R | AG(Q -> !E[!R U (P & !R & (E[!S U R] \| EG(!S & !R) \| E[!R U (S & !R & (E[!T U Z] \| EX(E[!T U R] \| EG(!T & !R))))])))]) |
| | !EF(Q && !R*; P & !R && ((!S*; !R) \| (!S & !R)@ \| (!R*; S & !R && ((!T*; Z) \| EX(!T*; R) \| (!T & !R)@)))) |

# Appendix C

# User manual

For the demonstration of practical usage of BP-CTL was developed a NuSMV preprocessor called bpctl. In the following text we at first show its usage on implementation of the example described in Section 4.2 and then we describe all options in reference manual.

## C.1 Example

On the CD enclosed to this thesis is an implementation of the example from Section 4.2. Now we show how to run the bpctl and NuSMV for the example.

1) Run the command line or shell (depending on the used platform)
2) Navigate to the base directory of the enclosed CD
3) Run the command

```
bin/<platform>/bpctl example/semaphores.smv | bin/<platform>/NuSMV
```

where <platform> denotes `i386` for GNU Linux and `win` for Microsoft Windows.

We get the NuSMV results. All the specified properties are true.

## C.2 Reference manual

### C.2.1 The bpctl usage

The bpctl reads the stream from the standard input (if no other source is given), transforms the data and prints the results to standard output (if no other target is given). The usage schema can be expressed as:

```
bpctl <options> [<input file>] [<output file>]
```

where possible options are:

| | |
|---|---|
| -v <file> | which means print debug information to given <file> |
| -h | which prints usage information to standard output. |

The optional parameters <input file> and <output file> specify another source and target for the preprocessor than standard input and output.

The NuSMV model checker can read its input from standard input hence it is very useful to redirect the bpctl output stream to NuSMV input via pipe.

## C.2.2 The bpctl input language

The bpctl input language is the NuSMV input language [9] extended with new keywords for using the BP-CTL for specifying properties. Thus it is necessary to know at least the structure of NuSMV input language described in [9].

**BP-CTL specifications**

A BP-CTL specification is given as a formula in the temporal logic BP-CTL, introduced by the keyword '`BPSPEC`'. The syntax of this declaration is similar to the NuSMV original `SPEC` and `LTLSPEC` declarations.

The syntax of BP-CTL was formally described in the Section 3.3. It can be expressed as:

```
bpctl_expr ::
  p_state_expr
```

where path-like state expression syntax is:

```
p_state_expr ::
  e_state_expr
  | path_expr
  | e_state_expr "&&" p_state_expr
  | e_state_expr ";" p_state_expr
  | e_state_expr "," p_state_expr
  | e_state_expr ";+" p_state_expr
  | e_state_expr "*"
  | e_state_expr "*" ";" p_state_expr
  | e_state_expr "+"
  | e_state_expr "+" ";" p_state_expr
  | e_state_expr "@"
  | e_state_expr "@" ";" p_state_expr
  | "A" "(" path_expr ")"
  | "A" "(" path_expr ")" "&&" p_state_expr
  | "A" "(" path_expr ")" ";" p_state_expr
  | "A" "(" path_expr ")" "," p_state_expr
  | "A" "(" path_expr ")" ";+" p_state_expr
  | "A" "(" path_expr ")" "*" ";" p_state_expr
  | "A" "(" path_expr ")" "+" ";" p_state_expr
  | "A" "(" path_expr ")" "@" ";" p_state_expr
  | path_expr "&=" path_expr
  | "(" path_expr "&=" path_expr ")" "&&" p_state_expr
  | "(" path_expr "&=" path_expr ")" ";" p_state_expr
  | "(" path_expr "&=" path_expr ")" "," p_state_expr
  | "(" path_expr "&=" path_expr ")" ";+" p_state_expr
  | "(" path_expr "&=" path_expr ")" "*" ";" p_state_expr
  | "(" path_expr "&=" path_expr ")" "+" ";" p_state_expr
  | "(" path_expr "&=" path_expr ")" "@" ";" p_state_expr
```

Extended state expression syntax is:

```
e_state_expr ::
  state_expr
  | "!" p_state_expr
  | p_state_expr "&" p_state_expr
  | p_state_expr "|" p_state_expr
  | "EX" p_state_expr
  | "EF" p_state_expr
  | "EG" p_state_expr
  | "E" "[" p_state_expr "U" p_state_expr "]"
  | "AX" p_state_expr
  | "AF" p_state_expr
  | "AG" p_state_expr
  | "A" "[" p_state_expr "U" p_state_expr "]"
```

Path expression syntax is:

```
path_expr ::
  state_expr
  | state_expr "&&" path_expr
  | state_expr ";" path_expr
  | state_expr "," path_expr
  | state_expr ";+" path_expr
  | state_expr "*"
  | state_expr "*" ";" path_expr
  | state_expr "+"
  | state_expr "+" ";" path_expr
  | state_expr "@"
  | state_expr "@" ";" path_expr
```

State expression syntax is:

```
state_expr ::
  simple_expr
  | "!" path_expr
  | path_expr "&" path_expr
  | path_expr "|" path_expr
  | "EX" path_expr
```

```
   | "EF" path_expr
   | "EG" path_expr
   | "E" "[" path_expr "U" path_expr "]"
   | "AX" path_expr
   | "AF" path_expr
   | "AG" path_expr
   | "A" "[" path_expr "U" path_expr "]"
```

The BP-CTL specification is converted by the bpctl preprocessor to the CTL specification which can be then checked by the NuSMV symbolic model checker.