Journal of Universal Computer Science, vol. 10, no. 11 (2004), 1498-1518
submitted: 10/5/04, accepted: 20/8/04, appeared: 28/11/04 © J.UCS

FBT: A Tool for Applying Interval Logic Specifications to On-the-fly Model Checking

Miguel J. Hornos
(Dpto. de Lenguajes y Sistemas Informáticos, University of Granada, Spain
mhornos@ugr.es)

Abstract: This paper presents the FBT (FIL to Büchi automaton Translator) tool which automatically translates any formula from FIL (Future Interval Logic) into its semantically equivalent Büchi automaton. There are two advantages of using this logic for specifying and verifying system properties instead of other more traditional and extended temporal logics, such as LTL (Linear Temporal Logic): firstly, it allows a succinct construction of specific temporal contexts, where certain properties must be evaluated, thanks to its key element, the interval; and secondly, it also permits a natural, intuitive, graphical representation. The underlying algorithm of the tool is based on the *tableau method* and is specially intended for application in *on-the-fly model checking*. In addition to a description of the design and implementation structure of FBT, we also present some experimental results obtained by our tool, and compare these results with the ones produced by another tool of similar characteristics (i.e. based on an *on-the-fly tableau* algorithm), but for LTL.

Key Words: Specification, Temporal Logic, Interval Logic, FIL (Future Interval Logic), GIL (Graphical Interval Logic), Automatic Verification, On-the-fly Model Checking, Tableau Method, Büchi Automata.

Categories: F.3.1, D.2.4, D.2.1

1 Introduction

The FBT (FIL to Büchi automaton Translator) tool presented in this paper automatically translates a formula from FIL (Future Interval Logic) [Ramakrishna et al. 1992] into its semantically equivalent Büchi automaton [Büchi 1962]. The underlying algorithm [Hornos, Capel 2002] is based on the *tableau method* [Wolper 1985], and is specially intended to be applied to *on-the-fly model checking* [Gerth et al. 1995], in such a way that the property automaton can be generated simultaneously with, and guided by, the construction of the system model. It is therefore possible to detect that a property has been violated by constructing only a part of both state spaces. Until very recently, the integration of both approaches (*tableau* and *on-the-fly*) for an interval logic was considered unfeasible for this type of logic. For this reason, we consider FBT to be not only an innovation but also an important achievement.

Traditional temporal logics, such as LTL (Linear Temporal Logic) [Manna, Pnueli 1992] and CTL (Computation Tree Logic) [Ben-Ari et al. 1983], allow reasoning about the relative ordering of events in a system. However, we must formulate quite intricate expressions with them in order to describe a temporal context in which certain requirements or properties must only be satisfied within it. This, together with the fact that these logics do not have an intuitive representation, such as a semantically equivalent graphical notation, has lead many system designers to believe that they are difficult to use as formal description languages for the requirement specification of systems, and that many of the specifications created with them are formulas which are too complicated to understand. All of this has hindered a more extended use of the mentioned logics as specification languages during the analysis phase of the development cycle of industrial applications.

Unlike these logics, the formal specification language FIL, which our tool uses, allows the succinct construction of bounded temporal contexts, thanks to its key element, the *interval*, which defines such contexts clearly and concisely. In addition to the textual representation of its formulas, this logic also has a natural, intuitive, graphical representation called GIL (Graphical Interval Logic) [Dillon et al. 1994], and both the textual and graphical representation are semantically equivalent.

This paper is organized as follows. [Section 2] introduces the specification formalism used, i.e. the graphical syntax of GIL and the textual syntax of FIL, and also the semantics associated with their constructions. Moreover, it shows some examples of how certain temporal properties of a real-world system are specified in both representations. [Section 3] describes the main characteristics of the design and implementation structure of our tool (FBT) in addition to its input and output interfaces, while [Section 4] presents some of the experimental results obtained with it, and compares these with the results obtained with another similar tool (i.e. based on an *on-the-fly tableau* algorithm), but with LTL as the specification formalism. Finally, we present the conclusions and related future lines of research to be followed.

2 Specification Formalism

The specification formalism used in this paper is a propositional, linear and discrete time temporal logic with two different representations: one graphical and the other textual. Its key construct is the interval, which allows us to carry out logical reasoning at the level of time intervals, instead of instants. However, its primitive elements are not intervals but instants. An interval is therefore formed by identifying its end-points, which are instants satisfying certain properties. These points are searched for in the global context, which represents an infinite sequence of states corresponding to a system execution. Once the end-points of an interval have been located, the semantics of the nested formula (to the interval) is restricted to the subtrace delimited by these points. Each interval therefore represents a specific temporal context.

There are two main reasons for using this logic: firstly, the visually intuitive and natural representation of its graphical specifications makes them easier to develop and understand (even for experts involved in the system development process who are not familiar with this notation) than the textual representation of more traditional temporal logics, such as LTL (Linear Temporal Logic) [Manna, Pnueli 1992]; and secondly, in spite of having a graphical representation, it has a formally defined semantics on the basis of its textual representation.

2.1 Graphical Specifications of Temporal Properties

In order to specify the temporal properties of a system in a formalism that is very close to the way in which a human being reasons, we use the pictorial version of the logic, called GIL (Graphical Interval Logic) [Dillon et al. 1994].

The three graphical formulas in [Figure 1] show the basic types of properties that can be expressed over an interval. Thus, (a) is an *initial property*, which states that the formula *f* expressing such a property holds at the first state of the interval, where *f* is drawn left-justified below its left end-point; (b) represents an *invariant property* over the interval, where *f* is placed below it and indented to the right of its left end-point to express that *f* holds at every state of the interval; and (c) give us an *eventuality property* stating that *f* eventually holds at some state within the interval, where a diamond is placed on it with the target formula left-justified below the diamond.

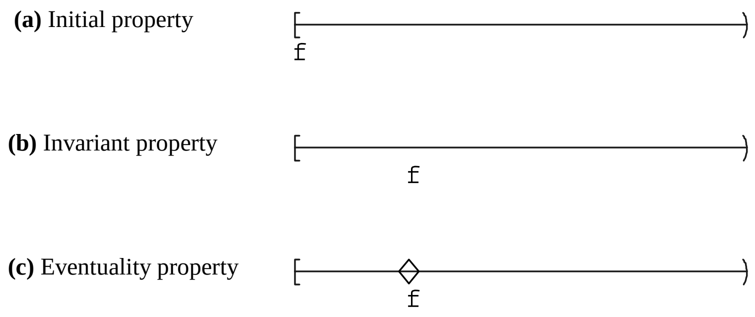


Figure 1: *GIL formulas expressing the basic types of properties over an interval*

In the formulas explained, *f* can be any GIL formula, even another interval formula, and the intervals in them can represent the global context or a subinterval extracted from a larger interval. Each end-point of a subinterval is defined by a *search pattern* represented by a horizontal concatenation of dashed search arrows, where search targets are left-justified below the arrowheads. Thus, the formula in [Figure 2] states that *i* is an initial property in the subinterval extracted from the global context by using a first search that locates the earliest state at which *f* holds (its left end-point), and from this, the right end-point is located by searching for the formula *h* from the state where *g* is found.

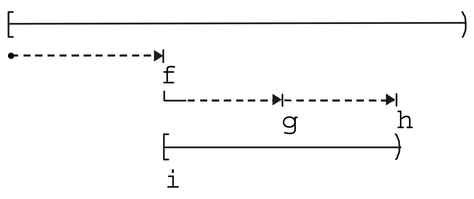


Figure 2: *Initial property within a subcontext*

GIL formulas can be joined using the usual infix Boolean operators laid out horizontally or vertically. In addition, a specification can be refined by replacing the names of the formulas appearing in it with any GIL formula. Following these rules, therefore, the (more abstract) specification of [Figure 2] is converted into the (more concrete and complex) formula of [Figure 3]. It should be noted that the conjunction symbol can be omitted in the vertical layout, as occurs between the two lower formulas (¬*i4* and *i5*) in [Figure 3].

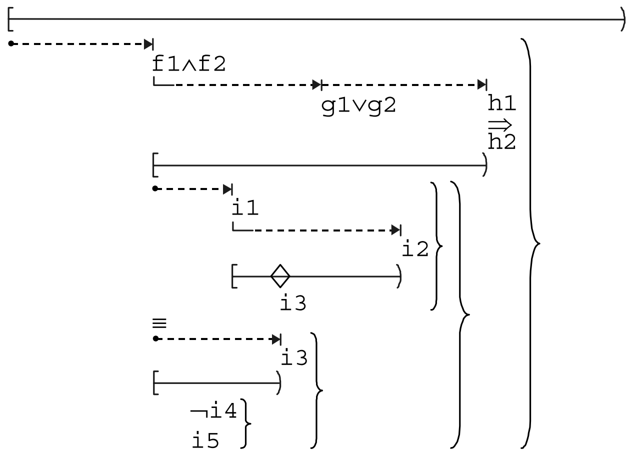


Figure 3: *Specification obtained by replacing the formulas f, g, h and i in [Figure 2]*

Every GIL formula is read from top to bottom and from left to right. The topmost interval represents the global context (i.e. the entire computation of the system). Braces can be used in order to disambiguate formulas. Consequently, it is clear in [Figure 3] that the equivalence of the two bottom intervals is nested to the previous one, which in turn is nested to the topmost interval. Further details about the visual syntax of GIL can be found in [Dillon et al. 1994].

2.2 FIL as the Textual Representation of GIL

We could say that FIL (Future Interval Logic) [Ramakrishna et al. 1992] is the formal basis for GIL and its textual representation since it serves as the basis for defining the semantics of all the constructions of the logic and there is an equivalence (established in [Dillon et al. 1994]) between the textual formulas of FIL and the corresponding graphical formulas in GIL.

The *syntax* of FIL for a finite set $\mathcal{P}$ of primitive propositions, where $p \in \mathcal{P}$, is defined as follows:

f	::==	$p \mid \neg f \mid f_1 \vee f_2 \mid I\,f$	/* FIL formulas */
I	::==	$[\theta_1 \mid \theta_2) \mid [- \mid \theta_2) \mid [\theta_1 \mid \rightarrow)$	/* Intervals */
θ	::==	$\rightarrow f \mid \rightarrow f, \theta$	/* Search patterns */

A FIL formula is *purely propositional* when it does not contain any interval. Otherwise, it is an *interval formula*, with its structure given by $I\,f$, where I represents an interval and f represents any other FIL formula nested to it. All the intervals are half-open, including their left but not their right end-point. Each interval end-point is defined by a *search pattern*, which is either a sequence of one or more searches or a *trivial* pattern (represented by – or $\rightarrow$). Each *search*, e.g. $\rightarrow f$, locates the first point in the reflexive future (which includes the current state) where the target formula f holds. When several searches are sequentially composed, such as in $\rightarrow g, \rightarrow h$, each subsequent search begins in the state located by the previous search; the last of these therefore locates the end-point of the interval that such a pattern defines. The *trivial search pattern* – leaves us at the point where we are, while $\rightarrow$ takes us to the end of the current context. Thus, the interval modality $[-|\theta_2)$ tries to construct a prefix of the current context, which goes from the current state to the state prior to the one located by the search pattern θ_2; whereas the expression $[\theta_1|\rightarrow)$ tries to construct a suffix of the current context, beginning in the state located by θ_1 and extending it until the last state of the current context.

Every interval modality $[\theta_1|\theta_2)$ defines a context, which is either the null context or the subsequence that begins in the state located by the search pattern θ_1, and finishes in the state prior to the one located by the search pattern θ_2. A *null context* occurs when a search fails (i.e. its target formula cannot be located in the current context) or, since the searches of both end-points start at the same state, when the state located by θ_1 does not precede the state located by θ_2; the interval cannot therefore be constructed, which is why the formula $[\theta_1|\theta_2)f$ is assumed to be *vacuously* satisfied. The semantic interpretation of $[\theta_1|\theta_2)f$ would therefore be: "If the interval $[\theta_1|\theta_2)$ can be identified within the current context, then f must be satisfied within the subcontext that it represents". This *default-to-true* semantics yields the following meaning for $\neg[\theta_1|\theta_2)f$: "An interval of the form $[\theta_1|\theta_2)$ compulsorily exists in the reflexive future and f does not hold at its first state". The complete FIL formal semantics can be found in [Ramakrishna et al. 1996].

The other standard constructs of Propositional Logic are defined as abbreviations of certain expressions, i.e. $\mathbf{T}{=}p\vee\neg p$, $\mathbf{F}{=}\neg\mathbf{T}$, $\neg\neg f = f$, $f_1\wedge f_2 = \neg(\neg f_1\vee\neg f_2)$ and $f_1\Rightarrow f_2 = \neg f_1\vee f_2$. The *restricted* syntax presented above for FIL can be extended with several LTL temporal operators, defined as abbreviations for the following interval formulas. Since the logical constant $\mathbf{F}$ can only hold in the null context (which is given by an interval that cannot be constructed), the formula $[\rightarrow\neg f\,|\rightarrow)\mathbf{F}$ can never therefore be satisfied in a trace in which $\neg f$ holds at some state of the reflexive future, with this formula being equivalent to $\Box f$. Its dual, $\neg[\rightarrow f\,|\rightarrow)\mathbf{F}$, states that there is some instant in the future where f holds, which is why it is equivalent to $\Diamond f$. The operator *strong until* is defined as $f_1 \mathrm{U} f_2 = \neg[\rightarrow(\neg f_1\vee f_2)|\rightarrow)\neg f_2$.

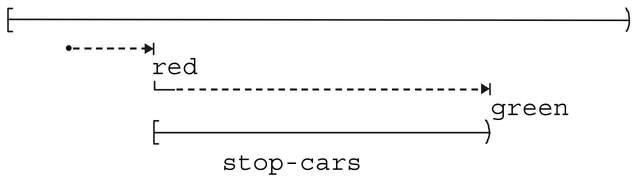
Consequently, using the *extended* syntax of FIL, the formulas corresponding to the GIL ones in [Figure 1] are: (a) f, (b) $\Box f$, and (c) $\Diamond f$, while $[\rightarrow f\,|\rightarrow f,\rightarrow g,\rightarrow h)i$ is the equivalent FIL formula to the GIL one represented in [Figure 2] and this other $[\rightarrow f1\wedge f2|\rightarrow f1\wedge f2,\rightarrow g1\vee g2,\rightarrow h1\Rightarrow h2)([\rightarrow i1|\rightarrow i1,\rightarrow i2)\Diamond i3\equiv[-|\rightarrow i3)\Box(\neg i4\wedge i5))$ is the analogous one to the formula shown in [Figure 3].

2.3 Examples of Specifications: Properties of a Traffic Light

In this subsection, we present some examples of how certain temporal properties (security, recurrence, precedence and response) are specified in both representations (GIL and FIL) by applying them to the specification of a real-world system, specifically a traffic light.

The meaning of the *security property* represented in [Figure 4] is that whenever the traffic light is red (i.e. in all the intervals from when the traffic light changes to red until it changes to green again), the cars must stop.

(a)

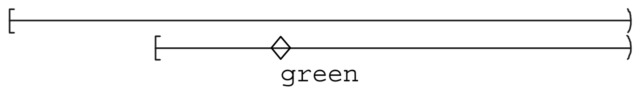


(b) □[→red|→red,→green)□stop-cars

Figure 4: *Security property expressed in:* ***(a)*** *GIL, and* ***(b)*** *FIL*

The formula of [Figure 5], which can be read as “infinitely often the green lights”, asserts that for each instant of the execution, there will always be a state in the future at which the traffic light is green, which explains why it is a *recurrent property*. It should be noted that the right end-points of both intervals must match, since every time the green must come on between the corresponding current state and the end of the current context, which is given by the upper interval.

(a)

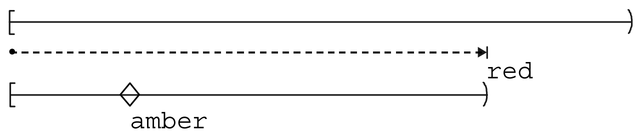


(b) □◊green

Figure 5: *Recurrent property expressed in:* ***(a)*** *GIL, and* ***(b)*** *FIL*

The specification of [Figure 6] is a *property of precedence*, since it expresses the condition that before the red lights, the amber must light at some previous instant.

(a)

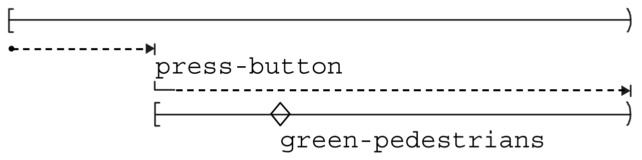


(b) [−|→red)◊amber

Figure 6: *Property of precedence expressed in:* ***(a)*** *GIL, and* ***(b)*** *FIL*

Finally, the formula of [Figure 7] states that in response to the request made by a pedestrian pressing the button, the traffic light will eventually respond by lighting the green for the pedestrians. It is therefore a *property of response*.

(a)



(b) [→press-button|→)◊green-pedestrians

Figure 7: *Property of response expressed in:* ***(a)*** *GIL, and* ***(b)*** *FIL*

3 Design and Implementation of FBT

3.1 General Structure of the Tool

The structure of the different classes of FIL formulas considered in the design and implementation of FBT is described in [Figure 8], using a class diagram in UML (Unified Modelling Language) [Rumbaugh et al. 1999]. *Fil* is an abstract class that defines the elements which are common to the different types of formulas, defined in its subclasses: *FilAtom*, the class that represents the literals (atomic propositions, negated or not); *FilConstant* stands for the logical constants (**T** or **F**); *FilJunct* implements the conjunctions and disjunctions of (two) FIL formulas, while *FilIff* does something similar, but with equivalences and exclusive disjunctions; and *FilInterval*, the class that builds interval formulas, comprising two search patterns which attempt to locate each end-point of the interval, and a FIL formula nested to this. An instance of the class *SearchPattern* is a sequence of zero (if it is a trivial pattern) or more FIL formulas, as many searches as comprise that pattern.

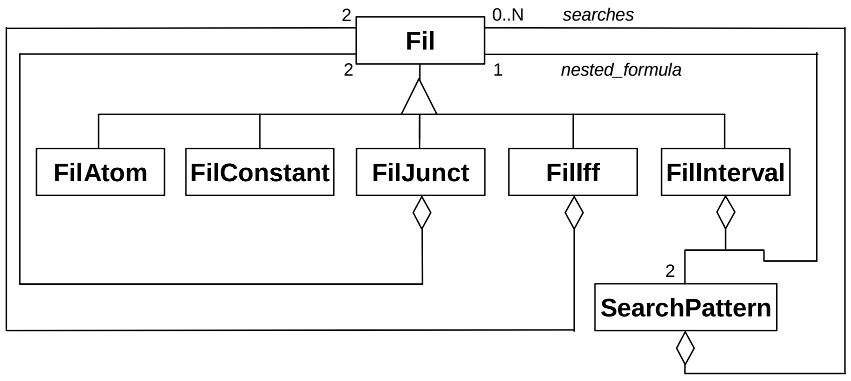


Figure 8: *UML class diagram representing the different types of FIL formulas*

The UML class diagram shown in [Figure 9] describes the structure of the graph that FBT generates from a FIL formula. *FilGraph* is therefore the class representing the graph, which is formed by the aggregation of nodes (i.e. objects of class *FilGraphNode*). Two nodes of the graph are related if a transition exists between them. This has been represented by means of an association with two role names: *predecessor* and *successor*. The structure of each node is constituted for the aggregation of the following sets of (zero or more) FIL formulas:

- *New*: Temporal properties that must hold in the node and have not yet been processed. When a node has been processed completely, this set is empty, which is why all the resulting nodes do not contain any formula in it.
- *Old*: Formulas that must hold in the node and have already been analysed. It should be noted that all the formulas that finally constitute this set in a node have previously been part of the set *New* of the same node.
- *Next*: Temporal properties that must be satisfied in all the next nodes (i.e. states which are the immediate successors of the node). This set can only contain interval formulas, since these are the only FIL formulas that can postpone their fulfilment.
- *Literals*: Literals stored in *Old*. Although this is a redundant set, it is used to obtain greater efficiency, since certain searches and verifications can be carried out more quickly on this set than on the set *Old*.

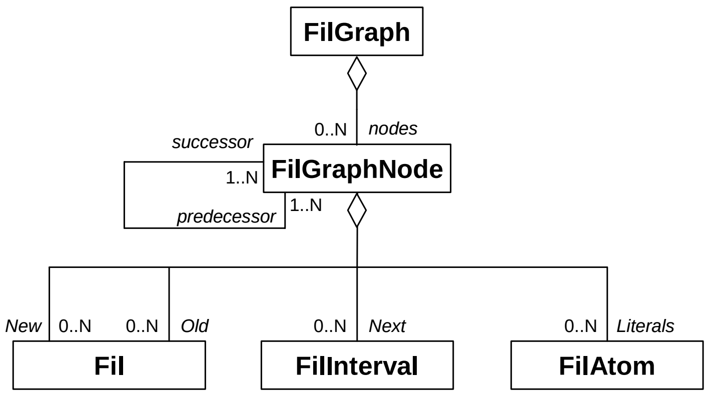


Figure 9: *Structure in UML of the graph generated from a FIL formula*

3.2 About the Implementation

The implementation of FBT is based on the C++ [Stroustrup 1986] code of LBT (LTL to Büchi automaton Translator)[1], a tool of similar characteristics, since it is based on an on-the-fly tableau algorithm [Gerth et al. 1995], but with LTL as the specification formalism. Consequently, both tools share the same input and output interface, and the syntax and notation for all the formula components that are commonly accepted by both, i.e. the logical constants, literals, propositional operators and temporal operators

[1] http://www.tcs.hut.fi/Software/maria/tools/lbt/

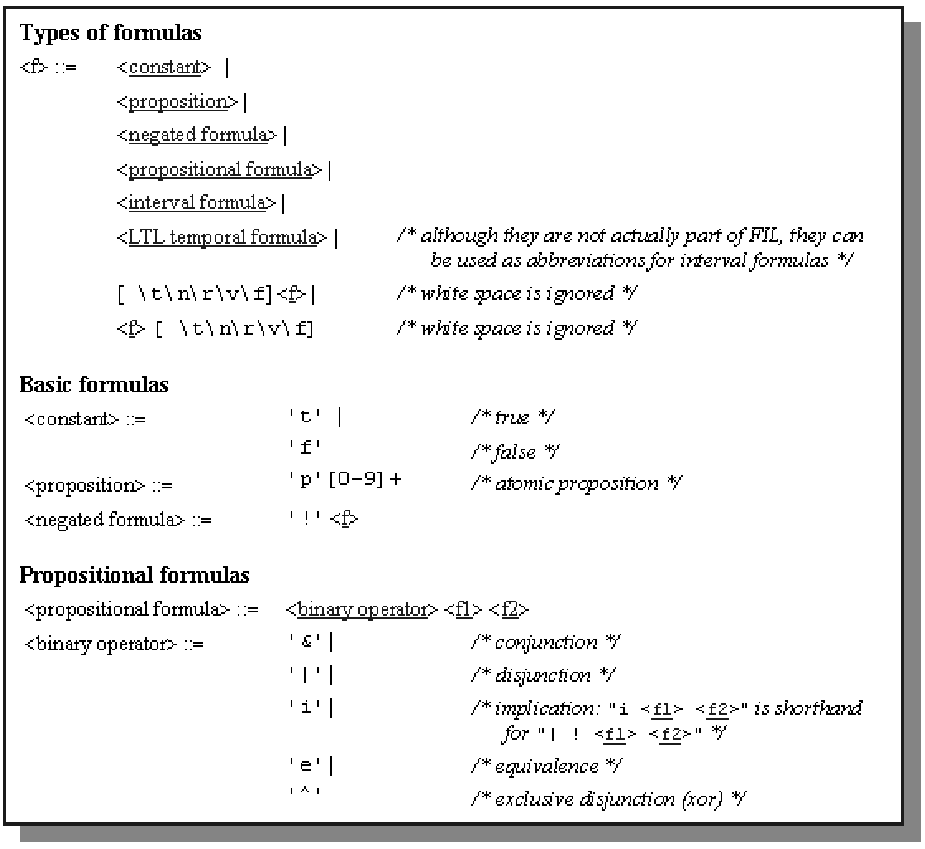
of LTL. Our initial intention was to integrate FBT into MARIA [Mäkelä 2002], a tool that performs on-the-fly model checking. This was the main reason for reusing part of the LBT code in order to make integration easier, since LBT is the translator used in MARIA. Obviously, we have had to incorporate a series of specific classes and functions for the analysis of interval formulas, which are not present in LTL. We have also implemented a series of heuristics in order to improve and optimise the code; we have therefore managed to generate automata with fewer nodes and edges and in less time, and to determine the accepting states more quickly than if these heuristics had not been used. Among these heuristics, the following should be highlighted:

1. During the expansion process of the graph nodes, FBT considers two nodes to match (i.e. they represent the same state) if their fields *Literals* and *Next* store the same respective set of formulas, and this is why the two nodes are merged into one. LBT, however, only fuses two nodes when the fields *Old* and *Next* of one node contain the same respective sets of formulas as their homologous ones in the other node. The advantage of applying our approach is that in many cases it is possible to reduce the number of nodes generated with regard to those that would be produced if the condition considered in LBT had been implemented.
2. The first thing that FBT does with the chosen formula (η) from the field *New* is to check whether its negation ($\neg\eta$) is included in the field *Old* and if so, this node should be discarded. In this way, the contradictions in a node are rapidly detected, thus avoiding the processing of many formulas of a node (which will eventually be discarded) until we reach the level of literals, which is where the contradictions are detected in LBT. With this heuristics, we are able to carry out the expansion process of interval formulas more quickly and efficiently.
3. In order to establish the acceptance sets (i.e. the acceptance conditions), our translator uses an improved heuristics with respect to LBT, where the search for the formulas expressing eventuality (i.e. those having either the operator U (*strong until*) or the operator $\Diamond$ (*finally*, *eventually*) as the main operator), is carried out in the field *Old* of the nodes stored in the set of the graph nodes, instead of in the field *Next*, as FBT does. This last field obviously contains fewer formulas than the previous field as it only stores the temporal formulas (interval formulas, in the case of FBT) that must be satisfied (although not immediately) in the state which that node represents. The field *Old*, meanwhile, holds not only these formulas, but also all the propositional and temporal formulas that have been processed in such a state. Some of these formulas are either totally or partially satisfied in that state and they are not therefore present in the field *Next*. Consequently, this heuristics enables FBT to calculate the acceptance conditions more quickly.
4. In order to determine the accepting states, i.e. the states that belong to each acceptance set, FBT need only check that none of the eventuality formulas associated to the corresponding acceptance condition is stored in the field *Next* of a node. By checking in the field *Next* rather than in the field *Old* (as LBT does) our procedure is able to gain the advantages of speed and simplicity. Moreover, in order to determine whether a state belongs to an acceptance set or not, LBT must check not only for the presence or absence of the corresponding eventuality formula (which is either of the type $\Diamond f$ or $g\mathrm{U}f$, where f and g represent any formula accepted by LBT), but also for the formula that satisfies that eventuality (f).

Consequently, the procedure for calculating the accepting states is more complex in LBT than in FBT.

3.3 Input and Output Interface

Designed to be invoked as a subprocess for an on-the-fly model checker, FBT analyses a FIL formula supplied in textual format using the standard input. Once it has been processed, FBT writes (also in textual format) the generalized Büchi automaton which is semantically equivalent to this formula in the standard output. Both in the input and output, the formulas are written in *prefix notation*, since this facilitates the recursive-descent parsing that our translator must carry out. The final part of this section explains how a graphical representation may be obtained from the textual output generated by FBT.



Types of formulas

```
<f> ::=   <constant> |
          <proposition> |
          <negated formula> |
          <propositional formula> |
          <interval formula> |
          <LTL temporal formula> |    /* although they are not actually part of FIL, they can
                                         be used as abbreviations for interval formulas */
          [ \t\n\r\v\f]<f> |          /* white space is ignored */
          <f> [ \t\n\r\v\f]           /* white space is ignored */
```

Basic formulas

```
<constant> ::=                't' |               /* true */
                              'f'                 /* false */
<proposition> ::=             'p'[0-9]+           /* atomic proposition */
<negated formula> ::=         '!' <f>
```

Propositional formulas

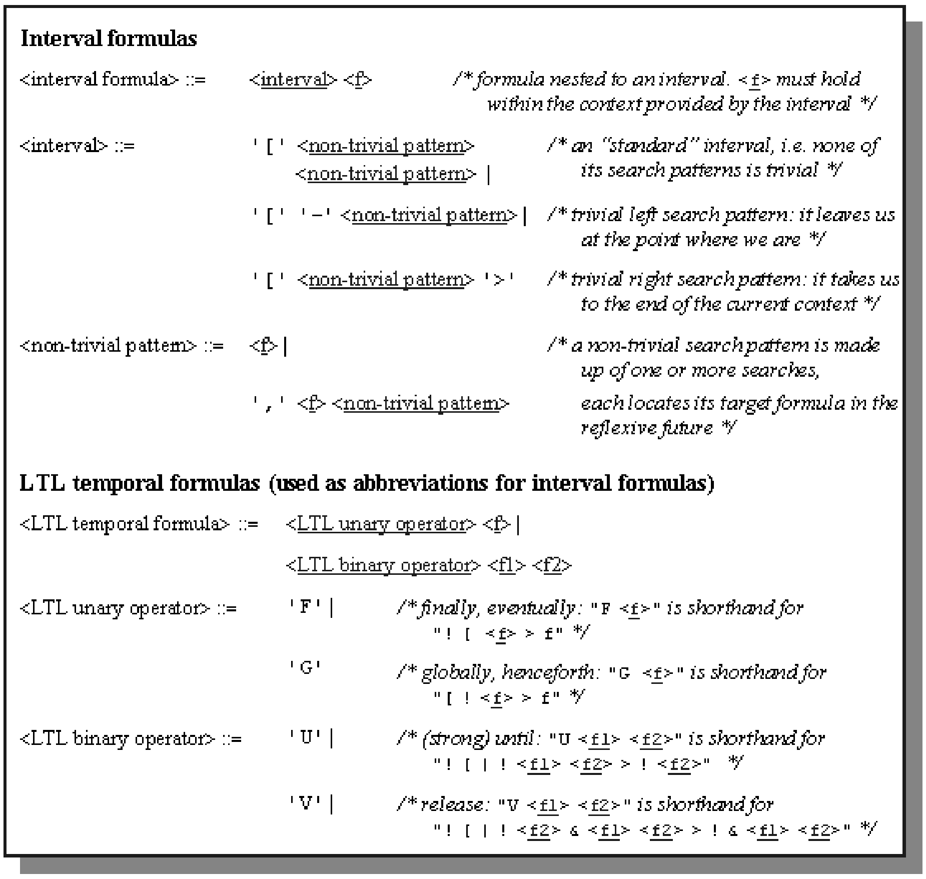
```
<propositional formula> ::=   <binary operator> <f1> <f2>
<binary operator> ::=         '&'|                /* conjunction */
                              '|'|                /* disjunction */
                              'i'|                /* implication: "i <f1> <f2>" is shorthand
                                                     for "| ! <f1> <f2>" */
                              'e'|                /* equivalence */
                              '^'                 /* exclusive disjunction (xor) */
```

Figure 10: *Syntax of the formulas accepted by FBT (I): the simplest formulas*

3.3.1 Syntax of the Input Formulas

This subsection presents the grammar that defines the syntax of all the types of formulas accepted by FBT as input. The grammar, which is shown using the BNF

(Backus-Naur Form) [Naur 1960], has been divided into two figures in order to obtain a more elegant presentation, since not all its production rules fit into the available space of a page. [Figure 10] shows the rules which build the simplest types of FIL formulas, while [Figure 11] describes how the formulas containing some temporal operators are formed. Comments delimited by the symbols /* and */ have been added to the right of certain production rules in both figures. These comments and the expressions in bold that separate the grammar rules into different groups or sections (so as to make them easier to read) are not part of the formal grammar. Terminal symbols are enclosed within single quotes or presented as regular expressions in the style used by the lexical analyser generator FLEX[2]. Non-terminal symbols are enclosed within angles, and these are also represented in the usual link-style (i.e. underlined) when they appear in the righthand part of a production rule.



Interval formulas

```
<interval formula> ::=      <interval> <f>            /* formula nested to an interval. <f> must hold
                                                         within the context provided by the interval */

<interval> ::=              '[' <non-trivial pattern>         /* an "standard" interval, i.e. none of
                                <non-trivial pattern> |          its search patterns is trivial */

                            '[' '-' <non-trivial pattern> |   /* trivial left search pattern: it leaves us
                                                                 at the point where we are */

                            '[' <non-trivial pattern> '>'     /* trivial right search pattern: it takes us
                                                                 to the end of the current context */

<non-trivial pattern> ::=   <f> |                             /* a non-trivial search pattern is made
                                                                 up of one or more searches,

                            ',' <f> <non-trivial pattern>        each locates its target formula in the
                                                                 reflexive future */
```

LTL temporal formulas (used as abbreviations for interval formulas)

```
<LTL temporal formula> ::=   <LTL unary operator> <f> |

                             <LTL binary operator> <f1> <f2>

<LTL unary operator> ::=     'F' |      /* finally, eventually: "F <f>" is shorthand for
                                            "! [ <f> > f" */

                             'G'        /* globally, henceforth: "G <f>" is shorthand for
                                            "[ ! <f> > f" */

<LTL binary operator> ::=    'U' |      /* (strong) until: "U <f1> <f2>" is shorthand for
                                            "! [ | ! <f1> <f2> > ! <f2>"  */

                             'V' |      /* release: "V <f1> <f2>" is shorthand for
                                            "! [ | ! <f2> & <f1> <f2> > ! & <f1> <f2>" */
```

Figure 11: *Syntax of the formulas accepted by FBT (II): temporal formulas*

[2] http://www.gnu.org/software/flex/flex.html

It should be noted that FBT also accepts LTL temporal operators, but only as abbreviations of the corresponding FIL interval formulas (as indicated in the first comment of [Figure 10] and also in [Figure 11]). This means that once FBT has parsed one of these formulas, it immediately transforms it into its equivalent interval formula, which is the one that it actually stores and processes. Consequently, all the temporal formulas that our algorithm must decompose (i.e. expand) are interval formulas, and therefore it has no expansion rule for decomposing formulas with LTL temporal operators, since these operators are not present in the internal formulas that must be processed. In other words, the interval is the only temporal operator that the algorithm must consider internally.

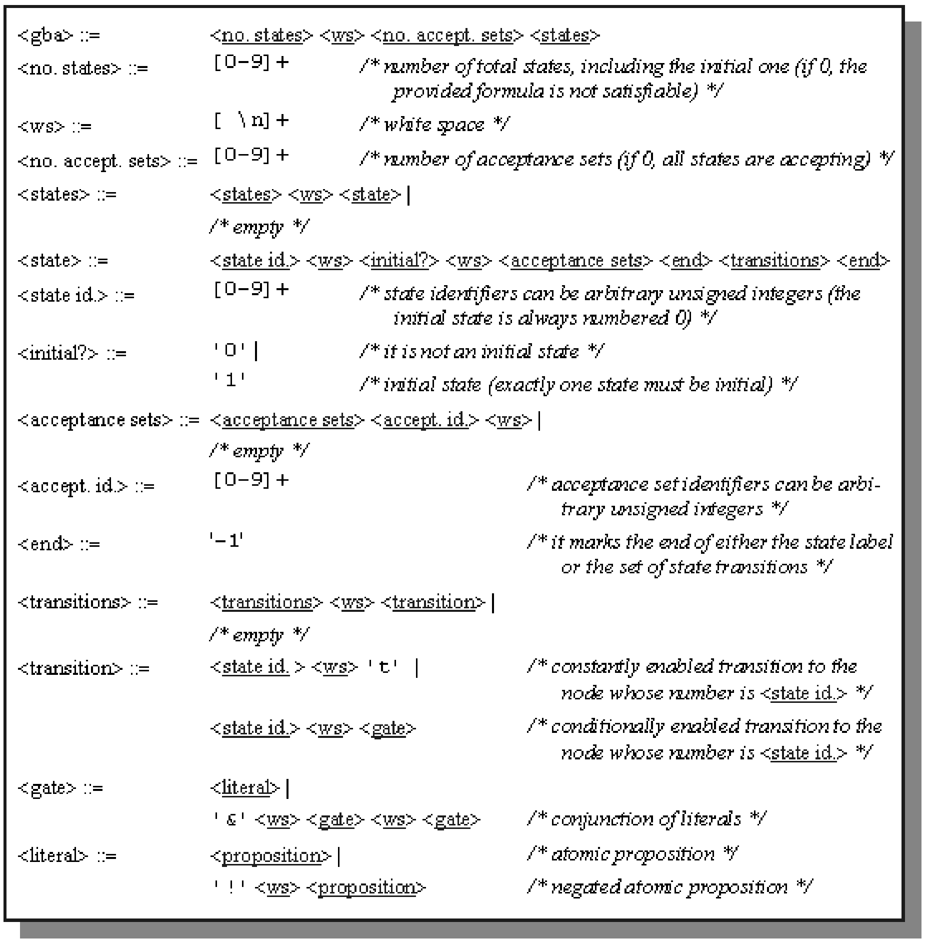
It can be seen in the last two lines of the first production rule in [Figure 10] that FBT ignores white spaces, horizontal and vertical tabulators (`\t` and `\v`), carriage returns (`\r`), and line and form feeds (`\n` and `\f`). We can therefore introduce any number of these separators between two terms of the formula that we want to supply as input to FBT.

3.3.2 Syntax of the Generated Output

[Figure 12] shows the syntax used by FBT for returning the generalized Büchi automaton (*gba*, as it is referred to in the lefthand part of its first production rule), which is equivalent to the FIL formula that was introduced as input. It should be noted that the same notation explained in the previous subsection has been used in this figure, and that the non-terminal symbol <proposition> used in it is defined in [Figure 10].

The following example attempts to clarify the grammar presented in [Figure 12], and also to explain how the tool may be operated and some of its main characteristics.

Example 1: Let us suppose that the formula $\neg\Box p0$ (i.e. `!Gp0`, in the FBT syntax) is supplied to the tool and that we want to obtain the corresponding output in a file named `automaton.txt`. In this case, the command that we would have to type would be: `echo '!Gp0' | fbt >automaton.txt`. If this formula is the content of a file (for example: `formula.txt`), then we should type: `fbt <formula.txt >automaton.txt`. Obviously, if we want the output to appear on the screen, we only need to omit the last part of the command (from the redirection symbol `>`) in both cases. [Figure 13] shows what FBT returns after executing one of the previous commands (i.e. the contents of the file `automaton.txt`) and explains its meaning. FBT converts the formula given into its equivalent interval formula: `![!p0 > f` (i.e. $\neg[\rightarrow\neg p0 \,|\!\rightarrow)\mathbf{F}$, in the standard FIL syntax presented in section 2.2), which is the one that it actually stores and processes. FBT would obviously perform in exactly the same way and would generate the same output if the formula inputted had been, for example, any of the following: `F!p0` ($\Diamond\neg p0$), `| F!p0 !Gp0` ($\Diamond\neg p0 \vee \neg\Box p0$) or `& F!p0 !Gp0` ($\Diamond\neg p0 \wedge \neg\Box p0$), since all of these are equivalent.



```
<gba> ::=              <no. states> <ws> <no. accept. sets> <states>
<no. states> ::=       [0-9]+          /* number of total states, including the initial one (if 0, the
                                          provided formula is not satisfiable) */
<ws> ::=               [ \n]+          /* white space */
<no. accept. sets> ::= [0-9]+          /* number of acceptance sets (if 0, all states are accepting) */
<states> ::=           <states> <ws> <state> |
                       /* empty */
<state> ::=            <state id.> <ws> <initial?> <ws> <acceptance sets> <end> <transitions> <end>
<state id.> ::=        [0-9]+          /* state identifiers can be arbitrary unsigned integers (the
                                          initial state is always numbered 0) */
<initial?> ::=         '0' |           /* it is not an initial state */
                       '1'             /* initial state (exactly one state must be initial) */
<acceptance sets> ::=  <acceptance sets> <accept. id.> <ws> |
                       /* empty */
<accept. id.> ::=      [0-9]+                          /* acceptance set identifiers can be arbi-
                                                          trary unsigned integers */
<end> ::=              '-1'                            /* it marks the end of either the state label
                                                          or the set of state transitions */
<transitions> ::=      <transitions> <ws> <transition> |
                       /* empty */
<transition> ::=       <state id. > <ws> 't'  |        /* constantly enabled transition to the
                                                          node whose number is <state id.> */
                       <state id.> <ws> <gate>         /* conditionally enabled transition to the
                                                          node whose number is <state id.> */
<gate> ::=             <literal> |
                       '&' <ws> <gate> <ws> <gate>     /* conjunction of literals */
<literal> ::=          <proposition> |                 /* atomic proposition */
                       '!' <ws> <proposition>          /* negated atomic proposition */
```

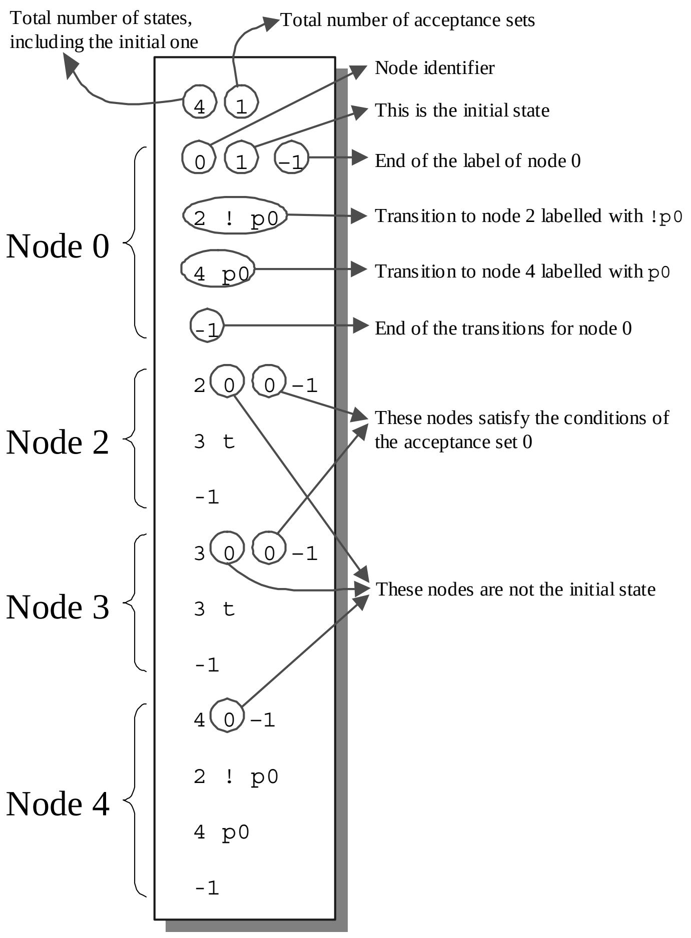
Figure 12: *Syntax of the output generated by FBT*

3.3.3 Graphical Visualization of the Generated Automata

In order to graphically visualize the textual output generated by FBT, we should execute a filter, which we have named GBA2DOT since it converts the generalized Büchi automaton that FBT produces into the language *dot*, which is directly accepted by the directed graph visualization tool GRAPHVIZ[3] [Gansner, North 2000].

The following example shows how a graphical representation may be obtained from the textual output produced by our translator. It uses the result obtained in [Example 1] (i.e. the contents of the file `automaton.txt`), which is shown in [Figure 13].

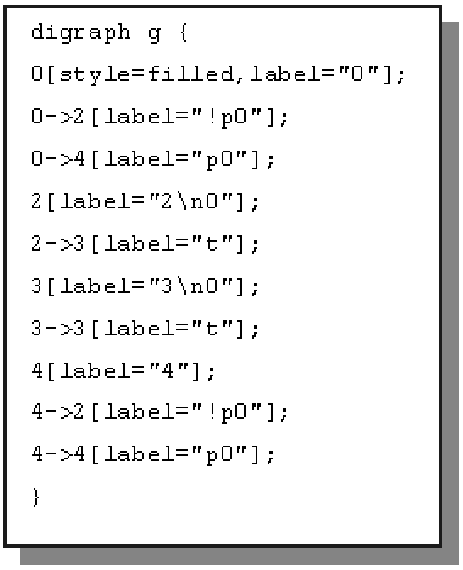
[3] http://www.graphviz.org/


Total number of states, including the initial one
Total number of acceptance sets
Node identifier
This is the initial state
End of the label of node 0
Transition to node 2 labelled with !p0
Transition to node 4 labelled with p0
End of the transitions for node 0
These nodes satisfy the conditions of the acceptance set 0
These nodes are not the initial state
Node 0
Node 2
Node 3
Node 4

```
4 1
0 1 -1
2 ! p0
4 p0
-1
2 0 0 -1
3 t
-1
3 0 0 -1
3 t
-1
4 0 -1
2 ! p0
4 p0
-1
```

Figure 13: *Textual output generated by FBT from the formula* ¬□p0

Example 2: The execution of the command `gba2dot <automaton.txt >graph.txt` translates the generalized Büchi automaton stored in the file `automaton.txt` (presented in [Figure 13]) into the language *dot*. The execution result of the previous command is shown in [Figure 14] and stored in the file `graph.txt`. From this file, with its easier to understand contents, the visualization tool GRAPHVIZ automatically generates the corresponding graphical representation. We therefore only need to type the following command: `dotty - <graph.txt`, and the window presented in [Figure 15] will appear on the screen. This contains the graphical representation of the generalized Büchi automaton that FBT produces for the input formula !`Gp0` (i.e. ¬□p0).



```
digraph g {
0[style=filled,label="0"];
0->2[label="!p0"];
0->4[label="p0"];
2[label="2\n0"];
2->3[label="t"];
3[label="3\n0"];
3->3[label="t"];
4[label="4"];
4->2[label="!p0"];
4->4[label="p0"];
}
```

Figure 14: *Filter execution result for the output generated by FBT from the formula ¬□p0*

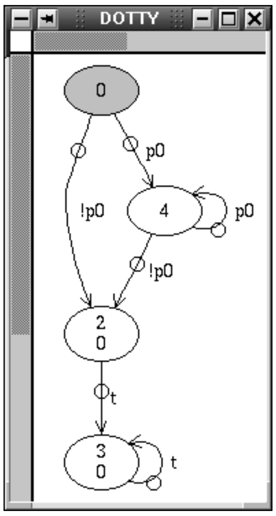


Figure 15: *Graphical representation of the Büchi automaton generated by FBT from ¬□p0*

Additional details about the tool design and implementation can be found in [Hornos 2002] and at http://www-lsi.ugr.es/~mhornos/fbt, where the tool code can be downloaded.

4 Experimental Results

As FBT has been implemented from the LBT code, this section not only presents some experimental results obtained with our tool, but also those generated by LBT for the same or equivalent specifications so that both tools may be compared.

As [Figure 11] shows, not only does FBT recognize the only temporal operator of FIL, i.e. the interval, but also the following temporal operators of LTL: $\Box$ (*always* or *henceforth*) and its dual $\Diamond$ (*eventually*), and U (*until strong*) and its dual V (*release*), but only as abbreviations of the corresponding FIL interval formulas. FBT therefore accepts LTL formulas, such as the one shown in [Example 3], but automatically transforms them into their FIL equivalents, these being the ones that it actually stores and processes, as mentioned previously.

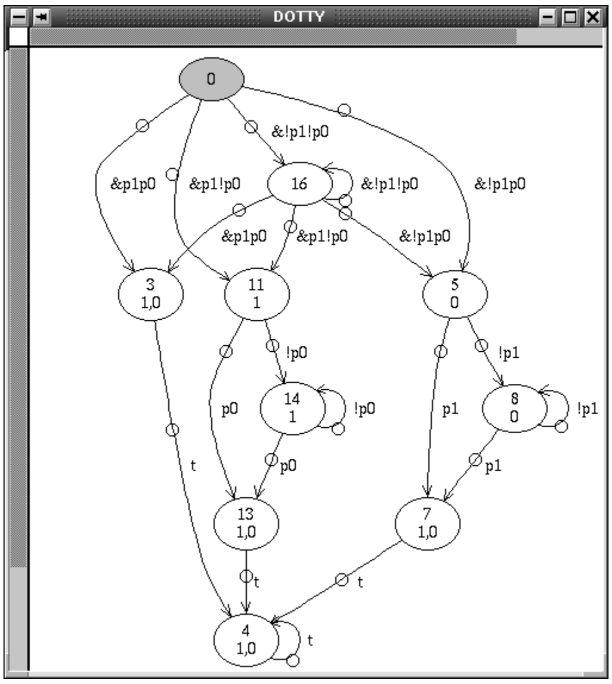


Figure 16: *Büchi automaton generated by FBT from the formula* $\Diamond$p0 $\wedge$ $\Diamond$p1

Example 3: For the input specification $\Diamond$p0 $\wedge$ $\Diamond$p1, FBT processes the FIL formula $\neg[\rightarrow p0|\rightarrow)_F \wedge \neg[\rightarrow p1|\rightarrow)_F$ in order to build its semantically equivalent automaton, which is graphically represented in [Figure 16]. It should be noted that each edge is labelled with the conjunction of literals (in prefix notation) that enables that transition. The upper number labelling each node is its state identifier, while the lower numbers identify the acceptance conditions that it satisfies. The initial state is always shaded and numbered 0. In order to compare the automata generated by FBT and LBT, we count the number of nodes, edges and acceptance conditions, as well as the number of states that satisfy each condition. Since both tools produce *generalized* Büchi automata, the resulting automaton generally have *k* acceptance conditions; each one defines a set of accepting states, F_i (with i=1..k), which contains those states that satisfy it. The automaton in [Figure 16] has 9 nodes (the initial node is not considered), 20 edges (the ones leaving the initial node are counted), and two acceptance conditions (k=2), the first (identified by the number 0) is satisfied by six states (nodes 3, 4, 5, 7, 8 and 13), while the second one (identified by the number 1) is satisfied by another six (nodes 3, 4, 7, 11, 13 and 14). When k>1, if we want to obtain a *classic* Büchi automaton, i.e. one with only one set of accepting states, *F*, we only need to obtain the states that are in the intersection of the sets F_i (i.e. $F=\cap F_i$). There are therefore four accepting states in our example (nodes 3, 4, 7 and 13).

[Table 1] gathers the values counted in the automata generated by FBT and LBT for various input specifications which only have LTL temporal operators. The one explained in [Example 3] is shown in Case 5. Each case occupies two rows: the first corresponds to the results obtained with LBT, and the second is for those produced by FBT. It should be noted that in the second row of each case, the input specification is represented in the extended syntax of FIL, while the processed formula is expressed in its restricted syntax. We can observe that for the simplest specifications (Cases 1 and 2), the same values are obtained in the automata generated by both tools (except in the column Time, a variable which will be discussed at a later stage). However, for slightly more complex specifications (Cases 4, 5 and 6), FBT generally produces smaller automata than those obtained with LBT. The formulas containing either the operator U or its dual V are usually the exception to this rule, since for these, LBT usually produces automata which are slightly simpler than those generated by FBT (Case 3). This is due to the fact that the corresponding interval formula which is analysed by FBT attempts to "simulate" the property that these operators represent more naturally. Something similar happens in the opposite way with interval formulas: since LBT does not admit interval formulas, it is necessary to resort to appreciably more complicated expressions (see [Table 2]) in order to provide it with an equivalent LTL formula, as [Example 4] explains in detail for Case 1 of [Table 2].

Example 4: Since LBT does not recognize the formula $[\rightarrow p0|\rightarrow p1)\Box\neg p2$, we must input an equivalent LTL formula. The semantics associated with this interval formula indicates that it holds whenever one of the following four conditions is fulfilled:

1. p0 does not hold in the reflexive future, i.e. the LTL formula $\Box\neg p0$ holds.
2. p1 never holds in the reflexive future, i.e. $\Box\neg p1$ is true.
3. p1 precedes p0, which is expressed as $p1 \mathcal{V} \neg p0$ in LTL. This formula asserts that in the first state where p1 holds as well as in all the previous ones to it $\neg p0$ holds.
4. Either p0 and p1 hold in the same state or p0 holds strictly before p1 and (in this latter case) $\neg p2$ is invariantly satisfied from the instant in which p0 holds until p1 is fulfilled. Both conditions are formulated in LTL as $\neg p1 \,\mathcal{U}\, (p0 \wedge \neg p2 \,\mathcal{U}\, p1)$.

Consequently, the equivalent formula to $[\rightarrow p0|\rightarrow p1)\Box\neg p2$ that must be inputted into LBT is the disjunction of the previous four LTL formulas, i.e. $\Box\neg p0 \vee \Box\neg p1 \vee p1\mathcal{V}\neg p0 \vee \neg p1 \;\mathcal{U}\; (p0 \wedge \neg p2 \;\mathcal{U}\; p1)$. The interval formula clearly expresses the explained property more concisely and elegantly. Moreover, the analysis carried out by FBT produces a simpler automaton (see Case 1 in [Table 2]).

	Specification		Size		Accepting states			Time
	Input	**Processed**	**Nodes**	**Edges**	***k***	**$\|F_i\|$**	**$\|\cap F_i\|$**	**µs**
1	□¬p0		1	2	0	—	—	30
	□¬p0	[→p0|→)**F**	1	2	0	—	—	120
2	◊p0		3	6	1	2	—	50
	◊p0	¬[→p0|→)**F**	3	6	1	2	—	238
3	p1 U p2		3	6	1	2	—	60
	p1 U p2	¬[→(¬p1∨ p2)|→)¬p2	4	9	1	3	—	624
4	◊◊p1		6	13	2	4, 5	3	98
	◊◊p1	¬[→¬[→p1|→)**F**|→)**F**	4	8	2	3, 3	2	435
5	◊p0 ∧ ◊p1		13	29	2	8, 8	4	232
	◊p0 ∧ ◊p1	¬[→p0|→)**F** ∧ ¬[→p1|→)**F**	9	20	2	6, 6	4	999
6	□◊p1 ⇒ □◊p2		9	19	2	7, 7	5	172
	□◊p1 ⇒ □◊p2	¬[→[→p1|→)**F**|→)**F** ∨ [→[→p2|→)**F**|→)**F**	5	15	3	4, 3, 4	2	900

Table 1: *Comparison of results obtained with LBT* ⬜ *and FBT* ▨ *from LTL formulas*

The contents of [Table 2] are similar to those of [Table 1] but for more complex properties which are inputted as interval formulas to FBT and as their equivalent LTL formulas to LBT. Except for the formulas of Case 1 (explained in [Example 4]), the remaining formulas of [Table 2] have been taken from an online repository of property specification patterns[4] that are commonly used in the specification and (finite-state) verification of concurrent and reactive systems. These patterns, which are available in various formalisms (including LTL and GIL), are classified in terms of the kinds of properties they describe and are defined using specific scopes (i.e. the extent of the execution over which the property must hold). Case 2 specifies that p0 is always false before p1; it is therefore an *absence property* defined over the scope *before*. Case 3 states that p0 becomes true after p1; it is therefore an *existence property* defined over the scope *after*. Case 4 represents a *universality property* over the scope *between*, since it specifies that p0 is always true between p1 and p2. Finally, the two last cases are *bounded existence properties*, meaning that transitions to p0-states occur at most 2 times *globally* (i.e. its scope is the entire execution) in Case 5 and *before* p1 in Case 6.

In both tables, the last column indicates the average execution time taken by each tool to translate the corresponding formula. These times have been calculated on average by running the same formula 30 times under the SuSE Linux 7.2 operating system on a PC with an AMD Athlon 1 GHz processor and 256 MB of RAM. It should be noted that these times are expressed in terms of microseconds (µs) in [Table 1], while milliseconds (ms) are used in [Table 2]. It can be appreciated that LBT is faster than FBT for all the formulas analysed. This is due to the fact that processing interval formulas is more complex than processing LTL formulas since every expansion (or tableau) rule of the LBT algorithm [Gerth et al. 1995] generates two new formulas at most, whereas the expansion rules of the FBT algorithm [Hornos, Capel 2002] can generate more than two new formulas for each analysed formula

[4] http://patterns.projects.cis.ksu.edu/documentation/patterns.shtml

when it is decomposed (i.e. expanded). Nevertheless, the running time of FBT can be perfectly assumed since it is sufficiently small, and in most of the analysed formulas, the resulting automata are simpler than those produced by LBT, as both tables show. Particularly significant is the difference in size between the automata generated by LBT and FBT in bounded existence properties (Cases 5 and 6 of [Table 2]).

	Specification		Size		Accepting states			Time
	Input	**Processed**	**Nodes**	**Edges**	***k***	**|*F**i*|**	**|∩*F**i*|**	**µs**
1	□¬p0 ∨ □¬p1 ∨ p1V¬p0 ∨ ¬p1 U (p0 ∧ ¬p2 U p1)		17	33	2	15, 14	12	0.3
	[→p0|→p1)□¬p2	[→p0|→p1)[→p2|→)**F**	12	25	2	9, 9	7	3.4
2	◊p1 ⇒ (¬p0 U p1)		7	12	1	5	—	0.1
	[−|→p1)□¬p0	[−|→p1)[→p0|→)**F**	4	8	1	3	—	0.6
3	□¬p1 ∨ ◊(p1 ∧ ◊p0)		11	22	2	8, 9	6	0.2
	[→p1|→)◊p0	[→p1|→)¬[→p0|→)**F**	7	15	2	5, 6	4	0.9
4	□((p1 ∧ ¬p2 ∧ ◊p2) ⇒ (p0 U p2))		13	66	1	8	—	0.7
	□[→p1|→p2)□p0	[→¬[→p1|→p2)[→¬p0|→)**F** | →)**F**	13	58	2	9, 9	7	16.8
5	□¬p0 ∨ (¬p0 U (□p0 ∨ (p0 U (□¬p0 ∨ (¬p0 U (□p0 ∨ (p0 U □¬p0)))))))		31	86	4	29, 28, 27, 26	17	0.9
	[→p0,→¬p0,→p0,→¬p0|→)□¬p0	[→p0,→¬p0,→p0,→¬p0|→)[→p0|→)**F**	9	20	1	5	—	7.6
6	◊p1 ⇒ ((¬p0 ∧ ¬p1) U (p1 ∨ ((p0 ∧ ¬p1) U (p1 ∨ ((¬p0 ∧ ¬p1) U (p1 ∨ ((p0 ∧ ¬p1) U (p1 ∨ (¬p0 U p1)))))))))		43	134	5	41, 40, 39, 38, 37	23	1.5
	[−|→p1)[→p0,→¬p0,→p0,→¬p0|→) □¬p0	[−|→p1)[→p0,→¬p0,→p0,→¬p0|→) [→p0|→)**F**	12	30	1	3	—	50.1

Table 2: *Comparison of results obtained with LBT* ⬜ *and FBT* ▨ *from interval formulas*

5 Conclusions and Future Work

In this paper, we have presented the FBT tool, which is specially intended to be applied to the automatic verification of systems, using the on-the-fly model checking method and FIL formulas. We have adopted this logic as the specification formalism of our tool for two reasons: firstly, its ability to succinctly express limited temporal contexts in which certain properties must be satisfied; and secondly, its natural, intuitive, graphical representation, which makes the specifications easier to develop and understand.

We have shown class diagrams illustrating the design structure of FBT, and discussed the main characteristics of its implementation. We have also included some experimental results which have been compared with the results obtained with LBT for the same or equivalent formulas. The automata generated by both translators are of a similar complexity, but those produced by FBT are slightly simpler in most of the analyzed cases. As a good specification formalism is the one that describes the most frequently used properties in verification with specifications that are relatively short and not difficult to check in practice, we can conclude that FIL is a good specification formalism and that FBT is a good tool for the efficient translation of its formulas into Büchi automata.

In our most immediate future work, we intend to supply FBT with a graphical editor for GIL formulas so that the specifications can be pictorially inputted rather than in the FIL textual syntax. One editor of this type is GILED [Kutty et al. 1993], which automatically translates the editor-created graphical specifications into the corresponding FIL formulas. The idea is to adapt this editor or to build one of similar characteristics for FBT. We also intend to integrate our translator into an on-the-fly model checking tool. For this purpose, FBT has been designed so that it may be easily incorporated into the model checker of MARIA [Mäkelä 2002], and as we outline in [Gallardo et al. 2004] we are working on the integration of FBT into SPIN [Holzmann 2003], which is one of the most popular finite-state verification tools. Our final aim is to apply our tool to the specification and automatic verification of real-world systems, using interval logic formulas for describing the properties to be checked.

References

[Ben-Ari et al. 1983] Ben-Ari, M., Pnueli, A., Manna, Z.: “The temporal logic of branching time”; Acta Informatica, 20 (1983), 207–226

[Büchi 1962] Büchi, J.R.: “On a Decision Method in Restricted Second-Order Arithmetic”; Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science; Stanford University Press, Stanford (1962), 1–11

[Dillon et al. 1994] Dillon, L.K., Kutty, G., Melliar-Smith, P.M., Moser, L.E. Ramakrishna, Y.S.: “A Graphical Interval Logic for Specifying Concurrent Systems”; ACM Transactions on Software Engineering and Methodology, 3, 2 (1994), 131–165

[Gallardo et al. 2004] Gallardo, M.M., Hornos, M.J., Martínez, J., Merino, P.: “Integration of Interval Logic Specifications into the Model Checker SPIN”; XII Jornadas de Concurrencia y Sistemas Distribuidos, Las Navas del Marqués, Ávila, Spain (2004), 317–322

[Gansner, North 2000] Gansner, E.R., North, S.C.: “An open graph visualization system and its applications to software engineering”; Software: Practice and Experience, 30, 11 (2000), 1203–1233

[Gerth et al. 1995] Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: “Simple On-the-fly Automatic Verification of Linear Temporal Logic”; Proceedings of the 15th International Symposium on Protocol Specification, Testing and Verification, Warsaw, Poland; Chapman & Hall (1995), 3–18

[Holzmann 2003] Holzmann, G.J.: “The SPIN Model Checker: Primer and Reference Manual”; Addison-Wesley, Boston (2003)

[Hornos 2002] Hornos, M.J.: “Tool Design and Implementation”; In: *From Interval Logic Specifications to Property Automata: A Tableau Construction for Application to On-the-fly Model Checking*, Chapter 6, PhD. Thesis, University of Granada (2002), 153–182 (in Spanish)

[Hornos, Capel 2002] Hornos, M.J., Capel, M.I.: “On-the-fly Model Checking from Interval Logic Specifications”; ACM SIGPLAN Notices, 37, 12 (2002), 108–119

[Kutty et al. 1993] Kutty, G., Dillon, L.K., Moser, L.E., Melliar-Smith, P.M., Ramakrishna, Y.S.: “Visual Tools for Temporal Reasoning”; Proceedings of the IEEE Symposium on Visual Languages, Bergen, Norway (1993), 152–159

[Mäkelä 2002] Mäkelä, M.: “Maria: Modular Reachability Analyser for Algebraic System Nets”; Proceedings of the 23rd International Conference on Application and Theory of Petri Nets, Adelaide, Australia; Lecture Notes in Computer Science, 2360, Springer-Verlag (2002), 434–444

[Manna, Pnueli 1992] Manna, Z., Pnueli, A.: “The Temporal Logic of Reactive and Concurrent Systems: Specification”; Springer-Verlag, New York (1992)

[Naur 1960] P. Naur (ed.): “Revised Report on the Algorithmic Language ALGOL 60”; Communications of the ACM, 3, 5, (1960), 299–314

[Ramakrishna et al. 1992] Ramakrishna, Y.S., Dillon, L.K., Moser, L.E., Melliar-Smith, P.M., Kutty, G.: “An Automata-Theoretic Decision Procedure for Future Interval Logic”; Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science, New Delhi, India; Lecture Notes in Computer Science, 652, Springer-Verlag (1992), 51–67

[Ramakrishna et al. 1996] Ramakrishna, Y.S., Dillon, L.K., Moser, L.E., Melliar-Smith, P.M., Kutty, G.: “Interval Logics and Their Decision Procedures. Part I: An Interval Logic”; Theoretical Computer Science, 166, 1–2 (1996), 1–47

[Rumbaugh et al. 1999] Rumbaugh, J., Jacobson, I., Booch, G.: “The Unified Modeling Language Reference Manual”; Addison-Wesley, Reading (1999)

[Stroustrup 1986] Stroustrup, B.: “The C++ Programming Language”; Addison-Wesley, Reading (1986)

[Wolper 1985] Wolper, P.: “The Tableau Method for Temporal Logic: An Overview”; Logique et Analyse, 110–111 (1985), 119–136