Discovering of System's Invariants by Temporal Reasoning

Bolotov, A.

Discovering of System's Invariants by Temporal Reasoning.

Alexander Bolotov
Department of Computer Science,
University of Westminster, London, W1W 6UW, UK
A.Bolotov@wmin.ac.uk

Abstract

We present a technique to handle invariants in the branching-time setting, for the specifications written in the formalism called Branching Normal Form (BNF). The language of BNF was previously used as part of the deductive clausal resolution method for a variety of branching-time logics. We show how this framework can tackle useful periodic properties, or invariants. We emphasise the potential power of this approach to the process of reconfiguration of an adaptive system where preserving invariant properties is essential.

Keywords: software verification, formal specification, temporal reasoning, invariants, branching-time. **Descriptors:** Software and its engineering; Theory of Computation.

1. Introduction

In the area of formal specification/verification the most important properties to tackle are dynamic and an appropriate reasoning framework is given by temporal logics. It allows the representation of a software execution as a sequence of time moments while a possible behaviour of the system is represented by an execution path. Among these dynamic properties some properties represent patterns, i.e. when some event (process, etc) is repeated with some periodicity. We call these properties periodic. Some famous examples of periodic properties are: an invariant property, i.e. the repetition at every moment of time, which corresponds to always operator in temporal logic; some regular pattern properties, such as the repetition at every even moment of time (we abbreviate this as even-moment), Periodic properties represent patterns and play special role in many areas - software integration, scheduling of communication protocols, etc. For example, during the component-based system integration it is crucial to guarantee that periodic properties of components are consistent with each other and conform to the configuration protocol.

We concentrate on adaptive systems which are flexible and long lived systems with the ability to change bahaviour autonomously. Although there has been significant research invested in the analysis of adaptive systems, it has not led to a commonly accepted rigorous definition of what 'changing the system's behaviour' entails. Our analysis will be carried out in the context of a generic architecture based on three layers: the functionality layer ('FL' for short), the management layer ('ML') and the reasoning layer ('RL'). These layers represent respectively the functionality of the system, the configuration and re-configuration management, and, finally, the automated reasoning engine. Thus, within each of the layers we embed a relevant model: of the main functionality of the system within the FL consists of functional components which carry out the required processing tasks. The management layer manages the configuration of these 'components' by providing the means to monitor and reconfigure the functional components. Finally, the reasoning layer communicates with the management layer in order to determine which reconfigurations are plausible and propose reconfigurations to the management layer. Our main focus in this paper will be on the reasoning layer.

Properties and states within our system will be represented through propositions. For example, we may let p stand for 'printer is printing'. Through the use of a suitable logic we may then express the behaviour $\Box p$, meaning that the printer is always printing. Such a behaviour may be termed a 'loop' or 'invariant' of the system.

We use a specific formlaism, BNF (Separated Normal Form for Branching Time Logic). It has been shown that BNF can express simple fairness constraints and their Boolean combinations [2], thus having the same expressiveness as the branching-time logic ECTL$^+$. At the same time the structure of the specifications written in the BNF language enables us to apply to them directly a deductive verification technique, namely, a clausal temporal resolution [2]. This, unlike in the case of commonly used model-checking technique, overcomes the restrictions of finite state systems, providing us with uniform proofs.

2 Verification Framework

Notation. In the rest of the paper, let **T** abbreviate any unary BNF temporal operator and **P** either of path quantifiers. Any formula of the type **PT** is called *a basic* BNF *modality*. Finally, a *literal* is a proposition or its negation.

Indices. The language for indices is based on the set of terms $IND = \{\mathsf{f}, \mathsf{g}, \mathsf{h}, \ldots\}$, where $\mathsf{f}, \mathsf{g}, \mathsf{h}\ldots$ denote constants. Thus, $\mathbf{E}A_{\mathsf{f}}$ means that A holds on some path labelled as f. Note that indices play essential role in the formulation of BNF as they help identifying a specific path context for given formulae. Indices are used to label all formulae of BNF that contain the basic modality $\mathbf{E}\bigcirc$ or $\mathbf{E}\Diamond$. Specifically, the modality $\mathbf{E}\bigcirc$ is associated with BNF step clauses (see below) and, thus, an index ind here simply represents that some formula is evaluated at the successor state of the current state along the path associated with the 'direction' ind - speaking informally, we only take 'one step' along this path. The modality $\mathbf{E}\Diamond$ is associated with evaluating eventualities over longer period of time and thus the label ind for this formula represents that some formula is evaluated at some state along the path which goes from the current state along the 'direction' ind and every successor state along this path, speaking informally, is obtained by taking 'a step' along this 'direction' ind. This corresponds to the limit closure of the path ind and hence the existence of such a path is always guaranteed. Finally, to define the BNF language, we need classically defined constants $\mathbf{true}$ and $\mathbf{false}$, and the operator, $\mathbf{start}$ ('at the initial moment of time').

Definition 1 (Branching Normal Form) *Given $Prop$, a set of atomic propositions, and IND, a countable set of indices,* BNF *has the structure* $\mathbf{A}\Box\left[\bigwedge_i C_i\right]$, *where each of the* clauses C_i *is defined as below where each α_i, β_j or γ is a literal,* $\mathbf{true}$ *or* $\mathbf{false}$ *and* $\mathsf{ind} \in IND$ *is some index.*

$$\mathbf{start} \Rightarrow \bigvee_{j=1}^{k} \beta_j \quad \text{an Initial Clause}$$

$$\bigwedge_{i=1}^{l} \alpha_i \Rightarrow \mathbf{A}\bigcirc\left[\bigvee_{j=1}^{k} \beta_j\right] \quad \text{an } \mathbf{A} \text{ step clause}$$

$$\bigwedge_{i=1}^{l} \alpha_i \Rightarrow \mathbf{E}\bigcirc\left[\bigvee_{j=1}^{k} \beta_j\right]_{\mathsf{ind}} \quad \text{a } \mathbf{E} \text{ step clause}$$

$$\bigwedge_{i=1}^{l} \alpha_i \Rightarrow \mathbf{A}\Diamond\gamma \quad \text{an } \mathbf{A} \text{ sometime clause}$$

$$\bigwedge_{i=1}^{l} \alpha_i \Rightarrow \mathbf{E}\Diamond\gamma_{\mathsf{ind}} \quad \text{a } \mathbf{E} \text{ sometime clause}$$

2.1 Interpretation of BNF

For the interpretation of BNF clauses, we introduce an indexed tree-like model. Let IND be a countable set of indices, S be a set of states, $R \subseteq S \times S$ be a total binary relation over S, and L be an interpretation function $S \longrightarrow 2^{Prop}$, which maps a state $s_i \in S$ to the set of atomic propositions that are true at s_i. Then an indexed model structure $\mathcal{M} = \langle S, R, L, [\mathsf{ind}], s_0\rangle$ where $s_0 \in S$, and $[\mathsf{ind}]$ is a mapping $IND \longrightarrow 2^{S\times S}$ of every index $ind \in IND$ to a successor function $[\mathsf{ind}]$ such that $[\mathsf{ind}]$ is a total functional relation on S, such that for any s_i, s_j, if $s_i, s_j \in [\mathsf{ind}]$ then $s_i, s_j \in R$ (i.e. $[\mathsf{ind}]$ is the determinsaiton of R, and for any $s \in S$, there exists only one state $s' \in S$ satisfying $(s, s') \in [\mathsf{ind}]$.

It is easy to see that the underlying tree model above is an ω-tree.

A state $s_i \in S$ is an ind-successor state of state $s_j \in S \Leftrightarrow (s_i, s_j) \in [\mathsf{ind}]$. An infinite path $\chi^{\mathsf{ind}}_{s_i}$ is an infinite sequence of states $s_i, s_{i+1}, s_{i+2}, \ldots$ such that for every j $(i \leq j)$, we have that $(s_j, s_{j+1}) \in [\mathsf{ind}]$.

Below, we define a relation '$\models$', omitting cases for Booleans and classically defined $\mathbf{true}$ and $\mathbf{false}$. The relation $\models$ evaluates well-formed BNF clauses at a state s_i in a model $\mathcal{M}$.

$\langle \mathcal{M}, s_i\rangle \models \mathbf{start}$ iff $i = 0$

$\langle \mathcal{M}, s_i\rangle \models \mathbf{A}\bigcirc B$ iff *for each* $\mathsf{ind} \in IND$ *and each* $s' \in S$, *if* $(s_i, s') \in [\mathsf{ind}]$ *then* $\langle \mathcal{M}, s'\rangle \models B$

$\langle \mathcal{M}, s_i\rangle \models \mathbf{E}\bigcirc B_{\mathsf{ind}}$ iff *there exist* $\mathsf{ind} \in IND$, *and* $s' \in S$, *such that* $(s_i, s') \in [\mathsf{ind}]$ *and* $\langle \mathcal{M}, s'\rangle \models B$

$\langle \mathcal{M}, s_i\rangle \models \mathbf{A}\Box B$ iff *for each* χ_{s_i} *and* $s_j \in \chi_{s_i}$, if $(i \leq j)$ *then* $\langle \mathcal{M}, s_j\rangle \models B$

$\langle \mathcal{M}, s_i\rangle \models \mathbf{A}\Diamond B$ iff *for each* χ_{s_i}, *there exists* $s_j \in S$, *such that* $(i \leq j)$ *and* $\langle \mathcal{M}, s_j\rangle \models B$

$\langle \mathcal{M}, s_i\rangle \models \mathbf{E}\Diamond B_{\mathsf{ind}}$ iff *there exist* $\chi^{\mathsf{ind}}_{s_i}$ *and* $s_j \in \chi^{\mathsf{ind}}_{s_i}$, *such that* $i \leq j$ *and* $\langle \mathcal{M}, s_j\rangle \models B$

Definition 2 *[Satisfiability, Validity] If $\mathcal{C}$ is in* BNF *then*

- *$\mathcal{C}$ is satisfiable if, and only if, there exists a model $\mathcal{M}$ such that $\langle \mathcal{M}, s_0\rangle \models \mathcal{C}$*
- *$\mathcal{C}$ is valid if, and only if, it is satisfied in every possible model.*

The natural intuition behind BNF is that the initial clauses provide starting conditions while step and sometime clauses constrain the future behaviour. An initial BNF clause, $\mathbf{start} \Rightarrow F$, is understood as "$F$ is satisfied at the initial state of some model $\mathcal{M}$". Any other BNF clause

is interpreted taking also into account that it occurs in the scope of $\mathbf{A}\,\Box$.

3 Temporal Logic Specifications as Metadata

We follow an assumption that the behaviours of individual and composite components in a component system are specified. Our subsequent tasks are to

1. Verify that when components are composed, the resulting component does not possess any contradictions.

2. Verify that invariant properties that we may require for the components are maintained.

The following example is provided in order to illustrate the framework.

The method used to specify components in Fractal and relies on an XML schema, known as ADL [5]. The ADL can be extended in order to provide for the inclusion of a Temporal Logic Specification for each component. For instance,

```
<!ELEMENT behaviour-specification
(comment*, interface*,component*)>
<!ATTLIST behaviour
   variables CDATA #REQUIRED
   TL-SPEC   CDATA #REQUIRED
   invariant CDATA #REQUIRED
>
```

Within each component we associate properties which represent the requirements and behaviour of the system. For clarity of exposition we will remove XML wrappers from now on and just present the variables, the Temporal Logic clauses that describe the behaviour, and the clauses that represent our required invariant. For example, a component responsible for adding two integers may have the following set as part of its metadata [11].

The Variables:

1. a - adder is free
2. b - variables are bound
3. d - computation active
4. e - computation error

and the TL-SPEC can be a full Temporal Logic Specification in either ECTL$^+$ or BNF form. For instance, we can represent the required behaviour by the following set of BNF clauses. Note that the required behaviour could just as easily be specified in ECTL$^+$ and then reduced into the following BNF form required by the algorithms.

1. $start \Rightarrow \neg\mathbf{A}\bigcirc(a \wedge \neg b)$
2. $b \Rightarrow \mathbf{A}\bigcirc\neg a$
3. $\neg a \Rightarrow \mathbf{A}\bigcirc d$
4. $e \Rightarrow \mathbf{E}\bigcirc\neg d$

The ML monitors and maintains the current state of these variables. The behaviour should be read as specifying that

1 At the beginning, on all future computations, at the next moment of time the adder is free and neither of the variables are bound.

2 When the variables are bound then the adder is not free.

3 When the adder is not free then the computation is active, and

4 When there is an error, the computation is not active.

It is the role of the RL (Reasoning Layer) to monitor these states for consistency.

For instance, we may require that

$$b \wedge \neg e => \mathbf{A}\,\Box d$$

should be an invariant of the system. A statement that describes the required invariant property would be

"Whenever variables are bound and there is no computation error, then the computation is active in all futures"

In the following section we describe how we are able to determine this using our Loop search algorithms.

4 Deductive verification for invariants

In [4] a clausal resolution method over the set of BNF clauses was developed. It has been shown that BNF can serve as a normal form for the logics CTL, ECTL and ECTL$^+$ ([2]) (where the corresponding procedures for translating ECTL and ECTL$^+$ formulae into BNF were defined). The core procedure for the application of the resolution method is the discovery of loops. Formally loops are defined as follows:

Definition 3 (Loop in BNF) *A loop in l is a set of merged clauses (possibly labelled) of the form*

$B_0 \Rightarrow \mathbf{P}_0\bigcirc C_{0_{\langle \mathsf{ind}_0\rangle}}, \ldots, B_n \Rightarrow \mathbf{P}_n\bigcirc C_{n_{\langle \mathsf{ind}_n\rangle}}$
where $\mathbf{P}$ *is any of path quantifiers and the following conditions hold* $\models C_i \Rightarrow l$ *and* $\models C_i \Rightarrow \bigvee_{j=0}^{n} B_j$, *for all* $0 \leq i \leq n$.

We will abbreviate a loop introduced in Definition 3 by $(B_0 \vee \ldots \vee B_n) \Rightarrow \mathbf{P}\bigcirc\mathbf{P}\,\Box l_{\langle \mathsf{ind}\rangle}$, where

(i) if for all i, $(0 \leq i \leq n)$, $\mathbf{P}_i$ is the '**A**' path quantifier then $\mathbf{P} = \mathbf{A}$, $\langle \mathsf{ind}\rangle$ is empty, and we have an **A**-loop in l,

(ii) if for all i $(0 \leq i \leq n)$, $\mathbf{P}_i$ there is only one '**E**' quantifier or every $\mathbf{P}_i$ is the '**E**' quantifier with the same

label $\langle \mathsf{ind_i} \rangle$ then we have an **E**-loop in l on the path $\langle \mathsf{ind_i} \rangle$, otherwise

(iii) we have indicated a hidden **E**-loop in l on an infinite path, $\langle \mathsf{ind} \rangle$, combined from $\langle \mathsf{ind_1} \rangle \ldots \langle \mathsf{ind_n} \rangle$.

For a given set, R, of BNF clauses the breadth-first and depth-first loop searching algorithms have been developed. For the purposes of the completeness of our presentation we overview here the depth-first search [1] referring an interested reader to [3] for details of breadth first search.

4.1 Depth First Loop search algorithms

In the descriptions that follow we shall use the term *self loop* in l to signify a loop of the form $B_i \Rightarrow \mathbf{P}\bigcirc(l \wedge B_i)$ for some i. Further, $B_i \Rightarrow \mathbf{P}\bigcirc(l \wedge (B_i \vee Y_1 \vee \ldots \vee Y_n))$, for some n, and for each $Y_i (0 \leq i \leq n)$, Y_i is a conjunction of literals, represents a *partial loop* in l. A partial loop becomes a loop once we have established that each Y_i is also part of a loop in l. Finally, a "leading loop" in l is a sequence of m clauses of the form $B_i \Rightarrow \mathbf{P}\bigcirc(B_{i+1} \wedge l)_{<ind_s>}$ for $0 \leq i < m$, and for m, $B_m \Rightarrow \mathbf{P}\bigcirc\mathbf{P}\Box l$.

The depth first search method we propose is an adaptation of the depth first search method for PLTL by Dixon in [6]. Accordingly, we will adhere to the same terminology wherever possible and highlight the major differences that had to be made to the algorithm to adapt it to the Branching Time case.

We construct a search graph in which edges represent BNF rules and the nodes represent the left hand side of these rules. Nodes are added to the graph depth first if they satisfy the expansion criteria for either backward or forward search in order to find a subgraph where one of the nodes recurs. Backtracking is used if a particular path leads to a "dead-end". The rules governing expansion guarantee that the desired looping occurs.

Graphs in the algorithm are represented as nested lists in which successive entries represent the next node in the graph and where each additional nesting of a level indicates branching, e,g. $[n_0, n_1, [n_2, n_0], [n_4]]$ represents the two paths $[n_0, n_1, n_2, n_0]$ (which is a loop) and $[n_0, n_1, n_4]$. As each entry in this graph is guaranteed to also imply l in the next moment of time (by the expansion rules), this example represents a partial loop in l. It becomes loop in l if we successfully expand to another loop in l from n_4

In the remainder of this section, we present the core algorithms for the Depth-First search **A** algorithm.

The notion behind the DFS search algorithm is to build a search graph such that nodes represent the left or right hand side of clauses within BNF. As the search proceeds a graph is constructed where each edge of the graph represents a transition from one node to another. Thus the algorithm creates chains linking the right hand side of clauses to the left hand side of other clauses or vice-versa, depending on whether backwards or forwards search is being utilised. Backward search is used to start the search procedure and forward search is used when we have reached a partial loop and disjuncts remain to be processed.

4.2 Depth-First A-Search algorithm

When we are looking for an **A**-loop in l we no longer need to consider any BNF clauses of the form

$$\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{E}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)_{<ind_s>}$$

since these would never force a condition to hold on all paths in the graph. Hence, for Step 1, we initialize the $toExpand$ variable only considering clauses of the form $B_k \Rightarrow \mathbf{A}\bigcirc l$. Similarly when expanding subsequent clauses we only consider **A** clauses from BNF when evaluating the expansion criteria for backwards and forwards search.

4.3 Backwards Search Algorithm

During the backwards search we seek clauses in BNF whose right hand side contains a conjunct with l, and also implies the current node.

1. Given the current node n_i, expand the next node n_{i+1} in the search tree by looking for clauses or combinations of clauses of the form

   $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{A}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)$ or

   $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{E}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)_{<ind_s>}$, where $\vdash C_b \Rightarrow n_i$.

2. If such a new rule exists

   a. set the current node n_{i+1} to be $\bigwedge_{a=0}^{g} k_a$ and retain the label $_{<ind_s>}$ if looking for an **E** loop;

   b. if $r > 1$ (i.e. there is more than one disjunct on the right hand side of the rule) structure the search path to represent this and store the disjuncts that have not been matched to the current node in a list for future processing; and

   c. goto step 3;

   otherwise, if no such rule exists

   a. if $i > 0$ (i.e. this is not one of the start nodes) backtrack setting the current node to n_{i-1} and repeat step 1; or

b. if $i = 0$ (i.e. this is one of the start nodes) terminate backwards search and return to the main algorithm.

3. a. if $n_{i+1<ind_s>}$ is already in the search path return to the main algorithm - a loop or partial loop has been detected on $_{<ind_s>}$; otherwise

   b. increment i and continue at step 1.

4.4 Forwards Search Algorithm

The forward search algorithm is invoked after a partial loop has been detected using Backwards Search but disjuncts remain to be processed. The algorithm works by finding clauses in the set BNF such that the current node implies the left hand side of the next node, and the right hand side of the next node also contains l.

1. Given the current node n_i, expand the next node n_{i+1} in the search tree by looking for clauses or combinations of clauses of the form

   $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{A}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)$ or
   $\bigwedge_{a=0}^{g} k_a \Rightarrow \mathbf{E}\bigcirc(\bigvee_{b=0}^{r} C_b \wedge l)_{<ind_s>}$, where $n_i \Rightarrow \bigwedge_{b=0}^{r} k_a$.

2. If a new such rule exists

   a. amend n_i to be $\bigwedge_{a=0}^{g} k_a$;

   b. set the current node n_{i+1} to be C_b and

   c. if $r > 1$ (i.e. there is more than one disjunct on the right hand side of the rule) structure the search path to represent this and store the disjuncts that have not been matched to the current node in a list for future processing; and

   d. goto step 3;

   otherwise, if no such rule exists

   a. if $i > 0$ (i.e. this is not one of the start nodes) backtrack setting the current node to n_{i-1} and repeat step 1; or

   b. if $i = 0$ (i.e. this is one of the start nodes) terminate forwards search and return to the main algorithm.

3. a. if there is a node in the search path n_j (associated with the $_{<ind_s>}$ for a **E** search), such that $n_{i+1} \Rightarrow n_j$ then replace n_{i+1} by n_j in the search path and return to the main algorithm - a loop or partial loop has been detected (on $_{<ind_s>}$ for a **E** search); otherwise

   b. increment i and continue at step 1.

4.5 Example

Example Here we present the **A** search algorithm along with its application to the following example set of BNF clauses which might be derived from composing our component with another component.

1.	$a \Rightarrow \mathbf{E}\bigcirc d_{\langle \mathsf{ind}_1 \rangle}$	5.	$e \Rightarrow \mathbf{E}\bigcirc e_{\langle \mathsf{ind}_2 \rangle}$
2.	$b \Rightarrow \mathbf{A}\bigcirc d$	6.	$(a \wedge c) \Rightarrow \mathbf{A}\bigcirc a$
3.	$c \Rightarrow \mathbf{A}\bigcirc d$	7.	$b \Rightarrow \mathbf{A}\bigcirc b$
4.	$d \Rightarrow \mathbf{A}\bigcirc d$	8.	$d \Rightarrow \mathbf{E}\bigcirc b_{\langle \mathsf{ind}_3 \rangle}$

We will assume that the creator of the component wants the computation to always be active. Hence we are looking for a loop in d.

Before we begin with the algorithm we initialize the following variables: $currentNode = \{\}$, $path = []$, $toExpand = \{\}$, $loopsFound = \{\}$, and since we are looking for a loop in d we set $seekLoopIn = d$,

From Step 1. Find the BNFclauses of the form $B \Rightarrow \mathbf{A}\bigcirc l$, and add these nodes to the "toExpand" variable. Here $toExpand = \{b, c, d, \{b, c\}, \{b, d\}, \{c, d\}, \{b, c, d\}\}$. Note that a was excluded because it can only satisfy loop conditions on a path labelled $< ind_1 >$ and we are searching for an **A** loop. We will now use each of the elements in $toExpand$ as a root for the Depth First expansion.

From Step 2. Set $currentNode = \{b\}$, $path = [(b)]$, and remove the first node from "toExpand", $toExpand = \{c, d, \{b, c\}, \{b, d\}, \{b, c, d\}\}$.

From Step 3. Perform a backwards search from $currentNode$. We are looking for a clause or set of merged clauses of the form $\bigwedge k_j \Rightarrow \mathbf{A}\bigcirc(\bigvee C_b \wedge d)$ such that $C_b \Rightarrow currentNode$. We find that merging clauses 7+2: $b \Rightarrow \mathbf{A}\bigcirc(b \wedge d)$ satisfies these criteria. Now set $currentNode = b$, $path = [(b), (b)]$, and we have found a self-contained loop, $b \Rightarrow \mathbf{A}\bigcirc\mathbf{A}\,\Box d$.

From step 2 (continuation). We note that this loop satisfies the condition that we have found an n_0 as a loop and goto step 6.

From Step 6. Extract the set of nodes from the path constructed and add them to loops found $loopsFound = \{b\}$, and goto step 2.

The algorithm returns $loopsFound = b$ and terminates.

We return having found the loop

$$b => \mathbf{A}\bigcirc\mathbf{A}\,\Box(b \wedge d)$$

5 Discussion

We proposed an extension to the ADL which enabled the embedding of temporal logic specifications. This specification describes the required behaviour of the components. This enabled the verification of the fact that certain invariant properties hold.

It is notable that our specification for the components in the language of the normal form has been used to ease the model checking approach, namely, for the *bounded model checking*. Let us recall that bounded model checking was initially proposed in [8] as one of the solutions to overcome the famous model checking' state explosion problem by introducing a so called 'temporal bound'. The encoding then converts the specification into a Boolean formula which is subsequently used for a SAT checker. In [9] the authors utilised our specification language of the normal form for branching-time logic, BNF, for such the encoding of the specification into a Boolean formula. Their research showed that a SAT solver which 'performs a repetition check and a check for trivial clauses on its input may succeed more quickly than one which tries to apply a decision procedure to the whole clause set. This opens an interesting perspectives of research into the selection of clauses, for example, useful clauses.

References

[1] A. Basukoski and A. Bolotov. Search strategies for resolution in CTL-type logics: Extension and complexity. In *Proceedings of the Time-2005/International Conference on Temporal Logic*, pages 195–197, IEEE, 2005.

[2] A. Bolotov and A. Basukoski. Clausal resolution for extended computation tree logic ECTL$^+$. In *Journal of Annals of Mathematics and Artificial Intelligence, in press*.

[3] A. Bolotov and C. Dixon. Resolution for Branching Time Temporal Logics: Applying the Temporal Resolution Rule. In *Proceedings of the 7th International Conference on Temporal Representation and Reasoning (TIME2000)*, pages 163–172, Cape Breton, Nova Scotia, Canada, 2000. IEEE Computer Society.

[4] A. Bolotov and M. Fisher. A Clausal Resolution Method for CTL Branching Time Temporal Logic. *Journal of Experimental and Theoretical Artificial Intelligence.*, 11:77–93, 1999.

[5] E. Bruneton, T. Coupaye, and J. B. Stefani. Recursive and dynamic software composition with sharing. *Proc. of the 7th International Workshop on Component-Oriented Programming (WCOP2002).*, 2002.

[6] C. Dixon. Search Strategies for Resolution in Temporal Logics. In M. A. McRobbie and J. K. Slaney, editors, *Proceedings of the Thirteenth International Conference on Automated Deduction (CADE)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 672–687, New Brunswick, New Jersey, July/August 1996. Springer.

[7] E. A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics.*, pages 996–1072. Elsevier, 1990.

[8] A. M. Frisch, D. Sheridan and T.Walsh. A Fixpoint Based Encoding for Bounded Model Checking. In *FMCAD* 2002: 238-255.

[9] A. M. Frisch, D. Sheridan and T.Walsh. Comparing SAT Encoding for Model Checking. In *Proceedings of the 7th International Conference Principles and Practice of Constraint Programming - CP 2001, Paphos, Cyprus, November 26 - December 1, 2001*, Lecture Notes in Computer Science, Volume 2239, 2001, 784.

[10] CoreGRID Project. Proposals for a Grid Component Model, Deliverable D.PM.02, February 2006, http://www.coregrid.net.

[11] J. Thiyagalingam and V. Getov. A Metadata Extracting Tool for Software Components in Grid Applications. *IEEE JVA 2006 Symposium on Modern Computing*, 189-196, IEEE CS Press, 2006.

[12] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proc. of the ACM SIGSOFT 12th Symposium on the Foundations of Software Engineering (FSE 2004)*, (Newport Beach, CA, USA), November 2-4, 2004, pp. 23-32.

[13] J. Thiyagalingam, S. Isaiadis, and V. Getov. Towards building a generic services platform: A components-oriented approach. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications.* Springer-Verlag, 2004.

[14] P. Wolper. On the relation of programs and computations to models of temporal logic. In L. Bolc and A. Szałas, editors, *Time and Logic, a computational approach*, chapter 3, pages 131–178. UCL Press Limited, 1995.