LeeTL: LTL with Quantifications over Model Objects

Pouria Mellati
University of Tehran, School of ECE
Tehran, Iran
p.mellati@ut.ac.ir

Ehsan Khamespanah
University of Tehran, School of ECE
Tehran, Iran
e.khamespanah@ut.ac.ir
Reykjavik University, School of Computer Science
Reykjavik, Iceland
ehsan13@ru.is

Ramtin Khosravi
University of Tehran, School of ECE
Tehran, Iran
r.khosravi@ut.ac.ir

ABSTRACT

Dynamic creating of objects and processes as one of the widely used techniques for developing models does not support by the majority of model checking tools. In addition, although there exist few model checking tools which support dynamic creation of model elements, e.g. Spin, they do not provide a property language for presenting the behavioral specifications of dynamically created elements. In this paper, we address this shortage and provide proper support for the model checking of object-based models which contain dynamic object creation. To this aim, we propose LeeTL, a new temporal logic that supports quantifications over model objects. Using LeeTL, it is also possible to traverse objects to access the variables of the objects for defining property formulas. We propose an algorithm for transforming LeeTL formulas to Büchi automata to be able to use the existing model checking tools which support Büchi automata.

CCS CONCEPTS

•Theory of computation → Verification by model checking; Logic and verification; Modal and temporal logics;

KEYWORDS

Model Checking, LeeTL, Dynamic Creation, Büchi Automata, Rebeca

ACM Reference format:
Pouria Mellati, Ehsan Khamespanah, and Ramtin Khosravi. 2017. LeeTL: LTL with Quantifications over Model Objects. In *Proceedings of International SPIN Symposium on Model Checking of Software , Santa Barbara, CA, USA, July 2017 (SPIN'17),* 9 pages.
DOI: 10.1145/3092282.3092294

1 INTRODUCTION

Model checking [2] has been successfully applied to high level modeling languages (e.g., Promela [10], Rebeca[16]) and programming languages (e.g., JPF [8], Erlang [1]). However, there is limited support for model checking of systems with dynamically created elements (processes, actors, etc.). The problem with model checking of such systems is usually with the property specification language which must support referring to dynamically created elements as well as quantification over sets of such elements. An example would be: at every state during the computation, if there is an instance of process `P` with its local variable `i` set to one, eventually, all instances of process `Q` will have their local variables `j` set to zero. The widely used temporal logics for model checking, like Linear Temporal Logic (LTL) and Computational Tree Logic (CTL) [2], do not provide any means for specifying this type of properties.

There are some attempts for extending temporal logics to support analysis of these systems. As one of successful attempts to this aim, Bandera Specification Language (BSL) [3] is proposed for the model checking of Java programs. BSL provides a mechanism for defining universal quantifications over instances of classes. This was deemed necessary by the authors of BSL, since, objects are created dynamically in Java programs. Java programs also can benefit from JPF for model checking [9] of their dynamic behaviors. Claudia et. al. in [4] did the same for the model checking of object oriented models which are developed in Promela. In case of Erlang, although dynamic creation of elements (as the key feature in Erlang programs) is supported by McErlang [9], there is no formalism for specifying properties which address dynamically created elements of programs.

In this paper, we propose Linear entity-enumerating Temporal Logic (LeeTL) as an extension of LTL for specifying properties of object-based systems with dynamic object creation (Section 3). We also show how the LTL model checking algorithm can be adapted for model checking of LeeTL formulas using a transformation to an extension of Büchi automata (Section 4). To this end, we define a semantic framework for the high level modeling languages which can be model checked against LeeTL formulas. This way, the applicability of the proposed algorithm covers a wide range of object-based modeling and programing languages (Section 2).

To illustrate the applicability of this approach, we have used it for model checking of the actor based modeling language, *Rebeca* [14] (Section 6). The case studies carried out demonstrate the usefulness of the proposed logic to analyze class-based languages with dynamic object creation.

2 LEETL COMPATIBILITY SEMANTIC FRAMEWORK (LCSF)

Prior to proceeding to the presentation of the formal syntax and the semantics of LeeTL, we introduce *LeeTL Compatibility Semantic Framework (LCSF)* which is a semantic framework used to describe

SPIN'17, Santa Barbara, CA, USA
© 2017 ACM. 978-1-4503-5077-8/17/07…$15.00
DOI: 10.1145/3092282.3092294

the semantics of class-based modeling languages, offering the primitives needed for LeeTL model checking. In the following, we refer to the dynamically created elements in the language as 'objects'. But in general, the method is not limited to the object-oriented modeling style and the notions of 'class' and 'object' can be replaced by 'process type' and 'process' (encapsulating local variables).

To use LCSF, the language designer must specify the operational semantics of the language in terms of labeled transition systems, and provide four other elements to enable LeeTL model checking. We first describe the labeled transition system (LTS) notation (borrowed from [2]).

For a given model, its underlying LTS is denoted by $TS = (S, s_0, Act, \rightarrow, AP, L)$ where S is the set of states, s_0 is the initial state, Act is the set of action, and $\rightarrow$ is the transition relation. AP is the set of atomic propositions and $L : S \rightarrow 2^{AP}$ is a labeling function associating labels with the states. Note that the interpretation of states, actions, and transitions are determined by the language semantics and is out of scope of LCSF. The set of atomic proposition may include dotted expressions defined as follows.

In a language with dynamically created objects, the object identifiers are not known statically, hence the only way to refer to an object is to navigate through the references among the objects. As common in the programming languages, the expression `mathClass.teacher.age` refers to the age of the teacher of the math class object (whose ID is known statically). If the language supports arrays of object references, more complex expressions may be formed, e.g., `mathClass.student[i].grade`. Since our method is language independent, we assume the language designer specifies the set of all syntactically valid *dotted expressions* in a model, denoted by *DExpr*. Furthermore, since these expressions are to be used in atomic propositions, a function $eval : DExpr \times S \rightarrow Vals$ must be provided to evaluate a given dotted expression in a given state. The result may be an object or a primitive value, or a collection of objects. The set *Vals* includes all possible values of the mentioned types.

With the above definitions, we can use dotted expressions in atomic propositions. For example, we can have $p = person1.age < person2.age$ as an atomic proposition in AP. Then, $p \in L(s)$ if and only if $eval(person1.age, s) < eval(person2.age, s)$.

Additionally, we assume that the set of all class identifiers is denoted by *CID* and the set of all object identifiers is denoted by *OID*. Note that, objects may be removed during a transition from one state to its successors. We assume that once an object is removed from the model in a state, there is no object with the same id in the successor states of that state.

This way, to enable LeeTL model checking, one must provide the semantics of a model in LCSF as the tuple (*TS*, *CID*, *OID*, *DExpr*, *eval*) according to the modeling language semantics.

3 LEETL

Temporal logics, including LTL and CTL, are effectively used for specifying the desired properties of object-based models with static configuration. Atomic propositions of these properties, as the primitive building blocks of temporal logic formulas, are defined using identifiers of objects. In case of systems with dynamically created objects, this approach does not work, as the number of objects and their identifiers cannot be specified prior to the execution of the model. In this section, we propose LeeTL as a property language which considers the requirements of systems with dynamically created objects.

3.1 Syntax of LeeTL

LeeTL formulas over the set AP of atomic propositions are formed according to the following grammar:

$$\begin{aligned} \varphi ::= &\ true \mid a \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U}\, \varphi_2 \mid \\ &\forall v \in DExpr \cdot \varphi \mid \exists v \in DExpr \cdot \varphi \\ DExpr ::= &\ ID \cdot DExpr \mid ID \end{aligned}$$

where $a \in AP$. As mentioned before, LeeTL is an extension of LTL which is enriched by two binding terms $\forall v \in DExpr \cdot \varphi$ and $\exists v \in DExpr \cdot \varphi$. A given formula $\forall v \in expr \cdot \varphi$ or $\exists v \in expr \cdot \varphi$ *binds* all occurrences of v in φ. An LeeTL formula in which all occurrences of all variables are bound is a *closed* LeeTL formula. We only consider closed formulas in this paper. For the sake of simplicity, we also assume that in a nested LeeTL formula, the variables of quantifications are identical. In addition, Even though the syntax above allows variables and dotted expressions to be given to predicates as parameters, the semantics of LeeTL (given in Section 3.2) ensures that only members of *Vals* are actually input to predicates. In other words, in the specification of a predicate, it is not necessary to account for variables and dotted-expressions.

3.2 Semantics of LeeTL

LeeTL formulas as extensions of LTL formulas, stand for properties of paths. This means that a path can either fulfill a LeeTL formula or not. In a given transition system $TS_{\mathcal{M}} = (S_{\mathcal{M}}, s_{\mathcal{M}0}, Act_{\mathcal{M}}, \rightarrow_{\mathcal{M}}, AP_{\mathcal{M}}, L_{\mathcal{M}})$ a *path* is defined as an infinite sequence of states $\langle s_{\mathcal{M}0}, s_{\mathcal{M}1}, s_{\mathcal{M}2}, \cdots \rangle$ where $(s_i, a_i, s_{i+1}) \in \rightarrow_{\mathcal{M}}$. The semantics of a LeeTL formula is defined by the binary relation $\models$ *(satisfies)*, which maps a path (of a model) to a LeeTL formula. Assuming $\pi = \langle s_{\mathcal{M}0}, s_{\mathcal{M}1}, s_{\mathcal{M}2}, \cdots \rangle$ is a path, the following set of rules defines $\models$ relation. Note that in the following rules we use π_i to address sub-path $\langle s_{\mathcal{M}i}, s_{\mathcal{M}i+1}, s_{\mathcal{M}i+2}, \cdots \rangle$ and $AP(s_{\mathcal{M}})$ to address the set of atomic propositions which are assigned to the state $s_{\mathcal{M}}$.

- $\pi \models a \Leftrightarrow a \in AP(s_{\mathcal{M}0})$
- $\pi \models \neg\varphi \Leftrightarrow \neg\,(\pi \models \varphi)$
- $\pi \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \pi \models \varphi_1 \wedge \pi \models \varphi_2$
- $\pi \models \bigcirc\varphi \Leftrightarrow \pi_1 \models \varphi$
- $\pi \models \varphi_1 \mathbf{U}\, \varphi_2 \Leftrightarrow \exists j \in \mathbb{N}_0 \cdot \pi_j \models \varphi_2 \wedge \forall i < j \cdot \pi_i \models \varphi_1$
- $\pi \models \forall v \in DExpr \cdot \varphi \Leftrightarrow \left|eval(s_{\mathcal{M}0}, DExpr)\right| > 0 \wedge \forall\ OID \in eval(DExpr, s_{\mathcal{M}0}) \cdot \pi \models \varphi[OID/v]$
- $\pi \models \exists v \in DExpr \cdot \varphi \Leftrightarrow \left|eval(s_{\mathcal{M}0}, DExpr)\right| > 0 \wedge \exists\ OID \in eval(DExpr, s_{\mathcal{M}0}) \cdot \pi \models \varphi[OID/v]$

In the two last items, the expression $\varphi[OID/v]$ denotes a copy of φ in which all of the occurrences of the variable v are replaced by *OID*.

Having the syntax and semantics of LeeTL, we can define additional equivalences and operators such as $true \equiv \varphi \vee \neg\varphi$, $false \equiv$

$\neg true$, $\Diamond\varphi \equiv true\, U\, \varphi$, $\Box\varphi \equiv \neg\Diamond\neg\varphi$, $\varphi W\psi \equiv (\varphi U\psi) \vee (\Box\varphi)$, and $\varphi R\psi \equiv \neg(\neg\varphi U\neg\psi)$.

4 LEETL TO UTGBA TRANSFORMATION

As mentioned before, LeeTL is designed for the purpose of specification and model checking of systems with dynamically created elements. So, after proposing syntax and semantics of LeeTL formulas, we have to propose a model checking algorithm for these formulas. To this end, we decided to propose a transformation algorithm from LeeTL formulas to büchi automata to use the standard LTL model checking algorithm instead of proposing a new model checking algorithm for LeeTL.

For the standard LTL model checking, at the first step, the büchi automaton of a given LTL formula is generated. Then, the production of the büchi automaton with the state space is performed. Following the same approach, we have to transform a given LeeTL formula φ to an Unaccepting Transition-based Generalized Büchi Automaton (henceforth UTGBA) $UTGBA(\varphi)$, at the first step. The definition of UTGBA is based on the definition of TGBA, given in [7]. Then, produce the production of this UTGBA to a state space. But, unfolding the two binding terms $\forall$ and $\exists$ of a LeeTL formula and transforming them into states and transitions of a UTGBA requires some information about the number of the existing objects in the current state of its corresponding model. So, UTGBAs can not be generated without taking state spaces into account.

To over come this difficulty, our approach works in an on-demand and on-the-fly fashion (described in Section 4.1.2). In our approach, all the states and transitions of UTGBAs are generated lazily (i.e. with respect to the current states of models), except for the initial state. Starting from the initial state of a UTGBA and a state space, production of the initial states are computed then the successor states of both of the initial states are generated. The production of the successor states is performed the same as the initial states. This way, all of the reachable states of a UTGBA can be created according to the states of its corresponding model.

4.1 Preliminaries

4.1.1 UTGBA. A UTGBA is a kind of büchi automaton that its acceptance condition is defined based on its transitions. Formally, a UTGBA is a tuple $utgba = (S, s_0, Act, \rightarrow, U)$ where:

- S is a set of states,
- $s_0 \in S$ is the initial state,
- Act is a set of action labels (defined in an application-specific manner),
- $\rightarrow \subseteq S \times 2^{Act} \times S$ is a transition relation,
- $U \subseteq 2^{\rightarrow}$ is a set of sets of *unaccepting transitions.*

An *execution* $e = \langle w_0, w_1, \cdots \rangle$ of *utgba* is an infinite sequence over alphabet $\rightarrow$. The execution e is *accepting* if and only if for each $u \in U$, infinitely often times there is $j \in \mathbb{N}$ such that $w_j \notin u$. Subsequently, a path $\pi = \langle s_0, s_1, \cdots \rangle$ over alphabet S is *accepted* by *utgba* if and only if there exists an accepting execution $e = \langle (s_0,\, l_0,\, s_1), (s_1,\, l_1,\, s_2), \cdots \rangle$ in *utgba*.

4.1.2 Styles, Entities, and Assumptions. First of all, for sake of simplicity, the proposed algorithms of this section are presented in the object-oriented style: objects with local data variables (i.e. fields) are used. Also, methods may be invoked on objects, in the conventional sense. Additionally, in order to reduce the visual clutter in the pseudo-code, we prefix local variables of objects with "@" character (thus, @var is the short form *this.var* as used in typical object oriented languages), and prefix the global variables in the algorithm using "#" character.

Many entities, such as transitions, are conveniently treated as objects in the algorithm. However, we will only focus on two key object types, namely UbaState and UbaQuantifierState. UbaState represents normal UTGBA states, and contains the following local variables:

unexpandedSuccessor an unprocessed UbaState that acts as a temporary placeholder for the actual successors of this state (which may not have yet been calculated.)

transitions the set of outgoing transitions of this UbaState

toBeDone the set of formulas that must hold at this state, but are yet to be processed

next the set of formulas that must hold at all successors of this state

literals the set of literals that must hold at this state

quantifiers the set of formulas encountered in the processing of this state whose root operators are LeeTL quantifier operators

untilsToSatisfy the set of formulas encountered in the processing of this state whose root operators are U operators

rightsOfUs the set of formulas encountered in the processing of this state that are the right operands of U operators

processed the set of formulas encountered in the processing of this state, used solely in our proof of correctness, and thus removable

Objects of type UbaQuantifierState are corresponding to the on-demand rewriting of LeeTL quantifier operations into normal LTL operations. Except for the *unexpandedSuccessor* and *transitions* fields, UbaQuantifierState contains all of the local variables of UbaState, in addition to the following:

activeQuantifier the quantification operation formula that this state is responsible for rewriting

We also assume that there are two global sets *#storedStates* and *#storedQuantifierStates* which store all processed UbaStates and all finalized UbaQuantifierStates, respectively. Both sets, are initially empty and they are used to prevent the generation of duplicate states. This helps reducing the size of the generated UTGBAs.

For a given LeeTL formula φ, the initial UbaState s_0 is generated and $s_0.next$ is set to $\{\varphi\}$ and s_0 is added to #storedStates. Then, the production of the UbaState s_0 to the initial state of the state space can be computer. Starting from s_0, all of the states of the UTGBA which correspond to φ must be created using *expand* method, as described below.

4.2 *Expand* Method of UbaQuantifierStates

Expand method of UbaQuantifierStates is implemented by rewriting the active quantifier of the state into a form that does not use the active quantifier. This rewriting is performed according to the semantics of LeeTL: if the active quantifier is a universal quantifier, then the rewritten version is a conjunction of formulas, unless the set of involved objects is empty, in which case, the rewritten version is simply "true".

Similarly, the rewritten version of a existential quantifier is either a disjunction of formulas, or simply false in case the set of involved objects is empty.

As a result, a UbaState state is generated that the values of its state variables are the same as that of the UbaQuantifierState and the value of its *toBeDone* variable is set to the rewritten formula. This state is returned as the result of invoking the *expand* method.

4.3 *Expand* Method of UbaState

The expand method, takes one formula out of @toBeDone at a time and processes it, until @toBeDone is empty, at which point the finalization stage of the state begins. The following two subsections describe how formulas in @toBeDone are processed, and how a fully processed UbaState is finalized.

Listing 1: The expand method, used by UbaState

```
method expand(modelState, outputTrans [default=∅]) {
 if (@toBeDone == ∅)
   finalize(modelState, outputTrans)
 else
   processAFormula(modelState, outputTrans)
}
```

Because of the on-the-fly nature of the proposed transformation approach, when a UbaState s is created, fully processed, finalized, and added to #storedStates, its set of outgoing transitions is still not calculated. In which case, a transition to an unprocessed UbaState (which is not in #storedStates) is used as a temporary placeholder for future outwards transitions of s. This, unprocessed state is called the *unexpanded successor* of u. When the transitions method is invoked on u for the first time, the processing of the unexpanded successor is triggered by invoking the expand method on it. The result of the expansion of the unexpanded successor is the set of outgoing transitions of u, which u will store and simply return in response to future invocations of the transitions method.

In order to process a formula φ from @toBeDone, as previously mentioned, we will first remove it from @toBeDone. Then φ is added to @processed, since it will soon be processed. At the next step, if φ is found to be the right operand of any *Until* modality encountered in the algorithm, then φ needs to be added to @rightsOfUntils. This is necessary, since, as will be shown later, in the finalization stage, the @rightsOfUntils and @untilsToSatisfy fields are used in the calculation of unaccepting sets.

How φ is further processed depends on whether it is simply a literal or not, and if not, on the root operator of the formula: If φ is a literal, it is simply added to @literals, and the processing of φ is finished. Otherwise, if $\varphi = \bigcirc\mu$, then it is enough to add μ to @next. Similarly, if the root operator of φ is quantification, φ is just added to @quantifiers.

In case $\varphi = \mu \wedge \psi$, both μ and ψ are added to @toBeDone for further processing. In all of the above cases, we will next re-invoke the expand method on this state, so that either another formula will be processed, or the finalization stage will be entered.

The case when $\varphi = \mu \vee \psi$ is a bit more involved: this UbaState is split into two. Each split is identical to this state. We add μ to the toBeDone of one split, while ψ is added to the toBeDone of the other. Next, the expand method is called on both splits, the result of each of which is a set of transitions. The union of the two sets is returned by this UbaState as the result of the expansion.

If $\varphi = \mu U \psi$, first, φ is added to @untilsToSatisfy. Then, φ is rewritten as $\psi \vee (\mu \wedge \bigcirc\varphi)$ and simply added to @toBeDone. If, on the other hand, $\varphi = \mu R \psi$, it will be enough to rewrite it as $(\psi \wedge \bigcirc\varphi) \vee (\mu \wedge \psi)$ and add the rewritten version to @toBeDone. The pseudo code of this algorithm is depicted in Listing 2.

Listing 2: The processAFormula method, used for UbaStates

```
method processAFormula(modelState, outputTrans) {
  $\varphi$ = take a formula from @toBeDone
  @toBeDone = @toBeDone \ {$\varphi$}
  @processed = @processed ∪ {$\varphi$}

  if ($\varphi$ is the right operand of an until)
    @rightsOfUs = @rightsOfUs ∪ {$\varphi$}

  if ($\varphi$ is a literal)
    @literals = @literals ∪ {$\varphi$}
  else if ($\varphi = \mu \wedge \psi$)
    @toBeDone = @toBeDone ∪ {$\mu$, $\psi$}
  else if ($\varphi = \bigcirc\mu$)
    @next = @next ∪ {$\mu$}
  else if ($\varphi$ is a quantifier formula)
    @quantifiers = @quantifiers ∪ {$\varphi$}
  else if ($\varphi = \mu \vee \psi$) {
    create n1 & n2 as exact copies of this node
    n1.toBeDone = n1.toBeDone ∪ {$\mu$}
    n2.toBeDone = n2.toBeDone ∪ {$\psi$}
    return n2.expand(modelState, n1.expand(modelState,
         outputTrans))
  } else if ($\varphi = \mu U \psi$) {
    @untilsToSatisfy = @untilsToSatisfy ∪ {$\varphi$}
    @toBeDone = @toBeDone ∪ {$(\mu \wedge \bigcirc\varphi) \vee \psi$}
  } else if ($\varphi = \mu R \psi$)
    @toBeDone = @toBeDone ∪ {$(\psi \wedge \bigcirc\varphi) \vee (\mu \wedge \psi)$}

  expand(modelState, outputTrans)
}
```

Once the toBeDone field of a UbaState u is empty, it enters the finalization stage, wherein a new transition will be created, depicted in Listing 3. If there are no formulas in *u.quantifiers*, the destination of the newly created transition will be a UbaState. Otherwise, the destination will be a UbaQuantifierState.

In case *u.quantifiers* is empty, we will add u to #storedStates and use it as the destination of the transition, unless there is already a UbaState u' in #storedStates such that u and u' are equal (two UbaStates are equal if their next fields contain the same formulas), in which case, we will simply use u' as the destination of the transition, without adding u to #storedStates. This ensures that no duplicate states are created. If no u' is found to equal u, it is also necessary to build an unexpanded successor for u. All of the unexpanded successor's fields will be empty, except for its toBeDone field which will be set to $u.next$.

Listing 3: The finalization method, used for UbaStates

```
1  method finalize(modelState, outputTrans) = {
2    if (@quantifiers == ∅) {
3      $u'$ = find this in #storedStates
4      if ($u'$ was found)
5        newTransition = new Transition(labels ⇐ @literals,
             destination ⇐ $u'$)
6      else {
7        @unexpandedSuccssor = new UbaState
8        @unexpandedSuccssor.toBeDone = @next
9
10       #storedStates = #storedStates ∪ {this}
11       newTransition = new Transition(labels ⇐ @literals,
             destination ⇐ this)
12     }
13     unacceptingSets = {unaccepting set of $\mu U \psi$ | $\mu U \psi$ ∈
           @untilsToSatisfy and $\psi$ ∉ @rightsOfUs}
14     foreach unacceptingSet in unacceptingSets
15       unacceptingSet = unacceptingSet ∪ {newTransition}
16   } else {
17     uqState = this state as a UbaQuantifierState
18     uqState.activeQuantifier = an arbitrary member of
           @quantifiers
19     $q'$ = find uqState in #storedQuantifierStates
20     if ($q'$ was found)
21       newTransition = new Transition(labels ⇐ ∅,
             destination ⇐ $q'$)
22     else {
23       storedQuantifierStates = storedQuantifierStates ∪
             {uqState}
24       newTransition = new Transition(labels ⇐ ∅,
             destination ⇐ uqState)
25     }
26   }
27   outputTrans = outputTrans ∪ {newTransition}
28   return outputTrans
29 }
```

Finally, it is necessary to add the transition to the unaccepting sets to which it must belong. Note that for formulas containing *until* modality (henceforth *until* formulas) ever encountered in the algorithm, we consider a corresponding unaccepting set to exist. As shown in the pseudo-code, the set of unaccepting sets this transition will belong to is the set of unaccepting sets corresponding to *until* formulas in u.*untilsToSatisfy* whose right hand operands do not exist in u.*rightsOfUntils*. This ensures that, in the constructed UTGBA, executions in which certain *until* formulas are never satisfied (i.e. the right operand of the *until* modality never occurs) will not be considered accepting executions.

Let us now consider the case when u.*quantifiers* contains formulas. In this case, the destination of the transition will be a UbaQuantifierState. Furthermore, it will not be necessary to include the transition in a set of unaccepting sets or to annotate the transition with a set of literals.

As the first step, a UbaQuantifierState version q of u will be created: all of the fields of q are set to the values of their corresponding fields from u. Then, since q.*quantifiers* might contain more than one quantifier formula, an arbitrary member of q.*quantifiers* is chosen as the active, or main, quantifier of q. As will be shown in Section 4.2, it is this active quantifier formula that will be rewritten when the transitions method is invoked on q.

At this stage, similarly to the case of an empty u.quantifiers, it is necessary to ensure that no q' in #storedQuantifierStates already exists such that q and q' are equal.

Two UbaQuantifier states, such as q and q' are considered equal, if their next, toBeDone, quantifiers, literals, untilsToSatisfy and rightsOfUntils fields are equal. If q' is found, then q is dropped and q' is used as the destination of the transition. Otherwise, q is added to #storedQuantifierStates and used as the destination.

5 PROOF OF CORRECTNESS

The proof in this section is based on [6]. Theorem 5.1 establishes the correctness of the LeeTL to UTGBA translation algorithm.

Theorem 5.1. *UTGBA* $u = (S, D, \Delta, q_0, U)$ *constructed for LeeTL formula* φ *accepts exactly the same paths that satisfy* φ.

Proof. Lemmas 5.5 and 5.10 prove the two directions of this theorem. □

Let $t = (source, labels, dest) \in \Delta$ be a transition, *modelState* is its corresponding state in the state space of a system, and s be the unexpandedSuccessor in whose `finalize` method is created, then, in the following lemmas, $\nabla(t)$ denotes the value of $s.processed$ at the moment s begins the finalize method and is referred to as the *nabla* of t. Also, *labels*(t) denotes *labels*, while *next*(t) denotes *dest.next*. Finally, recall that if $\pi = x_0 x_1 x_2 \ldots$ is a path, then π_i denotes $x_i x_{i+1} x_{i+2} \ldots$.

Lemma 5.2. *If* $e = t_0 t_1 t_2 \ldots$ *is an execution of UTGBA u and* $\mu U \psi \in \nabla(t_0)$, *then one of the following holds:*

(1) $\forall i > 0 : \{\mu, \mu U \psi\} \subseteq \nabla(t_i)$ *and* $\psi \notin \nabla(t_i)$
(2) $\exists j > 0 : \forall\, 0 \le i < j : \psi \in \nabla(t_j)$ *and* $\{\mu, \mu U \psi\} \subseteq \nabla(t_i)$

Proof. Immediately from the algorithm. (Specifically, refer to line 22 of Listing 1.) □

In the above lemma, if e is an accepting execution, then only the second case is possible. The first case, defines an execution in which after $\mu U \psi$ is seen in the nabla of a transition t, ψ is never seen. This causes all transitions occurring after t to be in the unaccepting set $U_{\mu U \psi} \in U$ that corresponds to $\mu U \psi$ (see lines 13 to 15 of listing 3). Thus, based on the definitions in Section 4.1.1, such an execution can not be accepting, and only the second case is possible.

Let transitions of the form (q_0, l, d) of a UTGBA u, where q_0 is the initial state of u, be called the *initial transitions* of u.

Lemma 5.3. *For each initial transition t of the UTGBA created for the LeeTL formula* φ, *we have* $\varphi \in \nabla(t)$.

Proof. Immediately from line 4 of Listing 1. □

Let $\Xi = \{l_0, l_1, \ldots, l_n\}$ be a set of literals, then $\wedge\Xi = l_0 \wedge l_1 \wedge \ldots \wedge l_n$.

Lemma 5.4. *If $e = t_0 t_1 t_2 \ldots$ is an execution of a UTGBA, that accepts $\pi = x_0 x_1 x_2 \ldots$, then $\pi \models \wedge\nabla(t_0)$.*

Proof. Using induction on the size of formulae we prove that for all formulae φ, if $\varphi \in \nabla(t_0)$, then $\pi \models \varphi$:

- In the base case we have $\mu \in \nabla(t_0)$, where μ is a literal. Then, according to the algorithm (line 10 of listing 1), $\mu \in labels(t_0)$. But since π is accepted by e, and by the definitions in section 4.1.1, $\pi \models \mu$.
- If $\mu \wedge \psi \in \nabla(t_0)$, then according to the algorithm (line 12 of listing 1) we have $\{\mu, \psi\} \in \nabla(t_0)$. Then, by the induction hypothesis $\pi \models \mu$ and $\pi \models \psi$, and thus $\pi \models \mu \wedge \psi$. The cases for $\mu \vee \psi$, $\bigcirc\mu \in \nabla(t_0)$ are treated similarly.
- If $\mu U \psi \in \nabla(t_0)$, then as mentioned before, we have:

  $$\exists j > 0 : \forall\, 0 \le i < j : \psi \in \nabla(t_j) \textit{ and } \{\mu, \mu U\psi\} \subseteq \nabla(t_i)$$

  But based on the induction hypothesis $\pi_j \models \psi$ and $\forall\, 0 \le i < j : \pi_i \models \mu$. Therefore, based on the semantic definitions in section 3.2, $\pi \models \mu U \psi$.
- If $\varphi = \forall v \in DExpr \cdot \psi$ and $\varphi \in \nabla(t_0)$, then based on our algorithm, at some point in the creation of t_0, φ has been rewritten as $\varphi\prime$. The translation from φ to $\varphi\prime$ is done in accordance to the semantics given in Section 3: φ' will be either $true$ or $\psi_0 \wedge \psi_1 \wedge \cdots \wedge \psi_n$. In the former case, automatically $\pi \models \varphi\prime$, and in the latter case, by the induction hypothesis we have $\forall i \in \{0, 1, \cdots, n\} : \pi \models \psi_i$. Thus, $\pi \models \varphi\prime$ and since φ and φ' are equivalent, $\pi \models \varphi$.
- The case for $\varphi = \exists v \in DExpr \cdot \psi$ and $\varphi \in \nabla(t_0)$ is similar to the above case. In this case, $\varphi\prime = \mu_0 \vee \mu_1 \vee \cdots \vee \mu_n$. (Note that $\varphi\prime$ can not have been rewritten as false, since π was accepted by e.) Then, according to the algorithm, $\exists i \in \{0, 1, \cdots, n\} : \psi_i \in \nabla(t_0)$. But by the induction hypothesis $\pi \models \psi_i$. Thus, $\pi \models \varphi\prime$ and $\pi \models \varphi$.

□

Lemma 5.5. *If $e = t_0 t_1 t_2 \ldots$ is an execution of UTGBA u constructed for LeeTL formula φ, and accepts a path π, then $\pi \models \varphi$.*

Proof. Based on Lemma 5.3, $\varphi \in \nabla(t_0)$. By Lemma 5.4, $\pi \models \wedge\nabla(t_0)$. As a result, there is $\pi \models \varphi$. □

Let $\pi = x_0 x_1 \ldots$ be a path, then $first(\pi)$ denotes x_0. Additionally, let φ and ψ be two LeeTL formulae, then $\varphi \underset{x}{\longleftrightarrow} \psi$ iff:

$$\forall \pi : (first(\pi) = x \rightarrow (\pi \models \varphi \leftrightarrow \pi \models \psi))$$

Lemma 5.6. *If u is a UbaState or a UbaQuantifierState, let function $conjunctions(u)$ denotes:*

$$\bigwedge u.processed \wedge \bigwedge u.toBeDone \wedge \bigwedge u.quantifiers \wedge \bigcirc \bigwedge u.next$$

Then:

1. *During the execution of the expand method with argument of modelState = x, invoked on UbaState u, when u is split into states u_1 and u_2 (lines 21 to 26 of Listing 1) the following holds:*

   $$conjunctions(u) \underset{x}{\longleftrightarrow} (conjunctions(u_1) \vee conjunctions(u_2))$$

2. *For executions of the continueProcessing method with argument modelState = x in which the UbaState is not split, if the executing UbaState is denoted by u at the start of the method, and denoted by u' exactly before the next call to the expand method is made, then the following holds:*

   $$conjunctions(u) \underset{x}{\longleftrightarrow} conjunctions(u')$$

3. *During replacing the UbaQuantifierState u by the UbaState u' using rewriting rules, the following holds:*

   $$conjunctions(u) \underset{x}{\longleftrightarrow} conjunctions(u')$$

4. *In an invocation of the finalize method with argument of modelState = x, on UbaState u, let t be the (mathematical representation of the) transition eventually created. Then, the following holds:*

   $$conjunctions(u) \underset{x}{\longleftrightarrow} \Big(\bigwedge \nabla(t) \wedge \bigcirc \bigwedge next(t)\Big)$$

Proof. Immediately from Listing 1 and the semantics LeeTL. □

Let $s \in S$ and u be the unexpanded successor of s (as created in lines 7 and 8 of Listing 3). Then, *the descendant transitions of u for model state x* is defined to be $T_s(x)$, i.e. the set of outward transitions of s for model state x.

Lemma 5.7. *If u is an unexpanded successor of a state (i.e. the values of $u.literals$, $u.next$, $u.processed$, and $u.quantifiers$ is set to $\emptyset$), Ξ is the value of $u.toBeDone$, and $t_0, t_1, \cdots, t_n$ are the descendant transitions of u for model state x, then the following holds:*

$$\bigwedge \Xi \underset{x}{\longleftrightarrow} \bigvee_{i \in \{0,1,\cdots,n\}} \Big(\bigwedge \nabla(t_i) \wedge \bigcirc \bigwedge next(t_i)\Big)$$

Additionally, if π is a path, such that $first(\pi) = x$ and $\pi \models \bigwedge \Xi$, then $\exists i \in \{0, 1, \cdots, n\} : \pi \models (\bigwedge \nabla(t_i) \wedge \bigcirc \bigwedge next(t_i))$ such that for each $\mu U \psi$ in $\nabla(t_i)$ with $\pi \models \psi$, ψ is also in $\nabla(t_i)$.

Proof. The main claim is proven by induction on the algorithm using Lemma 5.6, while the additional claim is immediately evident from the algorithm, specifically line 24 of Listing 1, which causes t_i as required to exist. □

Lemma 5.8. *If π is a path and t is a transition in UTGBA u such that $\pi \models (\bigwedge \nabla(t) \wedge \bigcirc \bigwedge next(t))$, then u contains a transition t' following t (i.e. $t.destination = t'.source$) such that $\pi_1 \models (\bigwedge \nabla(t') \wedge \bigcirc \bigwedge next(t'))$. Furthermore, if $\Gamma(t, \pi) = \{\psi | \mu U \psi \in \nabla(t)$ and $\psi \notin \nabla(t)$ and $\pi_1 \models \psi\}$, then, there is specifically a transition t' following t that additionally satisfies that $\Gamma(t, \pi) \subseteq \nabla(t')$.*

Proof. In the execution of the finalize method that results in the creation of $t = (s, l, d)$, the value of $t.destination.next$ is assumed for $t.destination.unexpandedSuccessor.toBeDone$ (line 8 of Listing 3). Since all transitions that follow t for model state $first(\pi_1)$ (i.e. $T_d\ (first(\pi_1))$) are descendant transitions of the state which addressed by $t.destination.unexpandedSuccessor$ for model state

$\mathit{first}(\pi_1)$, then according to Lemma 5.7, a transition following t, as required, exists. □

Lemma 5.9. *If u is the UTGBA constructed for LeeTL property φ, q_0 is u's initial state, and $T_{q_0}(x)$ denotes the set of outward transitions of q_0 for an arbitrary model state x, then the following holds:*

$$\varphi \underset{x}{\longleftrightarrow} \bigvee_{t \in T_{q_0}(x)} \left(\bigwedge \nabla(t) \wedge \bigcirc \bigwedge next(t)\right)$$

Proof. According to the algorithm, $q0.next = \varphi$. Therefore, in the finalization stage of q_0, $q_0.unexpandedSuccessor.toBeDone$ is set to φ. Since $T_{q_0}(x)$ is the set of descendant transitions of $q_0.unexpandedSuccessor$ for model state x, Lemma 5.7 guarantees the correctness of this lemma. □

Lemma 5.10. *If u is the UTGBA constructed for LeeTL property φ, π is a path such that $\pi \models \varphi$, then there is an execution e in u that accepts π.*

Proof. First, $\pi \models \bigvee_{t \in T} (\bigwedge \nabla(t) \wedge \bigcirc \bigwedge next(t))$ is true based on Lemma 5.9, where $T = T_{q_0}\ (\mathit{first}(\pi))$ and q_0 is the initial state of u. Also, $\exists t \in T : \pi \models (\bigwedge \nabla(t) \wedge \bigcirc \bigwedge next(t))$ such that for each $\mu U \psi$ in $\nabla(t)$ with $\pi \models \psi$, ψ is also in $\nabla(t)$. Next, successors of t in e can be constructed by repeatedly choosing successive transitions in u using Lemma 5.8. That is, a transition t_{i+1} following the current transition t_i in u is chosen, such that if π_i satisfies $(\bigwedge \nabla(t_i) \wedge \bigcirc \bigwedge next(t_i))$, then π_{i+1} satisfies formula $(\bigwedge \nabla(t_{i+1}) \wedge \bigcirc \bigwedge next(t_{i+1}))$, and specifically, $\Gamma(t, \pi_i) \subseteq \nabla(t_{i+1})$. In the case where there exists a $\mu U \psi$ in $\nabla(t_i)$ with $\pi_i \nvDash \psi$ and $\pi_{i+1} \nvDash \psi$, since $\pi_i \models \mu U \psi$, there must be a minimal $j > i$ such that $\pi_j \models \psi$. However, according to Lemma 5.2, the $\mu U \psi$ formula is propagated into all successive transitions of t_i, until $\psi \in \nabla(t_j)$. Thus, t_j can be chosen such that $\psi \in \nabla(t_j)$. □

6 EXPERIMENTAL RESULTS

We developed a case study in two different sizes and model checked it against different properties to illustrate how effectively the proposed approach works. Note that based on the discussions in the related work section, we omitted comparison between the result of this work with the other existing model checking toolset as none of them fulfill the requirements of dynamic Rebeca models and LeeTL expression.

The developed case study for the comparisons of this paper is a special implementation of the leader-election problem using the extended version of Rebeca Language. Rebeca, is an operational interpretation of the actor model with formal semantics, supported by model checking tools [16]. Rebeca is designed to bridge the gap between formal methods and software engineering. The formal semantics of Rebeca is a solid basis for its formal verification. Compositional and modular verification, abstraction, symmetry and partial-order reduction have been investigated for verifying Rebeca models. The theory underlying these verification methods is already established and is embodied in verification tools [12, 13, 15, 16].

The implemented leader-election model contains only one type of actor which models the behavior of a node in a network. Different nodes of the model are instantiated from this type. A node has a unique identifier and the goal of the problem is finding a leader node which has the biggest identifier. In this extension of the model, communication among nodes takes place by broadcasting.

Using $p_1 : \forall i \in Node \cdot \forall j \in Node \cdot (i \neq j \wedge i.isLeader) \rightarrow \neg j.isLeader$ we addressed that there is at most one leader in each moment of the system. With the second one, $p_2 : \Box\Diamond\exists n \in Node \cdot n.isLeader$ we make sure that there is at least one leader in the model. So, combining p_1 and p_2 results in having exactly one leader in the model. Using $p_3 : \Box\forall i \in Node \cdot \forall j \in Node(\cdot j.id > i.id \wedge i.isLeader) \rightarrow$
$\Diamond j.isLeader$ we addressed how leader is changed during the execution of the model and by $p_4 : \Box\forall i \in Node \cdot \forall j \in Node(i \neq j \wedge i.isLeader \wedge (\Diamond j.isLeader))$
$\rightarrow j.id > i.id$ we make sure than none of the nodes can push the leader to resign.

We developed *BlueGrass* as a toolset for the model checking of Rebeca models with dynamic creation feature. LeeTL formulas together with the Rebeca models are fed into BlueGrass and the model checking results are reported as its output. The source code of the model is accessible from http://bitbucket.org/pouria_mellati/bluegrassmodelchecker. The result of using BlueGrass for the model checking of the leader-election problem against the four mentioned properties is present in Table 1.

7 RELATED WORKS

We are aware of two works that have previously attempted to enable quantification over model objects in LTL.

Corbett et. al. in [3] propose Bandera Specification Language (BSL) for model checking of Java source codes. BSL provides a mechanism for defining universal quantifications over instances of classes. This was deemed necessary by the authors of BSL, since, objects are created dynamically in Java programs. The support for quantifications in BSL is implemented in two steps (we consider the case where only one quantification variable is used, though more are possible in BSL):

(1) A statement is injected into the constructor of each class, that will, non-deterministically, either do nothing or do the following: flag the quantification variable as bound and bind the variable to the instance that is being created.
(2) A given quantification in the form of $\mathit{forall}[o : Class] \cdot predicate(o)$ is translated to $\neg bound\ W\ (bound \wedge predicate(o))$.

The above steps taken together make up the semantics of universal quantification in BSL. The difference between the work of [3] and LeeTL is in their formula semantics. For example, formula $forall[o : Class] \cdot predicate(o)$ in BSL is interpreted as *for each object o of type Class ever created, $predicate(o)$ must hold at the time of creation of o*; however, a similar formula $\forall o \in Class \cdot predicate(o)$ indicates that *for each object o of type Class that exists at this state, $predicate(o)$ must hold at this state*, based on the semantics of LeeTL. This way, the quantifiers of BSL only hold meaning when used at the very beginnings of formulas (this is also reflected in the syntactic specification of BSL). As a result the quantifiers of BSL are less expressive than those of LeeTL's. For instance, it is impossible to express LeeTL formula $\Diamond\forall p \in Person \cdot\ p.isHappy$ using quantifiers

Table 1: Comparing the size of the state space and time consumption in different case studies

Prop	Nodes' Ids	Time (sec)	Model States	BA States	Product States
p_1	[4, -23, 44]	1 sec	873	2	873
	[4, -23, 44, 100]	1.3 min	360K	2	360K
p_2	[4, -23, 44]	1 sec	873	3	1.2K
	[4, -23, 44, 100]	1.5 min	360K	3	574K
p_3	[4, -23, 44]	1 sec	873	5	1.1K
	[4, -23, 44, 100]	2.3 mins	360K	6	400K
p_4	[4, -23, 44]	2 secs	873	16	2.1K
	[4, -23, 44, 100]	3 mins	360K	25	1.2M

of BSL. However, having less expressive property specification language results in reducing time consumption of the model checking in comparison with the approach of this paper.

Distefano et. al. in [5] proposed a temporal logic that enables the specification of properties concerning the allocation and deallocation of entities and the values to which these entities may point. Inspired by OCL, the logic also supports quantification over entities. They also provided a model checking algorithm for NAllTL. The main draw-back of this work was reporting false counter-examples. The authors, argue that the problem is not necessarily inherent in NAllTL. An advantage of our work over NAllTL is that, with NAllTL, properties need to be checked against a certain type of automata, called HABA. This way, it is necessary for the semantics of modeling languages to be defined in terms of HABA. Even with the required semantics, converting models into HABA is not a straightforward task as certain issues must be avoided by adopting work-arounds including so called "duplication" and "streching". Furthermore, the model checking algorithm needs to be adjusted to support HABA and NAllTL. In contrast, the only requirement imposed by our work is the ability of performing query over the set of active objects of states.

A major strength of NAllTL and HABA is that in some cases HABA can be used for modeling of systems in which an infinite number of objects, using finite automata. This way, infinite state space of these types of systems can be model checked using finite automata. Our work does not support this feature.

In addition to these works, there are some attempts at using LTL like properties for programs and models with dynamic object creation. Iosif and Sisto in [11] proposed a notation for property specification in Java codes. Although their approach is developed for Java and Java supports dynamic object creation, their proposed syntax and semantics does not support object creations. Instead, they proposed writing precondition/postcondition like properties for methods of objects to examine properties of created objects.

There is also work on enriching LTL with existential and universal variable quantifiers, proposed by Song and Wu in [17]. From the syntax point of view, this work is very close to LeeTL. However, authors addressed variables with infinite values using existential and universal variable quantifiers not the objects of the model, as we did in LeeTL. Using the same approach, Ziemann et. al. in [18] combined OCL and CTL to propose a new specification language for object oriented programs. Although their approach covers a wide range of property specification requirements, they did not propose a model checking algorithm for it.

8 CONCLUSION

In this paper, we propose LeeTL, a new temporal logic that supports quantifications over model objects. Currently, support for both of the modeling and property specification of dynamic object-based systems, i.e. those in which, during the execution of the system (dynamically), new objects are created or the web of relationships between actors is manipulated, is less than satisfactory. Using LeeTL, it is also possible to traverse the web of references between objects to access to specific variables during property specification. To this end, we defined a semantic framework for the state spaces which must be supported by them to be able to model checked against LeeTL formulas. Given an LeeTL formula, we showed that who Unaccepting Transition-based Generalized Büchi Automaton is created and how the existing büchi automata based model checking toolsets is used to support LeeTL model checking.

We also extended the model checking toolset of Rebeca to support LeeTL to illustrate the applicability of this approach. To this end, we extended Rebeca Language to support dynamic actor creation and we implemented some case studies using the new features. As a result, beside providing support for realtime and probabilistic models, a modeler can use Rebeca family toolset to develop his actor-based models which have dynamic behaviors.

ACKNOWLEDGMENTS

The work on this paper has been supported by the project "Self-Adaptive Actors: SEADA" (nr. 163205-051) of the Icelandic Research Fund.

REFERENCES

[1] Joe Armstrong. 1997. The development of Erlang. *SIGPLAN Not.* 32 (August 1997), 196–203. Issue 8. DOI: http://dx.doi.org/10.1145/258949.258967

[2] Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking.* MIT Press. I–XVII, 1–975 pages.

[3] James C Corbett, Matthew B Dwyer, John Hatcliff, and others. 2002. Expressing checkable properties of dynamic systems: the Bandera Specification Language. *International Journal on Software Tools for Technology Transfer* 4, 1 (2002), 34–56.

[4] Claudio Demartini, Radu Iosif, and Riccardo Sisto. 1999. dSPIN: A Dynamic Extension of SPIN. In *Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Trento, Italy, July 5, 1999, Toulouse, France, September 21 and 24 1999, Proceedings (Lecture Notes in Computer Science)*, Dennis Dams, Rob Gerth, Stefan Leue, and Mieke Massink (Eds.), Vol. 1680. Springer, 261–276. DOI: http://dx.doi.org/10.1007/3-540-48234-2_20

[5] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. 2005. Who is pointing when to whom? In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science.* Springer Berlin Heidelberg, 250–262.

[6] Rob Gerth, Doron Peled, Moshe Y Vardi, and Pierre Wolper. 1995. Simple on-the-fly automatic verification of linear temporal logic. In *Proceedings of the Fifteenth IFIP WG6. 1 International Symposium on Protocol Specification, Testing and Verification.* IFIP.

[7] Dimitra Giannakopoulou and Flavio Lerda. 2002. From states to transitions: Improving translation of LTL formulae to Büchi automata. In *Formal Techniques for Networked and Distributed SytemsfiFORTE 2002*. Springer, 308–326.

[8] Klaus Havelund and Grigore Rosu. 2004. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design* 24, 2 (2004), 189–215.

[9] Klaus Havelund and Grigore Rosu. 2004. An Overview of the Runtime Verification Tool Java PathExplorer. *Formal Methods in System Design* 24, 2 (2004), 189–215. DOI:http://dx.doi.org/10.1023/B:FORM.0000017721.39909.4b

[10] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Software Eng.* 23, 5 (1997), 279–295. DOI:http://dx.doi.org/10.1109/32.588521

[11] Radu Iosif and Riccardo Sisto. 2003. Temporal logic properties of Java objects. *Journal of Systems and Software* 68, 3 (2003), 243–251. DOI:http://dx.doi.org/10.1016/S0164-1212(03)00062-1

[12] Mohammad Mahdi Jaghoori, Marjan Sirjani, Mohammad Reza Mousavi, Ehsan Khamespanah, and Ali Movaghar. 2010. Symmetry and Partial Order Reduction Techniques in Model Checking Rebeca. *Acta Inf.* 47, 1 (2010), 33–66.

[13] Hamideh Sabouri and Marjan Sirjani. 2008. Slicing-Based Reductions for Rebeca. In *Proceedings of FACS 2008*. ENTCS.

[14] Marjan Sirjani. 2007. Rebeca: Theory, applications, and tools. In *Formal Methods for Components and Objects*. Springer, 102–126.

[15] Marjan Sirjani, Frank S. de Boer, and Ali Movaghar-Rahimabadi. 2005. Modular Verification of a Component-Based Actor Language. *J. UCS* 11, 10 (2005), 1695–1717.

[16] Marjan Sirjani and Mohammad Mahdi Jaghoori. 2011. Ten Years of Analyzing Actors: Rebeca Experience. In *Formal Modeling: Actors, Open Systems, Biological Systems*. 20–56.

[17] Fu Song and Zhilin Wu. 2016. On temporal logics with data variable quantifications: Decidability and complexity. *Inf. Comput.* 251 (2016), 104–139. DOI:http://dx.doi.org/10.1016/j.ic.2016.08.002

[18] Paul Ziemann and Martin Gogolla. 2003. OCL Extended with Temporal Logic. In *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers (Lecture Notes in Computer Science)*, Manfred Broy and Alexandre V. Zamulin (Eds.), Vol. 2890. Springer, 351–357. DOI:http://dx.doi.org/10.1007/978-3-540-39866-0_35