# Extracting and Verifying Viewpoints Models in Multitask Applications

Selma Azaiez, Belgacem Ben Hedia, Vincent David
CEA, LIST, Embedded Real Time Systems Laboratory,
Point Courrier 94, Gif-sur-Yvette, F-91191 France
Email: {selma.azaiez,belgacem.ben-hedia,vincent.david}@cea.fr

*Abstract*—**Static analyzers are most of the time dedicated to checking runtime errors in sequential programs or are specific to one particular property in the multitasking domain such as deadlock detection. However, the safety of multitask and realtime applications relies on several properties (e.g., absence of deadlock, atomicity, respecting the temporal constraints, etc.). Verification of each property requires a specific abstract model. In this paper, we introduce a generic pattern-based method allowing automatic extraction of viewpoint models appropriate for verification of various properties. Each property is defined by a property analysis pattern specifying the algorithm for its verification (steps are defined to specify needed viewpoint models). The extraction of each viewpoint model is described within a dedicated model extraction pattern. Property analysis patterns and model extraction ones are the main achievement of our work. By introducing these patterns, our method allows harmonizing the validation process and capitalizing the knowhow by explicitly defining the verification and transformation processes.**

***Keywords- multitask applications; semantic-based static analysis; property verification; property analysis pattern; model extraction pattern.***

## I. INTRODUCTION

To validate multitask and real-time systems, developers or validators in independent certification authorities have to use different tools based on different methods (e.g., static analysis [5] or model checking [3]). Each tool is dedicated to a specific class of properties or even to a specific stage in the development process [15][16]. For instance, model checking tools are usually used to validate system specifications (expressed using dedicated formal languages); which does not really reflect what is really implemented. In the other hand, the safety of multitask and real-time applications relies on the satisfaction of several properties such as deadlock freedom, atomicity, respecting the temporal constraints, etc. The verification of each property is based on a different viewpoint model. For instance, models focusing on locks are necessary to detect deadlocks; models focusing on shared memory – to check atomicity. Using such different techniques and tools complicate the validation process and a high expertise is required for each type of property.

In this paper, we propose an approach targeting to harmonize validation process for multitask and realtime systems. The approach is applicable to check correctness in these systems from their source code. It is based on the extraction of different viewpoint models driven by the properties to verify. It uses property analysis and model extraction patterns. A property analysis pattern defines the algorithm for determining which models have to be extracted in order to verify the property. The extraction process of each model is described within a model extraction pattern.

The paper is structured as follows. In Section II, we provide an overview of our approach. In Section III, we explain the semantic annotation. Then, in Sections IV and V, we introduce property analysis and model extraction patterns, respectively. Finally, in Section VI we illustrate our approach on a simple example.

## II. RELATED WORK AND APPROACH OVERVIEW

To address our topic, we study different types of validation and verification techniques. We first look at techniques dealing with source code analysis: static analysis [5] and reverse engineering [19]. Then, we study model checking and theorem proving techniques that are suitable for verifying multitasking and realtime systems.

Most static analysis tools were developed for detecting numerical software bugs in sequential programs [15] (e.g., buffer overflows or underflows, null pointer references, etc.). Some examples of such analyzers are ASTREE [10], CAVEAT [11], PolySpace[12], Coverity [13], etc. Other existing tools are more suitable for our context (i.e., multitask realtime systems) but are specific to particular types of properties (e.g., deadlock freedom or race condition detection) [14][17][18]. In reverse engineering approaches, some tools [20] are only focusing on generating structural models such as UML diagram class or function dependencies while others are based on model checking techniques [21][22] but do not address concurrency issues. Hence, in both areas limited concurrency and realtime issues are addressed. By contrast, model checking and theorem proving techniques [4][6][7][8][9] focuses on safety properties in multitask realtime systems. However, verifications are performed on systems specifications which make them suitable for top-down approaches where developers need to be sure that their programs are correct by construction. In our context, a bottom-up approach is needed where different viewpoint models can be extracted from the source code driven by the properties to verify.

The method we propose allows bridging the gap between techniques such as static analysis that checks source code and

techniques such as model checking and theorem proving that uses more abstract models. It is based on three main levels. In the first level, source code is parsed and the AST (Abstract Syntax Tree) is produced. In the second level, AST nodes are annotated. The annotation allows capturing the semantics of specific real-time multitasking APIs objects (such as those provided by POSIX [1], OSEK VDX[2], etc.). A formalism (introduced in Section III) is used for this purpose. During annotation phase, nodes using multitasking elements (primitives or variables) are identified and they are assigned with annotations provided by the semantic description.

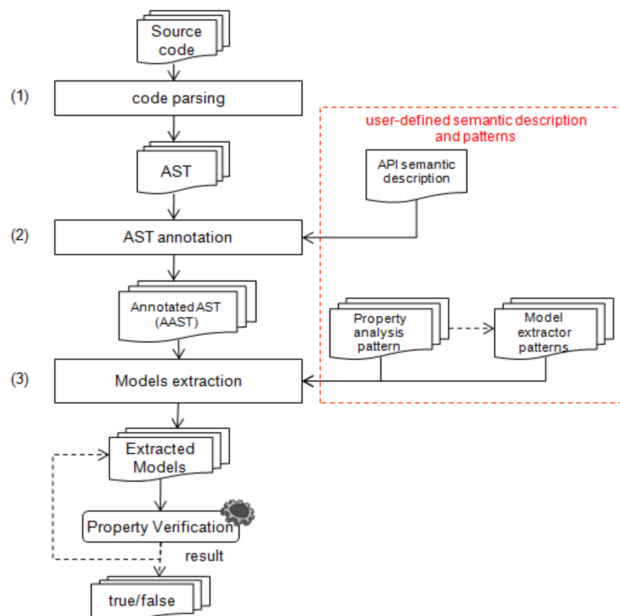


Figure 1. Layers of analysis

In the third level, models are extracted and properties are verified according to user-defined patterns. *Properties analysis patterns* specify within a *checking process* which models are necessary to perform the verification.

*Model extraction patterns* define *transformation rules* according to *pre-conditions* describing initial model configuration and *post-conditions* describing configuration of output model. Once models are extracted the verification of the property is performed. The result of the verification process can either be a Boolean value or an extracted model.

## III. Semantic Annotation

Multitasking concepts are usually implemented within API(s) that include a set of functionalities, data types, data structures, and protocols aiming to facilitate access to resources or services. The use of API(s) elements in the source code is identified during AST annotation phase according to the provided semantic description. In this section, we introduce the formalism that allows capturing API semantics. Then, we introduce features that facilitate models extraction from the AAST (Annotated Abstract Syntax Tree).

### A. Semantic description

We introduce a set of annotations in order to capture multitask API semantics, based on two main concepts:

- Semantic primitives: functions introduced by the API;
- Semantic variables: constraints on the type and values of the parameters and values returned by the primitive call.

#### 1) Multitask primitives

Multitask primitives are classified upon their general semantics:

- Task management: task creation, destruction, sleep and awakening;
- Critical sections management: locks acquisition and release;
- Communication mechanisms: creation or destruction of message-passing mechanisms or shared memories, sending and receiving of messages.

A multitasking primitive is described as follows:

$$P = (id,\ \varphi(parami),\ \varphi(res),\ sem\text{-}role)$$

where:

- `id` : is the identifier of the primitive (e.g., fork);
- $\varphi(param_i)$: value and type constraints having to be respected by primitive parameters;
- $\varphi(res)$: value and type constraints that have to be respected by the primitive return value;
- `sem-role`: annotation expressing the primitive role (e.g., *CREATE-TASK*, *TAKE-LOCK*, *RELEASE-LOCK*, *SLEEP*, etc.).

#### 2) Semantic variables

Semantic variables are the parameters or return values of a primitive. They are defined as following:

$$V = (id,\ \varphi(type),\ range)$$

where:

- `id`: is an identifier (used in semantic description of the primitive);
- $\varphi(type)$: are type constraints;
- `range`: is the range of acceptable values or a constant.

### B. AAST node structure

During parsing phase, the program is tokenized then, the AST is generated. A node in the AST is associated to each word in the source code. Additional nodes are added that we call branch-node and that inform about the syntactical nature of the branch (e.g., statement, function call, loop body, else body etc.). A token is associated to each node to inform about the node nature (e.g., FUNC-CALL, END-LOOP, etc.).

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-168-7 

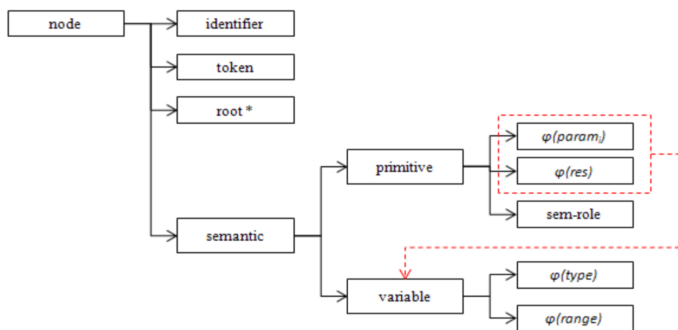


Figure 2. The structure of the annotated AST node

During annotation phase, AST nodes conserve information about syntactical structures. This information is augmented by semantic information mentioned in the previous Sections. After semantic annotation phase, AST nodes have the structure shown in Figure 2.

### C. From AAST to model extraction

Model extraction consists in applying a set of transformation rules to extract a new model configuration from one or several initial ones. To be able to define such rules, original and target models have to follow a common abstract definition. In this subsection, we introduce abstract definition of models as well as a navigation feature that will facilitate the definition of transformation rules.

#### 1) Abstract definition of models

A common representation to all models used in our approach (i.e., AAST, control graphs, synchronization and communication graphs) is graph representation. For all these representation, we adopt a generic definition stating that a graph is a quadruple $G(S,\ T,\ s_0,\ s_f)$ where

- $S$: is a set of nodes;
- $T$: is a set of transitions which can be labeled or not;
- $s_0 \subset S$: is the initial node;
- $s_e \subset S$: is the set of final nodes.

#### 2) Navigation formalism

During patterns specification, one can need to select specific model elements (nodes or transition sets) or to test nodes according to their identifiers, tokens, branches to which they belong or their semantic annotation. To this purpose, we introduce the operator "`::`" that allows such navigation. For instance, the expression G::S states that we refer to the set of nodes S in the graph G.

In Figure 3, we provide an example written in C that creates two tasks by using the `fork` primitive, provided by POSIX [1]. Tasks are synchronized using a producer/consumer process (for the sake of clarity, we suppose that P and V are lock acquisition primitives provided by the platform). In the corresponding annotated AST, all nodes calling a semantic primitive are red while nodes corresponding to semantic values are light gray.

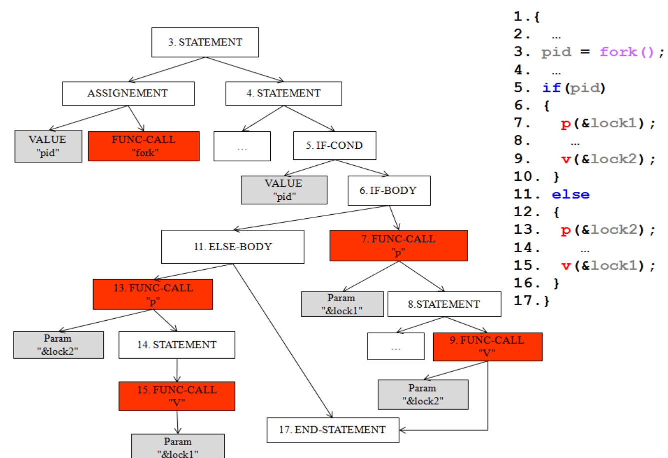


Figure 3. Annotated AST

To select a node calling a primitive with the semantic role `CREATE-TASK`, the following expression is used (where `node ∈ AST`).

```
node::semantic::primitive::sem-role==
CREATE-TASK
```

In order to check whether the returned value of the fork is tested, we use the following expression:

```
node::semantic::variable::type == PID-T ∧
node::root::token == IF-BODY
```

The first expression selects a node corresponding to a semantic variables having `PID-T` type. The second expression tests whether this node is in conditional branch which means that the token corresponding to the root of the current node is an `IF-BODY`.

## IV. SPECIFICATION OF PROPERTIES

Once the AST is annotated, verification of properties can be performed. Each property is described using a pattern provided by Table I.

TABLE I. A PROPERTY ANALYSIS PATTERN

| Identifier | *Property identifier* |
|---|---|
| Checking process | *Defines the steps of the property verification process. These steps can comprise extraction of various models.* |
| Property Specification | *Specification of the property* |

Several models can be used to check a single property. Steps of the checking process are specified by using the following formalism:

```
step-id: from [quantifier]{input models}
         extract[quantifier]{output model}
         according to{model extractor
                     pattern}
step-id: verify {prop-id} on {model}
```

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-168-7 

Property specification is a logical expression that can be stated using first order logic, CTL, LTL, or other specific formalism and that can be verified upon the last extracted model.

## V. MODEL EXTRACTOR PATTERNS AND TRANSFORMATION RULES

When stating the checking process within the property analysis pattern, steps refer to model extraction patterns. These patterns define how to build abstract models according to the API semantics. They have the format described in Table II.

TABLE II. MODEL EXTRACTOR PATTERN

| | |
|---|---|
| Input Models | *Input models from which output models will be extracted* |
| Output Model | *output model can be referred here, several instances can be extracted.* |
| Model Transformation Rules | *Model building rule* |

A model extraction is based on transformation rules. We introduce how to state the method for specifying pre-conditions and post-conditions constraining initial configurations in input models and resulting configurations in output models respectively.

### A. Models transformation rules

Model transformation rules allow deriving a new graph configuration from one or several initial ones.

$$\{G_0, \dots, G_n\} \Rightarrow G_d$$

Transformation rules are based on three elements introduced in Table III.

TABLE III. TRANSFORMATION RULES

| | |
|---|---|
| Pre-conditions | *Set of configuration rules that are respected by input models* |
| Nodes building rules | *Algorithm for building sets S, $s_0$ and $s_e$* |
| Transitions building rules | *Algorithm for building the set T* |

### B. Pre-conditions

In the pre-condition section, the user will define a set of nodes from which the output model will be derived. Pre-conditions are expressed using the following formalism:

`p-id : { nodes: input-graph | φ(nodes)}`

`p-id` describes a precondition φ that has to be respected by the set `nodes` ⊆ input-graph ("|" means "such that").
`p-id` is the identifier of the pre-condition. A pre-condition can state, for instance, "`There exists at least one node that is a fork call`" which is expressed as follows:

```
p1: {fork-node ∈ AST |
    (fork-node::identifier==fork)∧
    (fork-node!=0)}
```

### C. Nodes building rules

Nodes building rules define `S`, $s_0$ and $s_e$ of output models according to provided pre-conditions. They are considered as implication rules expressed as follows:

`{precondition}→{{graph}::set=building-rule}`

"→" means "implies". Building rules are expressed by using one of the following propositions ($n_i$, $n_j$ ∈ `{graph}::S` and one of expressions between brackets or even both):

- ***include all[from** $n_i$ **until** $n_j$**][such that** φ]: this rule includes into the specified set all the nodes of the subset specified by the optional expression [**from** $n_0$ **until** $n_i$] or [**such that** φ];
- ***exclude** all[**from** $n_0$ until $n_i$][**such that** φ]*: exclude from the set S nodes of the subset specified by the optional expression [**from** $n_0$ **until** $n_i$] or [**such the** φ];
- ***build** nodes **according to (f):*** build a node with a new format generated by the function *f* (e.g., from control graph, build a node with task-id).

### D. Transition rules

Transition rules define the algorithm for connecting nodes to each other in the output model $G_d$ according to their configuration in the initial model $\{G_0, \dots, G_n\}$. Transition rules are also expressed using:

- Pre-condition: we assume that there exists one or several element $s_i \in S$ that have their projection $s_d \in S_d$, pre-conditions introduce properties that have to be respected by $s_i$ in the initial model $G$;
- Post-condition: define the type of links that connect $S_d$ nodes in $G_d$.

Transition rules are expressed as follows:

`{precondition}→{connection-rule}`

Where connection-rule have the following format (the fourth optional parameter specifies the transition label):

$$\texttt{connection}(G_d::s_1, G_d::s_2, \texttt{type}, [l])$$

## VI. APPLICATION

We illustrate our methodology on the simple example provided in Section III.C.2). Let us suppose that we want to check whether locks are correctly released after their acquisition. We will provide the property pattern analyzer corresponding to this property. Then, we provide model extraction patterns used in the verification process.

### A. *CORRECT-LOCKS-USE property analysis pattern*

To check whether acquired locks are always released, we need to first extract control graphs in order to determine which task is using which lock. After that, a lock-use-graph is extracted where nodes represent tasks. These nodes are connected by transitions labeled by lock operations (cf. Figure 4).

The property consists of verifying that each node, which is a source of an arc labeled with `ACQ-LOCK` on a lock, has an entrant arc labeled with `RLS-LOCK` on the same lock.

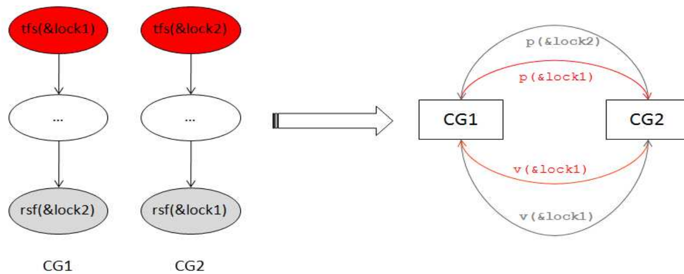


Figure 4. Lock-use-graph extraction

The pattern corresponding to this property is provided in the Table IV.

TABLE IV. CORRECT-LOCKS-USE ANALYSER PROPERTY PATTERN

| Identifier | `CORRECT-LOCKS-USE` |
|---|---|
| Checking Process | `step1 : from {AAST}`<br>`extract all {CG:CONTROL-GRAPH}`<br>`according to{CONTROL-GRAPH-EXTRACTOR-PATTERN}`<br>`step2 : from all {CG}`<br>`extract all {LG:LOCK-USE-GRAPH}`<br>`according to{LOCK-USE-GRAPH-EXTRACTOR-PATTERN}`<br>`step3 : verify {CORRECT-LOCKS-USE}`<br>`on{LG}` |
| Property Specification | `{∀n ∊ LG::S}`<br>`if`<br>`{∃t1 ∊ LG::T \|`<br>`((t1::org==n) ∧`<br>`(t1::label::sem==ACQ-LOCK))}`<br>`then`<br>`{∃t2 ∊ LG::T \|`<br>`((t2::dest==n) ∧`<br>`(t2::label::sem==RLS-LOCK)∧`<br>`(t1::label::param==t2::label::sem))}`<br>`else`<br>`Error` |

### B. *Model extraction patterns*

The property specification pattern refers to two model extraction patterns that are described in the following subsections.

#### 1) *Control graph extraction*

In this example, `fork` primitive is used to create tasks. *`Fork`* is provided by *`POSIX`* and allows the duplication of the current process. *`Fork`* does not take any parameters and returns either 0 (for the child process) or the PID value of the child process (for the parent process). F`ork` can be used differently within a conditional expression or not (cf. Figure 5) and this influences the extraction of the control graph.

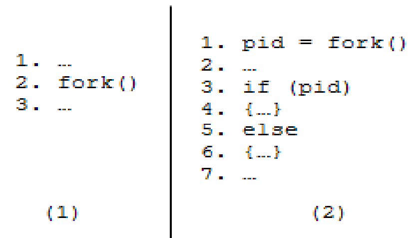


Figure 5. fork writing styles

The first writing style (1) implies that both created tasks have the same behavior and start on the instruction following the call to `fork` (i.e., the instruction in line 3) until the end of the program. In the second case, the behavior of both tasks starts by instructions following `fork` (e.g., instruction in line 2). However, according to the conditional expression that is testing the *`PID`* value, the control graph of the child task continues in else block (i.e., line 6) while the behavior of the parent task is defined by the if block (i.e., line 4). Then, both tasks behavior continues until the end of the program (i.e., line 7).

To specify such semantics, the corresponding pattern described in Table II is expressed as follows:

1. if there exists a conditional expression testing the returned `PID` value; then control graphs `CG1` and `CG2` are created where (1) $S_0$ refers to the statement following the call to `fork`, (2) $S_e$ points towards the end of the program, (3) S includes $S_0$, $S_e$ and all nodes following $S_0$ except those included in `IF_BODY` for `CG1` and those included in `ELSE_EXPR` for `CG2`;
2. if the return value of the `fork` is not tested, both control graphs include all nodes between $S_0$ and $S_e$.

A simplified control graph extractor pattern is provided in Table V.

TABLE V. FORK CONTROL GRAPH EXTRACTION PATTERN

| **Identifier** | `FORK-CONTROL-GRAPH-EXTRACTOR` |
|---|---|
| **Input** | `AAST` |
| **Output** | `CG1, CG2 : CONTROL-GRAPH` |
| **Pre-condition** | |

```
-- there exists in the AST at least one     --
-- with fork identifier and one node        --
-- with END-OF-PROGRAM node                 --
p1:{fork-node, end-node ∊ AAST |
      (fork-node::identifier==fork) ∧
      (end-node::token==END-OF-PROGRAM)}

--  return value of fork is tested          --
p2:{cond, fork-if-body, fork-if-end ∊ AAST |
     (cond::token={VALUE,FUNC-CALL} ∧
      ((cond::semantic::VAR::type=PID-T) ∨
       (cond::identifier==fork))    ∧
      ((fork-if-body::token==IF-BODY-EGIN(cond))
         ∧
       (fork-if-end::token==IF-BODY-END(cond))}
```

```
-- pre-condition stated to check whether   --
-- else exists                             --
p3:{fork-else-body, fork-end-body ∈ AAST |
    ((p2::cond)≠ 0 ∧
     (fork-else-body::token==ELSE-BODY-
BEGIN(cond))
      ∧
     (fork-else-end::token==ELSE-BODY-END(cond))}}
```

**Node building rules**

```
{p1 ∧ p2}→{{CG1, CG2}::S0 = NEXT(fork-node)
          ∧
         {CG1, CG2}::Se = END-OF-PROGRAM}

-- both graphs have the same behavior      --
{p1 ∧ ¬p2∧ ¬p3}→{{CG1, CG2}::S =
                include all from S0 until Se}

-- graphs have different behavior          --
{p1 ∧  p2 ∧ p3}→
  {CG1::S=include all from S0 until Se
            ∧
   CG1::S=exclude all from fork-else-body
                      until fork-else-end
            ∧
   CG2::S=include all from S0 until Se
            ∧
   CG2::S=exclude all from fork-if-body
                      until fork-if-end }
```

**Transition building rules**

```
{ ∃ {n1, n2} ∈ AAST, {gn1, gn2} ∈ CG::S
  | gn1=proj(n1), gn2=proj(n2)  ∧ next(n1,n2)}
→
{ connect(gn1, gn2, direct-transaction) }
```

### 2) *Extraction of the lock use graph*

The lock use graph is extracted according to the pattern provided in the Table VI.

TABLE VI. LOCK USE GRAPH EXTRACTION PATTERN

| **Identifier** | LOCK-USE-GRAPH-EXTRACTOR |
|---|---|
| **Input** | CONTROL-GRAPH |
| **Output** | LG : LOCK-USE–GRAPH |

**Pre-condition**

```
-- there exists a lock acquisition node          --
-- in the control graph                          --

p1:{lock, lock-acq-node ∈ CONTROL-GRAPH |
         (lock::sem::var::type==LOCK
          ∧
          lock-acq-node::sem::primitive::sem-role
          == LOCK-ACQ(lock))}
-- there exists a lock release node in the       --
-- control graph                                 --
P2:{lock, lock-rls-node ∈ CG |
       (lock::sem::var::type==LOCK
        ∧
        lock-rls-node::sem::primitive::sem-role
        == LOCK-RLS(lock))}
```

**Nodes building rules**

```
{∀cg : CG |
     (p1 ∨  p2)}→{LG::S = build(identifier(cg))}
```

**Transitions building rules**

```
{ ∀ l1, n1 ∈ CG1: CONTROL-GRAPH,
  ∀ l2, n2 ∈ CG2: CONTROL-GRAPH,
  ∃ n3,n4  ∈ LG::S |
   ((n1==lock-acq-node) ∧
    (n2==lock-rls-node) ∧
     l1::identifier==l2::identifier) ∧
    (n3==identifier(CG1) ∧
     n4==identifier(CG2)}
→
{ connect(n3,n4,direct,n1) ∧
  connect(n4,n3,direct, n2)}
```

## VII. CONCLUSION AND FUTURE WORKS

This paper deals with the question of how to automatically extract different viewpoint models from source code in order to validate system behavior according to a set of properties. We propose a pattern-based approach that allows specifying the property to check and the transformation rules to apply. For each pattern, a dedicated formalism was introduced. This approach provides more generality than the existing ones. It can be applied for different systems using different languages. Users can plug-in different language parsers and provide the corresponding API semantics.

This approach also allows knowledge capitalization by explicitly defining the verification and transformation processes. It can facilitate verification and validation processes, particularly when these are performed by a third-party organization.

Currently, a prototype was developed allowing checking several design rules such as correct use of locks, atomicity and deadlock. The next step will consist on testing its scalability on great systems.

For future work, we aim to improve our method in order to address temporal constraints. The key point is specifying how to extract a temporal viewpoint model and how to perform the analysis of the satisfaction of temporal constraints.

## ACKNOWLEDGMENT

We would like to thank Simon Bliudze for comments on this work.

## REFERENCES


[1] POSIX Certification – updated on 3 November 2003 - http://www.opengroup.org/certification/idx/posix.html

[2] OSEK VDX version 3.0.3 – July 2004 - http://portal.osek-vdx.org/index.php?option=com_content&task=view&id=9&Itemid=13.

[3] E. M. Clarke, O. Grumberg, and D. A. Peled, "Model checking", MIT Press, 1999, ISBN 0-262-03270-8.

[4] M. P. Bonacina: "On theorem proving for program checking: historical perspective and recent developments", PPDP 2010, pp. 1-12.

[5] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", Conf. Rec. of the 4th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL' 77), ACM Press (New York), Los Angeles, USA, Jan. 1977, pp. 238-252.

[6] K. M. Chandy and J. Misra. "Parallel Program Design: A Foundation". Addison-Wesley, 1988, ISBN 0-201-05866-9.

[7] L. Lamport., "The Temporal Logic of Actions". ACM Trans. on Prog. Lang. and Systems, 1994, pp. 872-923.

[8] Z. Manna and A. Pnueli. "Temporal Verification of Reactive Systems: Safety". Springer-Verlag, New York, 1995, ISBN 0-387-94459-1.


Copyright (c) IARIA, 2011. ISBN: 978-1-61208-168-7

[9] E.A. Emerson, "Temporal and modal logic", Handbook of Theoretical Computer Science, Chapter 16, the MIT Press, 1990, pp. 995-1072.

[10] P. Cousot, R. Cousot, J. Feret, A. Miné, D. Monniaux, L. Mauborgne, X. Rival. "The ASTRÉE Analyzer". ESOP 2005: The European Symposium on Programming, Edinburgh, Scotland, April 2-10, 2005. Lecture Notes in Computer Science 3444, © Springer, Berlin, pp. 21-30.

[11] P.Baudin, A.Pacalet, J.Raguideau, D.Schoen, N.Williams. "CAVEAT : a Tool for Software Validation". In Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), pp. 537-537.

[12] A. Deutsch. *"Static Verification Of Dynamic Properties".* PolySpace Technologies, 27 november 2007, www.polyspace.com

[13] Coverity prevent: Static Source Code Analysis for C and C++, 2008, http://www.coverity.com/library/pdf/coverity_prevent.pdf.

[14] D. Engler and K. Ashcraft, "RacerX: Effective, Static Detection of Race Conditions and Deadlocks", In Proceedings of the Symposium on Operating Systems Principles, October 2003, pp. 237-253.

[15] P. Cousot, R. Cousot, "A gentle introduction to formal verification of computer systems by abstract interpretation". In Logics and Languages for Reliability and Security, J. Esparza, O. Grumberg, & M. Broy (Eds), NATO Science Series III: Computer and Systems Sciences, © IOS Press, 2010, pp. 1-29.

[16] G.S. Avrunin, J.C. Corbett, M.B. Dwyer, C.S. Păsăreanu, S..F. Siegel, "Comparing Finite-State Verification Techniques for Concurrent Software". Technical Report UM-CS-1999-069, Department of Computer Science, University of Massachusetts, 1999.

[17] Sun MicroSystem "Analyzing Program Performance With Sun WorkShop",1999, http://www.atnf.csiro.au/computing/software/sol2docs/manuals/workshop/analyzing/AnalyzingTOC.html

[18] R. C. Seacord. "Secure Coding in C and C++". Addison-Wesley, September 2005. ISBN: 0321335724.

[19] B. Bellay and H. Gall. "A Comparison of Four Reverse Engineering Tools". In Proceedings of the 4th Working Conference on Reverse Engineering (WCRE '97), Washington, DC, USA, 1997. IEEE Computer Society, pp. 2-11.

[20] R. Kollman, P. Selonen, E. Stroulia, T. Syst, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02), Washington, DC, USA, 2002. IEEE Computer Society, pp. 22-33.

[21] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The Software Model Checker Blast: Applications to Software Engineering". *Int. Journal on Software Tools for Technology Transfer*, 9(5-6): pp. 505-525, 2007. Invited to special issue of selected papers from FASE 2004/05.

[22] T. Ball, E. Bounimova, R. Kumar, V. Levin, " SLAM2: Static Driver Verification with Under 4% False Alarms", In Formal Methods in Computer-Aided Design (FMCAD), 2010, pp. 35-42.