On Handling Source Code Positions and Local Variables in LTL Software Model Checking

Guillaume Hétier and Hanifa Boucheneb
Laboratoire VeriForm, Department of Computer Engeneering and Software Engeneering, École Polytechnique de Montréal, Montreal, Canada

Keywords: Model Checking, Concurrency, C, Specification, Verification, Instrumentation, LTL, Assertions.

Abstract: Software model checking techniques can provide the guaranty a system respects a specification. However, some limitations reduce the expressiveness of the most used specification formalisms (the assertions and LTL) and increase the risk of error, especially for concurrent programs. We design a new specification formalism that extends LTL by allowing local variables and code positions in LTL atomic propositions. We introduce validity areas to extend the definition of atomic propositions using local variables and to handle positions in source code. Then, we introduce a source to source transformation that aims to reduce the LTL verification problem to an assertion verification problem for finite programs by building the product between the program code source and the implementation of Büchi automaton. Eventually, we apply this transformation to verify a small benchmark specified with the specification formalism we proposed.

1 INTRODUCTION

Safety and security of critical systems are issues of prime importance. Formal methods aim to provide mathematical proofs that such systems behave as expected. However, they still struggle to scale to large systems. Improving tool performances is one of the main focuses of current research activities.

A crucial point for formal methods is the specification of the system. In order to prove a system correctness, one must first explain what it means for this system to be correct in a formalism understandable by tools. An error in the specification may lead to an invalid result: a correct proof of an unwanted result. Therefore, specification formalism must be user-friendly to reduce the risk of errors while being expressive enough to describe the system properties and features.

In this paper, we focus on a specific formal method: software model checking. More precisely, we restrict our study to software model checking tools that target C concurrent programs (based on the pThread POSIX library). The two main specification formalisms supported by software model checking tools are assertions and Linear Temporal Logic (LTL). However, their power of expression is limited. It is often necessary to modify the code of the program to express a property. Assertions express safety properties: some expression must be true when the assertion is reached during the execution of a program. It is not possible to specify a relation (order, simultaneity...) between events with assertions. For instance, in a concurrent context, ensuring some instructions are executed in a specific order is a non-trivial property that is often desirable to verify. However this property cannot be specified directly with assertions: additional code is necessary to keep track of the order of instructions. On the other hand, LTL may seem a better fit for this situation: it is designed to express properties about the order of events in an execution trace. However software model checkers do not allow the use of local variables and source code positions in LTL specification. This restriction makes it impossible to represent an assertion or to designate a set of instructions with an LTL specification, without additional code.

Mutual exclusion properties are a good example where those limitations make the specification less straightforward. Remarks that in the listings 1 and 2, global variables have to be inserted in the code for the need of specification: it is easy to make an error in this manual instrumentation or to forget to instrument one section of the code.

Hétier, G. and Boucheneb, H.
On Handling Source Code Positions and Local Variables in LTL Software Model Checking.
DOI: 10.5220/0006865706150622
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 615-622
ISBN: 978-989-758-320-9
Copyright © 2018 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Listing 1: Mutual exclusion specification with assertions.

```
int p = 0;
int flag = 0;
void* thread1(void* d) {
  ...
  assert(!flag);
  flag = 1;
  p += 1;
  flag = 0;
  ...
}
void* thread2(void* d) {
  ...
  assert(!flag);
  flag = 1;
  p += 1;
  flag = 0;
  ...
}
```

Listing 2: Mutual exclusion specification with LTL.

```
// LTL: G !(flag1 && flag2)
int p = 0;
int flag1 = 0;
int flag2 = 0;
void* thread1(void* d) {
  ...
  flag1 = 1;
  p += 1;
  flag1 = 0;
  ...
}
void* thread2(void* d) {
  ...
  flag2 = 1;
  p += 1;
  flag2 = 0;
  ...
}
```

In this paper, we introduce an extension of LTL aiming to overcome these limitations by bringing the support for local variables and source code positions in LTL formula, by using the concept of validity areas. Then, in order to verify whether a program respect a specification in the proposed formalism, we present a source to source transformation that reduces LTL verification to assertion verification on the transformed programs, in the restricted context of terminating programs.

2 PRELIMINARIES

2.1 Software Model Checking

Software model checking is a class of formal methods that allow to prove whether a model of a program satisfies a specification or not. In this context, the program is abstracted in a transition system where states represent the configuration of the program memory (or a position in the program) while transitions between states represent the effect of instructions. The specificity of software model checking is that the model is automatically extracted from the source code of a program, without human intervention. The specification is a logical property. Model checking consists in exploring exhaustively the reachable states of the model in order to check whether every path satisfies the specification.

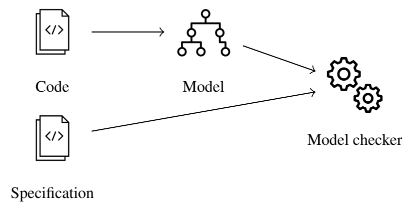

Figure 1: Principle of software model checking.

The main issue of software model checking is combinatorial explosion. The number of states in a model increases exponentially with the size of the program: the time and memory required to explore every state quickly become too important to complete the exploration. This problem is even more present for concurrent programs where every interleaving between threads has to be considered. Several algorithms (D'Silva et al.,) and reduction techniques have been developed in order to fight combinatorial explosion. They often trade precision or completeness for performances and limit themselves to specific categories of properties.

Different formalisms are used to express a specification. Two of the most popular and the most used by model checkers for concurrent programs are assertions and LTL formulas.

2.2 Linear Temporal Logic (LTL)

Linear temporal logic allows to express logical properties over the evolution of a system, represented by an execution trace.

Formally, let AP be a set of *atomic propositions* over the state of the system, and $\Sigma = 2^{AP}$ an alphabet. Each letter of Σ represents a possibly empty set of atomic propositions. We define infinite traces as Σ^ω. A trace $a = (a_0, a_1, ...) \in \Sigma^\omega$ represent the evolution of an execution.

We define inductively LTL by the following grammar, for ϕ and ψ two LTL formulas and $p \in AP$ an atomic proposition:

$\phi, \psi := \text{true} \mid p \mid \phi \wedge \psi \mid \neg\phi \mid X\phi \mid \psi U \phi$

Given an infinite trace $a = (a_0, a_1, ...) \in \Sigma^\omega$, LTL has the following semantics :

$$s \models p \equiv s_0 \models p \quad (1)$$

$$s \models X\phi \equiv (s_1, s_2, \dots) \models \phi \quad (2)$$

$$s \models \phi U \psi \equiv \exists k, (s_k, s_{k+1}, \dots) \models \psi \wedge \forall i <= k, (s_i, s_{i+1}, \dots) \models \phi \quad (3)$$

$\neg$, $\wedge$, and $true$ are interpreted in the usual way. $X\phi$ means the next state of the trace satisfy ϕ and $\phi U \psi$ means the states of the trace must satisfy ϕ until one state satisfy ϕ.

The operators F and G are then defined as $Fp \equiv trueUp$ and $Gp \equiv \neg F \neg p$. F express the fact an atomic proposition is eventually true and G express the fact it is always true.

LTL is mainly limited by the restrictions imposed over atomic propositions: global variables do not allow formulas to be expressive enough. Software model-checking tools usually allow an atomic proposition to be any side effect free boolean expression on global variables. Allowing local variables and code positions in LTL formulas would improve LTL expressiveness.

3 HANDLING LOCAL VARIABLE AND CODE SOURCES POSITIONS

Several issues must be overcome to allow local variables and code position in a LTL specification. These issues have been previously avoided by restricting atomic proposition to global variable.

3.1 Handling Local Variable

Global variables are uniquely named in a program, so their name is enough to refer to them without ambiguity. However, the name only does not allow to identify unambiguously a local variable.

- First, local variables in different contexts can share the same name. In order to identify them uniquely, a commonly used solution is to prefix the name of the variable with the name of the function: for instance, the variable foo in a function bar can be designated by bar::foo.
- Second, several instances of the same local variable (from the same lexical definition) can coexist in the memory. It happens for local variable in recursive functions or when several threads are executing the same code. In this situation, variable instances are defined dynamically therefore it is difficult to designate them in a static way. We do not propose a solution to this issue. We present a partial solution implemented by the model checker Divine (Barnat et al., 2013) in the section 5.

3.2 Handling Code Source Positions

In order to use code source positions in the specification, we need a way to designate a position that is both robust, user-friendly, and precise enough. An external mechanism, such as using line numbers is not robust: any modification of the code source could break the specification. Therefore, we use marks in the source code. We choose labels as they are part of the C language and are already designed to mark an instruction.

Further on, we say that a program reach a code position when an instruction pointer of the program point on the instruction designated by the label.

Next, we need to integrate code source locations to the LTL specification. Three options may be considered:

1) an atomic proposition is either a position or a condition on the program variable, on an exclusive way. For instance, if pos1 and pos2 represent two positions in the program, we specify the variable x is not null between pos1 and pos2 by $G(\{\text{pos1}\} \implies \{x \neq 0\} U \{\text{pos2}\})$.
2) an atomic proposition is composed of both a condition on the program variables and an information about positions. The previous example would become $G\neg p$, where p is the atomic proposition true for every state between pos1 and pos2 and where x is null. We could write p as $\{x! = 0\} \wedge [pos1, pos2]$ with the convention an interval of positions is evaluated to true when an instruction pointer designates an instruction between the two positions.
3) position information are constraints applied to temporal operators, in a similar way than the MTL logic (Koymans, 1990). We would then write $G_{[pos1,pos2]} \neg \{x \neq 0\}$.

We rejected the third option to preserve the standard behavior of LTL operators.

3.3 Extending LTL Atomic Propositions

An atomic proposition of a LTL formula is intrinsically global: for every state s of the program (i.e. a configuration of the program memory and a set of values for instruction pointers), it may be necessary to

evaluate whether s verify the atomic proposition. This depends upon the value of a boolean expression in the state s, evaluated using the value of variable in the state s.

However, when this expression use local variables, it can only be evaluated in a state s where all those local variables are defined. Therefore, an atomic proposition is correctly defined only in the context where all its variables are defined (are not out of scope). In order to get a valid definition of an atomic proposition involving local variable, it is necessary to extend its definition to the whole set of program states.

3.4 Syntax and Semantics of the Proposed Specification

3.4.1 Syntax

Our specification is composed of a standard LTL formula and the definition of the atomic propositions that are used. Atomic propositions are composed of the following parts:

- **a name:** it is the identifier of the atomic proposition.
- **a validity area:** it defines a block of instructions. A validity area is defined by two labels. Every instruction pointer that reach the entry label must reach the end label before it exits the context (a branching instruction must not allow to go out of the validity area while avoiding the exit label).
- **an evaluation function:** the name of a pure boolean C function. This function is used to decide whether a program state in the validity area verify the atomic proposition. Its parameters are provided, in order, by the parameter list.
- **a parameter list**, to specify the variables to transmit to the evaluation function. Every parameter has to be defined in the validity area.
- **a default value:** an atomic proposition takes its default value out of its validity area.

The Table 1 presents the grammar of an atomic proposition.

3.4.2 Semantics

Classic logic operators and temporal operators follow the usual semantics of propositional logic and LTL. An atomic proposition p is evaluated in a state s by:

- if the state s is not in the validity area of the atomic proposition p, then p takes its default value.
- if the state s is in the validity area of the atomic proposition p, then p takes the value returned by the evaluation of its evaluation function in the state s.

Validity areas have a double interest. The user can specify manually where an evaluation function should be used. He can exclude variable initializations and other areas where program invariants are temporarily broken. Moreover, validity areas allow to specify properties about intervals of program positions directly in atomic propositions. It reduces the complexity of the LTL formula, which improves the performances of the verification task.

Our specification is a weaker restriction of LTL than the one used by most model checkers, so expressing an LTL formula on global variables only is trivial (validity areas are the whole program). It is also possible to express assertions: it can be represented by the formula Gp with p a proposition atomic such as its validity area is the position of the assertion only, with true as a default value and the expression in the assertion as an evaluation function. Therefore, our specification formalism allows to express both the classical version of LTL and assertions, while being capable to specify directly properties such as mutual exclusion.

The listings 3 and 4 show how the presented formalism can be used to specify a mutual exclusion property. No additional variable is needed. In this example, the evaluation function does not depend of the value of the variables, which explains why parameter lists are empty in the specification.

Listing 3: Code with critical areas delimited using labels.

```
int p = 0;
int f_ev () { return 1; }
void* thread1(void* d) {
  ...
b_p1:
  p += 1;
e_p1: ;
  ...
}
void* thread2(void* d) {
  ...
b_p2:
  p += 1;
e_p2: ;
  ...
}
```

Table 1: Grammar of atomic propositions.

<atomic-proposition>	::=	<proposition-id> <evaluation-function> <parameters> <default> <validity-area>
<proposition-id>	::=	*name of the proposition*
<evaluation-function>	::=	*C pure boolean function*
<parameters>	::=	<parameter> <parameters> | *nil*
<parameter>	::=	<global-parameter> | <local-parameter>
<global-parameter>	::=	*variable name*
<local-parameter>	::=	*function name* :: *variable name*
<default>	::=	*boolean*
<validity-area>	::=	<label> <label>
<label>	::=	*name of a C label*

Listing 4: Specification for a mutual exclusion property.

```
{ "ltl": "G(! (p1 && p2))",
  "pa": [
    {"name": "p1",
      "default": false,
      "expr": "f_ev",
      "span": ["b_p1", "e_p1"],
      "params": []
    },
    { "name": "p2",
      "default": false,
      "expr": "f_ev",
      "span": ["b_p2", "e_p2"],
      "params": []
    }
  ]
}
```

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4 LTL TO ASSERTIONS

Most of the existing model checker for concurrent C programs only accept assertions as specification and do not support LTL. To adapt one of these tools or to create a new one would have been a task far beyond the scope of this work. Instead, we designed a source to source translation. From a code source and a specification in the formalism presented in section 3, we build a program specified with assertions. The transformation respect the following invariant: "the obtained program satisfy the assertion if and only if the initial program satisfy its specification".

4.1 Transformation Desing

The classic algorithm for LTL model checking consists in building the product between a model of the system and a Büchi automaton representing the negation of the LTL specification. The model checker then explores the product: every valid path that reach an accepting cycle corresponds to an execution violating the specification and is reported as an error.

Our transformation reuse this scheme with one major difference, as presented in (Morse et al., 2015): instead of building the product between the model of the system and the Büchi automaton, we implement the Büchi automaton in C and we build its product with the code of the system. Thus, we obtain a new code which contains the automaton and that can be explored by most model checking tools.

The global organization of our transformation is presented in Figure 2.

4.1.1 Implementing the Büchi Automaton

We use LTL2BA (Gastin and Oddoux,), an efficient and well-known tool, to compute the structure of the Büchi automaton from the LTL formula. Then, we need to translate the automaton to C code. The transformation uses the following elements:

- a global variable representing the current state of the automaton
- a set of boolean variables representing the current valuation of atomic propositions
- a function implementing the transition function of the automaton. Every call to this function allows the automaton to take a transition (if several transitions are possible, a non-deterministic choice is made).

4.1.2 Build the Product

The next step is to synchronize the execution of the program with the Büchi automaton. We instrument the program with instructions that update the values of atomic propositions and trigger transitions in the automaton. It would be useless and inefficient to instrument every instruction of the program. We only need to instrument instructions that could modify an atomic proposition value: the entrance and exit in a validity area, and the assignment of a parameter of the evaluation function inside the validity area are the first candidates. Assignments to global variables out of

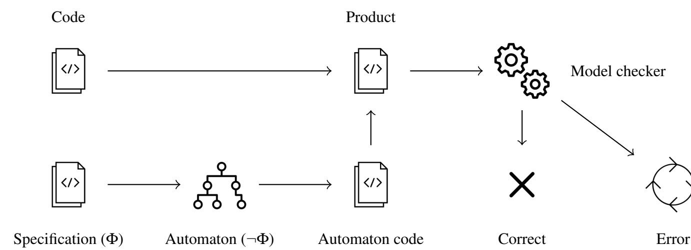

Figure 2: Organization of the source to source transformation.

validity areas must also be instrumented: they could have an impact on the valuation of an atomic proposition when another thread is in the validity area. Instrumentation is placed in atomic blocks to avoid the generation of additional interleaving. Examples of instrumentation can be found on the repository of the project: https://github.com/xNephe/baProduct.

4.1.3 Extract the Results

The last step is to use assertions to signal the model checker when an error state is reached during the exploration. This can be done easily when it is possible to know in a finite time whether the execution trace is valid or not. However, when it is not possible, it is extremely difficult to detect a cycle in the program from the inside of itself. To avoid this problem, we limit ourselves to terminating programs, where every execution terminates. This raise another issue: LTL is only defined on infinite traces. Although several extensions of LTL to finite traces exist (Beer et al.,), none make consensus. Here, we use the stuttering extension, that consists in making a finite trace infinite by repeating its last state: it matches well a program behavior, where the notion of next state is not precisely defined. Also, a program can easily be assumed to stay in its last state indefinitely at the end of the execution.

Given the infinite extension, we can now determine whether a finite trace is accepted by precomputing the result for every state and every configuration of atomic properties. To make a distinction between the results obtained using only a finite prefix of the trace and those that make use of the infinite extension (and so may be different with another trace suffix), we use a four valuated logic. The possible outcomes are:

- VALID SURE: the automaton rejects the execution trace for every extension
- VALID MAYBE: the automaton rejects the execution trace for the stuttering extension but may accept it with another extension
- ERROR MAYBE: the automaton accepts the execution trace for the stuttering extension but may reject it with another extension
- ERROR SURE: the automaton accepts the execution trace for every extension

4.1.4 Implementation

We implemented a tool to produce this instrumentation, available at https://github.com/xNephe/baProduct. The tool is implemented in OCaml (Leroy et al., 2016) and use the CIL library to instrument the code. Büchi automatons are produced using LTL2BA (Gastin and Oddoux,).

4.2 Experimental Results

We tested the presented tool and specification formalism on a benchmark of tests (available with the tool). It consists on 11 small scenarios (less than 100 line each) implementing common concurrent patterns (producer-consumer, data race, answer on request, road signals...). The tests focus on concurrency and errors related to the misuse of synchronization primitives. We used two bounded model checkers as backends to perform verification tasks over the instrumented code, ESBMC 4.2 (Cordeiro et al.,) and CBMC 5.7 (Clarke et al.,), running on a computer with a processor Intel Core i7 (4 cores at 3.7GHz, 6 Go of RAM and Fedora 25 (64 bits). The tables 2 and 3 summarize the results we obtained.

We observe that ESBMC and CBMC succeed in most of the verification tasks, although some unexpected results are observed: ESBMC find an error in a valid test case and CBMC miss an error in an invalid test case. As far as we can tell, these behaviors are not triggered by an error in the transformation. They may be related to a loss of precision due to performance optimizations in the backend model checkers. ESBMC fails on the test *battery_var* as well after having used all the available memory.

Performance measurements show that CBMC is much more efficient than ESBMC on this benchmark. We explain this difference by the different algorithms used to handle concurrency: ESBMC explores every interleaving independently while CBMC builds a single verification goal integrating all interleaving. Most

Table 2: Results of the verification of an instrumented benchmark.

Test scenario	CBMC	ESBMC	Expected result
answer_simple	ERROR MAYBE	ERROR MAYBE	ERROR MAYBE
answer_simple_valid	**ERROR MAYBE**	VALID MAYBE	VALID MAYBE
battery_simple	ERROR SURE	ERROR SURE	ERROR SURE
battery_var	ERROR SURE	*Runtime error*	ERROR SURE
crossing_exclusive_green	VALID MAYBE	VALID MAYBE	VALID MAYBE
crossing_exclusive_mutex	VALID MAYBE	VALID MAYBE	VALID MAYBE
crossing_GF_green	ERROR MAYBE	ERROR MAYBE	ERROR MAYBE
crossing_order	VALID MAYBE	VALID MAYBE	VALID MAYBE
prod_cons_mutex	VALID MAYBE	VALID MAYBE	VALID MAYBE
prod_cons_simple	ERROR SURE	**VALID SURE**	ERROR SURE
race	ERROR MAYBE	ERROR MAYBE	ERROR MAYBE

of our tests consist on finding a specific interleaving, which is more compatible with CBMC approach.

5 RELATED WORK

5.1 Support of Local Variable in LTL

Divine is a model checker for concurrent C programs which supports LTL specifications. In (Barnat et al.,), a formalism supporting local variables and positions in the program in LTL specification is presented. It is based on a set of macros that allow to modify directly the value of an atomic proposition in the code and to bind an expression (containing or not local variables) to an atomic proposition in a lexical context. This approach allows to express atomic propositions in a more intuitive manner at the cost of being more invasive in the code source. However, they do not introduce the concept of validity areas: it is more complicated to restrict the use of an evaluation function to specific instructions, and relations between values of variables and positions in the code has to be realized in the LTL formula, which increases its complexity.

A partial solution to the issue of the identification of local variable in recursive functions or threads is also introduced. Using the fact that most of the time, the interesting event is when an atomic proposition takes its non-default value, their solution is to compute the valuation of the atomic proposition in every possible context and to keep the non-default value if it is produced at least once. This solution corresponds to the expected behavior for safety or reachability properties. However it is an arbitrary decision in general.

5.2 Reduction of LTL Specification to Assertions

We based our source to source transformation on the work of J.Morse, in (Morse et al., 2015). However, in this work, the automaton is implemented in an additional thread, that behave as an observer. In order to avoid the generation of many interleavings, a model checker specific instruction is used, that allows context switch to the observer thread at specific position of the code only. By using a function instead of a thread, our version avoid the use of this model checker specific instruction and remains compatible with several tools.

6 CONCLUSION

In this paper, we highlighted some of the limitations of two of the most used specification formalism in the context of concurrent C programs. We proposed a new specification formalism based on LTL that answer to those limitations thanks to the concept of validity areas: this mechanism allows to handle both local variables and code positions in an LTL specification. Then, in order to test the proposed formalism, we present a source to source transformation that reduce the problem of LTL verification to the verification of assertion for terminating programs and we implement it in a tool.

However, our specification formalism may not be intuitive to write, which could be improved in a future work. Moreover, the instrumentation and code transformation we propose does not use any reduction technique, which limits it to very small programs. The addition of partial order reductions taking into account the structure of the Büchi automaton may improve the performances. It would also be interesting to implement the support for the proposed specification directly in a model checking tool. It would probably be possible to overcome the restriction to terminating programs and to gain both in precision and performances by doing so.

Table 3: Performances of the verification of an instrumented benchmark.

	CBMC		ESBMC	
Test scenario	Time (s)	Memory (MB)	Time (s)	Memory (MB)
answer_simple	0.87	42	5.57	56
answer_simple_valid	0.81	41	12.91	72
battery_simple	1.07	42	13.95	61
battery_var	666.04	954	N/A	OOM
crossing_exclusive_green	2.93	44	32.12	119
crossing_exclusive_mutex	1.73	44	32.20	120
crossing_GF_green	1.99	46	41.70	117
crossing_order	1.02	40	11.62	56
prod_cons_mutex	10,046.32	593	229.99	28
prod_cons_simple	1.70	52	454.00	2711
race	1.08	41	5.65	28

REFERENCES

Barnat, J., Brim, L., Havel, V., Havlíček, J., Kriho, J., Lenčo, M., Ročkai, P., Štill, V., and Weiser, J. (2013). DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)*, volume 8044 of *LNCS*, pages 863–868. Springer.

Barnat, J., Brim, L., and Rockai, P. Towards LTL model checking of unmodified thread-based c c++ programs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7226 LNCS, pages 252 – 266.

Beer, I., Ben-David, S., and Landver, A. On-the-fly model checking of RCTL formulas. In *Proceedings of the 10th International Conference on Computer Aided Verification*, CAV '98, pages 184–194. Springer-Verlag.

Clarke, E., Kroening, D., and Lerda, F. A tool for checking ANSI-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-540-24730-2_15.

Cordeiro, L., Morse, J., Nicole, D., and Fischer, B. Context-bounded model checking with ESBMC 1.17 (competition contribution). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7214 LNCS, pages 534 – 537.

D'Silva, V., Kroening, D., and Weissenbacher, G. A survey of automated techniques for formal software verification. 27(7):1165–1178.

Gastin, P. and Oddoux, D. Fast LTL to büchi automata translation. In *Computer Aided Verification*, pages 53–65. Springer, Berlin, Heidelberg.

Koymans, R. (1990). Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299.

Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., and Vouillon, J. (2016). *The OCaml system (release 4.04): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique.

Morse, J., Cordeiro, L., Nicole, D., and Fischer, B. (2015). Model checking ltl properties over ansi-c programs with bounded traces. *Software & Systems Modeling*, 14(1):65–81.