Model Checking Temporal Metric Specifications with Trio2Promela [⋆]

Domenico Bianculli[1], Paola Spoletini[2], Angelo Morzenti[2], Matteo Pradella[3], and Pierluigi San Pietro[2]

[1] Faculty of Informatics, University of Lugano, Switzerland
domenico.bianculli@lu.unisi.ch
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
[3] CNR IEIIT-MI, Italy
{spoleti, morzenti, pradella, sanpietr}@elet.polimi.it

Abstract. We present Trio2Promela, a tool for model checking TRIO specifications by means of Spin. TRIO is a linear-time temporal logic with both future and past operators and a quantitative metric on time. Our approach is based on the translation of TRIO formulae into Promela programs guided by equivalence between TRIO and alternating Büchi automata. Trio2Promela may be used to check both purely descriptive TRIO specifications, a distinguishing difference with other model checking tools, and usual Promela programs for which the user needs to verify complex temporal properties. Then, we report on extensive and encouraging experimentation results, and compare Trio2Promela with similar tools.

Keywords: temporal logic, model checking, Spin.

1 Introduction and Background

TRIO is a first order, linear-time temporal logic with both future and past operators and a quantitative metric on time, which has been extensively applied to the specification, validation and verification of large, critical, real-time systems [1, 2]. TRIO formulae are built much in the same way as in traditional mathematical logic, starting from variables, functions, predicates, predicate symbols, and quantifiers over finite or infinite, dense or discrete, domains (a detailed and formal definition of TRIO can be found in [3]). Besides the usual propositional operators and the quantifiers, one may compose TRIO formulae by using a pair of basic modal operators, called Futr and Past, that relate the *current time*, which is left implicit in the formula, with another time instant: the formula $\mathit{Futr}(F, t)$, where F is a formula and t a term indicating a time distance, specifies that F holds at a time instant at t time units in the future from the current instant (symmetrically for past). Notice that the usage of both past and future modalities in TRIO is widely recognized to make specifications simpler and more concise than using either only future or only past operators.

[⋆] Work partially supported by the MIUR project "FIRB: Applicazioni della Teoria degli Automi all'Analisi, alla Compilazione e alla Verifica di Sistemi Critici e in Tempo Reale", and the IST EU project "PLASTIC", contract number. 026955.

Many derived temporal operators can be defined from the basic operator through propositional composition and first order quantification on variables representing a time distance. The traditional operators Since and Until of linear temporal logics, as well as many other operators, can easily be obtained as TRIO derived operators. For instance, $\mathit{SomF}(F)$ (Sometimes in the Future) is $\mathit{Until}(\text{true}, F)$ and corresponds to the "Eventually" operator of temporal logic; $\mathit{AlwF}(F)$ (Always in the Future) is $\neg\,\mathit{SomF}(\neg F)$ (F will always hold), that is the "Globally" operator. Moreover, TRIO adds another level of succinctness because of the metric operators Lasts and Lasted and their duals $\mathit{WithinF}$ and $\mathit{WithinP}$. $\mathit{Lasts}(F, c)$ means that F will hold for c instants in the future and $\mathit{WithinF}(F, c)$ means that F will hold within c instants in the future. For instance, a TRIO formula such as: $\mathit{WithinF}(\mathit{Lasts}(B, h), k)$ for some $h, k > 0$, may be expressed in LTL only with a formula of length proportional to $h \cdot k$.

Over the years a variety of methods and tools have been defined to support typical validation and verification activities in TRIO, such as: 1) testing, by generation of execution traces or checking of such simulations for consistency against the TRIO specification [4], 2) derivation of system properties in the form of theorems, based on the definition of a suitable axiomatization of the logic and on its encoding in the notation of a general purpose theorem prover, such as PVS [5].

In the present paper, we present another tool, called Trio2Promela, for the mechanical verification of a decidable subset[4] of TRIO specifications, by using a well-known model checker such as Spin [6] to perform proof of properties and simulation. The approach and background theory upon which Trio2Promela is constructed was originally presented in [7, 8]. The main aim of the present paper is to report on our experience of actually using the tool, now fully implemented and publicly available, together with examples and the experiments summarized in the paper (at `http://www.elet.polimi.it/upload/sanpietr/Trio2Promela.zip`).

Trio2Promela can be used, at various levels of generality, to support satisfiability checking of generic TRIO formulae (and hence property proof) and model checking. In the former activity, every possible interpretation structure for the formula is potentially enumerated (but great care is of course taken to allow Spin to determine, during a verification, that many structures can be safely ignored as early as possible). In the latter activity, Trio2Promela translates the property to be checked from TRIO into Promela, combining the resulting code with a pure-Promela model to perform verification. When the desired property is fairly complex or contains several bounded temporal statements, which is typical e.g. of real-time systems, the traditional approach of generating a so-called *never claim* for Spin, i.e., an automaton specifying the negation of a temporal logic property over the already available state-transition system, becomes unfeasible. Both the internal LTL to Büchi automata translator, and other more recent tools, like LTL2BA [9] and Wring [10], simply cannot manage a complex real-time statement with metric operators, as we report in Sect. 3. For instance, the LTL version of a statement of the form "every occurrence of event A must be followed within 20 time units by an occurrence of event B", which in TRIO can be modeled as simply as $\mathit{AlwF}(A \rightarrow \mathit{WithinF}(B, 20))$ cannot be translated by LTL2BA or Wring (the system

[4] Essentially, a concise (thanks to metric operators) version of LTL with past operators. For instance, $\mathit{Lasts}(A, 3)$ corresponds to $\mathsf{X}(A \wedge \mathsf{X}(A \wedge \mathsf{X}(A)))$ in LTL.

was stopped after waiting for 24 hours), even if it actually corresponds (if one time unit is taken to correspond to one transition) to a Büchi automaton with only 21 states. Trio2Promela is able to translate the above statement into a short Promela program almost instantaneously.

The Trio2Promela tool is ideally based on the translation of TRIO formulae into a set of Promela processes, derived from a well known correlation between temporal logic and alternating automata [11]. As opposed to previous approaches, however, the Promela code generated from TRIO formulae performs an actual simulation of an alternating automaton, rather than simulating a Büchi automaton equivalent to the alternating one, resulting in a Promela code whose size is essentially proportional to the length of the TRIO specification (although of course the state space may not be affected in either way). This is by itself a remarkable result since TRIO, which contains metric and past operators, is quite concise compared with propositional, future-time temporal logics like LTL. Our approach can be naturally compared with recent works appeared in the literature (such as those on LTL2BA and Wring) that aim at the translation of LTL properties into Büchi automata and then Promela programs). We point out, however, that the result of those tools is usually the construction, as in the traditional model-checking scenario, of a *never claim*. In our approach, instead, the Promela processes obtained from the translation of the TRIO specification define an acceptor of a language over the alphabet of the specification, and therefore it must be coupled with some additional Promela program fragments generating the values, over time, for the logical variables that constitute the specification alphabet. Our translation techniques, combined with other optimizations, related for example with the management of TRIO past-time operators, allowed us to perform efficiently the verification in Spin of some significant benchmarks.

2 Trio2Promela

Trio2Promela translates a TRIO specification, i.e., a complex TRIO formula, into Promela code. The translation is based on a correspondence between TRIO and *Alternating Modulo Counting Automata* (AMCA) described in [8]. The main idea is based on the well-known correspondence between Linear Temporal Logic and Alternating Automata (which are a generalization of non-deterministic automata: see for instance [12]), together with *counters*, associated with states of the AMCA, that are used to express TRIO's metric temporal operators in a natural and concise manner.

Each temporal subformula, i.e., one of the form $Z(\_)$ for a temporal operator Z, of the original specification is translated into a single state of the AMCA. Then, an AMCA is *directly* translated into a Promela program: every state of the automaton will correspond to a single type of process (i.e., a Promela proctype), to be instantiated when needed. An or-combination of states $s_1 \vee s_2$ in the transition function corresponds to a non-deterministic choice `(if ::s1; ::s2; fi)`, while an and-combination $s_1 \wedge s_2$ corresponds to the starting of two new Promela process instances, having type `s1` and `s2`, respectively. Hence, the produced code consists of a network of processes, each corresponding to a temporal subformula of the original specification. Each Promela process receives as input a chronological sequence of values taken by the propositional letters that constitute the alphabet of the associated TRIO formula, and then it returns its computed truth value to the network. When the process representing the whole TRIO formula being analyzed returns `false`, every process in the network is stopped, and the analysis terminates.

When Trio2Promela is used to translate a complex TRIO property in the context of traditional model-checking, the input values to the processes come from a Promela program that encodes the operational model under analysis. On the other hand, when Trio2Promela is used to check satisfiability of a TRIO specification, the input values come from a purely generative Promela component that exhaustively enumerates all possible values over time of the propositional letters.

Notice that in the translation we encode time (integer-valued) constants of TRIO formulae into int variables, so the size of the resulting Promela code is linear in the size of the AMCA, and therefore also in the size of the original TRIO specification. Here our approach differs substantially from others (such as LTL2BA and Wring): these translate LTL formulae (which in the first place are less compact than TRIO formulae, as they cannot include integer values representing time constants, and therefore must use long chains of nested X operators), into Spin *never claims* or Büchi automata whose size can grow to become unmanageable even for relatively simple specifications, as it will be shown in Sect. 3 on experimental results.

As TRIO past operators are concerned, we take advantage of the fact that time is unlimited only towards the future (there is a *start* time instant) to treat past operators differently, using a technique illustrated in our paper [7] that stores a bounded amount of information derived from the past portion of the sequence of input values.

Our approach becomes practically feasible in Spin by adopting a set of optimizations, such as:

– Processes representing future operators are in general, with the only exception of Futr, grouped in a unique process.
– Process instances for Futr are reused (since every new process is very costly for verification).
– Various kinds of TRIO subformulae are simplified (e.g., the nesting of a bounded future and a bounded past operator is replaced by one equivalent operator).
– Communication channels are used to abort related processes when a process produces a `false` result.

Tool description. The translator front-end, implemented in Java with ANTLR[5], takes a TRIO file in input, containing the declarations of variables, constants and the logic formulae of the specification; each variable is associated with an integer domain. The output of the parsing phase is an array of abstract syntax trees, one for each formula in the specification. Each tree in the array is processed by a chain of tree walkers that perform optimizations on the propositional connectives of the TRIO formula and nested operators (such as $\mathit{Futr}(\mathit{WithinF}(\_,\_),\_)$ or $\mathit{Past}(\mathit{WithinP}(\_,\_),\_)$, apply derivation rules for derived operators as defined in TRIO (e.g., the formula $\mathit{NextTime}(A,t)$ becomes $\mathit{Futr}(A,t) \wedge \mathit{Lasts}(\neg A,t)$, push inward in the formula all the negations, taking into account the definition of TRIO operators (e.g., $\neg \mathit{Lasts}(A,t)$ is equivalent to

[5] http://www.antlr.org

$WithinF(\neg A, t)$). The back-end is composed of a set of translation procedures, each one implementing the translation schema for a TRIO operator; they produce: the temporal constant characterizing the metric operator, the local variables needed in the main Promela process, the body code for an autonomous Promela process and the associated launching and error propagation code, the additional code for formula evaluation, and the logic expression corresponding to the truth value of the formula. The code resulting from the translation of each operator is then composed with the generative component to produce the final Promela program.

3 Experimental results and comparisons

We extensively experimented with the tool and compared it with the main toolkits available for translating from temporal logic to Büchi automata. In particular, we compared Trio2Promela with LTL2BA, which is probably one of the most efficient LTL-to-Büchi automata translators. We also experimented with Wring, but we do not report the results here because its translation times were always worse than those of LTL2BA. The same happens for other translators, such as LTL→NBA [13] (which may lead to slightly smaller automata than LTL2BA, but it is significantly slower).

The setup for all experiments was the following: we used a laptop with an Intel Pentium M 1.2 GHz, and 632 Mb RAM. Spin's version was 4.2.3, LTL2BA's was 1.0.

The comparison was conducted with reference to two case studies: the Kernel Railroad Crossing Problem (KRC, see [14]) and a version of Fischer's protocol (FP, [15]). KRC and FP are two fairly good representatives of very different problems: the former is a typical real-time system, with a limited number of reachable configurations but with quantitative timing requirements; Fischer's protocol instead has basically very weak quantitative timing requirements, but it has strong combinatorial aspects, making the number of possible configurations grow quickly with the number of processes.

Satisfiability Checking and Model Checking. It is well-known that LTL model checking, while being PSPACE-complete, is linear in the number of the states of the automaton and exponential in the size of the LTL formula to be checked. On the other hand, LTL satisfiability checking is exponential in the size of the formula. Hence, our first set of experiments has studied the unavoidable loss in efficiency of going from model checking to satisfiability checking. Hence, we developed one logic model, defined in TRIO assuming an underlying discrete time model, and one automaton model of both KRC and FP, which were coded in Promela. Table 1 shows the experimental results. The KRC models used various integer values for the time constants of the problem, with KRC1 having the smaller constants and KRC3 the highest. The property proved was a safety property. Also, the results on Fischer's protocol FP are shown, with a number of processes going from 2 to 5, for checking a simple mutual exclusion property.

Clearly, the comparison of satisfiability checking and model checking may be affected by the choice of the two different, although equivalent, models for the same system: one could have defined a "smart" logic model and a "sloppy" automaton, or viceversa. The results are nonetheless of some interest. The exponential blow-up for satisfiability checking against model checking tends to show up, but still the tool can

Table 1. Comparison of satisfiability checking and model checking using KRC and FP

	Satisfiability checking (Trio2Promela)[ab]			**Model checking of a Promela model against a safety formula translated with Trio2Promela**[c]			**Model checking of a Promela model against a safety formula translated with LTL2BA**[c]		
	Mem (MB)	States	Time (s)	Mem (MB)	States	Time (s)	Mem (MB)	States	Time (s)
KRC1	8.2	105151	2	2.6	298	1	2.6	909	1
KRC2	31.8	361627	8	2.6	674	1	2.6	2233	1
KRC3	171.0	1371870	46	2.6	1390	1	2.7	3658	1
FP2	3.0	13179	1	2.6	345	1	2.6	326	1
FP3	13.3	304047	2	2.7	3355	1	2.6	3240	1
FP4	241.1	6321520	59	3.5	27977	1	3.3	41694	1
FP5	EX[d]	NC[e]	NC[e]	9.7	215886	1	9.8	222940	1

[a] A temporal logic model has been translated into Promela, along with a safety property.

[b] Satisfiability checking with LTL2BA is infeasible (i.e. no translation can be generated) for every example shown in the table.

[c] Only the safety property has been translated into Promela.

[d] Memory exhausted (>400 MB).

[e] Not completed.

manage with fairly large time constants in the KRC model (case KRC3). With Fischer's protocol, instead, the state explosion is more substantial. These data show also a large improvement over our previous work [8], where, for instance, the case KRC1 required four times the number of states and the case KRC3 could not be dealt with.

Satisfiability checking of KRC and FP temporal logic models is infeasible in LTL2BA, i.e., LTL2BA ran for more than 24 hours without providing a translation, while the translation time of both examples with Trio2Promela is negligible (under one second). When Trio2Promela is used for model checking (hence, only a short TRIO formula is translated into Promela), the performance of the verification does not appear substantially different from the case when LTL2BA is used for checking the same Promela program. Hence, even though Trio2Promela is geared towards translation of large metric-temporal logic formulae, rather than applying sophisticated optimizations to small LTL formulae as LTL2BA does, it appears that in practice Trio2Promela works at least as well (or even better) than LTL2BA even on small LTL formulae, at least when used for model checking.

Translation of short formulae. To better understand the relative strength of LTL2BA and Trio2Promela, we also ran both tools on a set of short formulae. The comparison could be done only on formulae without past operators, since, while the theory underlying LTL2BA has been extended to deal with past formulae [16], the toolkit publicly available for LTL2BA can only deal with the future fragment LTL. Purely past formulae are anyway translated by Trio2Promela into very small Promela programs with only one process (e.g., the formula $AlwF(A \to WithinP(B, 20))$ is translated into one single Promela process with 21 states).

In Table 2 we give translation times and also a measure of the size of the Promela code. For LTL2BA, we give the number of states of the corresponding Büchi automaton (since every state is explicitly listed), while for Trio2Promela we give the sum of the states of each Promela process defined by the translation, and also the total number of

Table 2. Comparison of translation times and size

	Translation time (s)		Size	
	LTL2BA	Trio2Promela	LTL2BA[a]	Trio2Promela[b]
Safety-KRC	<1	<1	2	9–1
Mutex-Fischer	<1	<1	2	8–1
$AlwF(A \rightarrow WithinF(B, 10))$	<1	<1	11	218–11
$AlwF(A \rightarrow WithinF(B, 15))$	222	<1	16	308–16
$AlwF(A \rightarrow WithinF(B, 17))$	3127	<1	18	308–18
$AlwF(A \rightarrow WithinF(B, 20))$	infeasible	<1	21[c]	398–21
$AlwF(A \wedge SomF(B))$	<1	<1	1	20–1
$AlwF(A \wedge SomF(B) \wedge Lasts(C, 5))$	<1	<1	5	26–1

[a] Number of states of the corresponding Büchi automaton.

[b] Sum of states of each Promela process – total number of processes.

[c] Estimated.

processes. For Trio2Promela, this is a very indirect measure of the number of states of the corresponding Büchi automaton, which can only be determined dynamically.

The results show that LTL2BA's translation times grow quickly also on very small formulae with metric temporal operators, even though the resulting Büchi automaton is small. For instance, a formula like $AlwF(A \rightarrow WithinF(B, c))$, where c is a constant, corresponds to a Büchi automaton with $c+1$ states, and it is translated by Trio2Promela into c processes of 18 states each, and one process of 38 states. However, LTL2BA fails to translate the formula within 24 hours for c greater than 19. The reason for LTL2BA's failure is that on this (and other) kind of formulae, the construction method of LTLBA must pass through a stage where a (generalized) Büchi automaton is built, describing all possible configurations of the original alternating automaton. LTL2BA then performs an optimization phase that may, at least for this kind of formulae, lead to an optimal Buchi automaton. However, the number of states of the intermediate automaton may be so large that it cannot be handled, causing the tool's failure.

4 Conclusions

We presented experimental evidence that, when dealing with metric temporal logic, the Trio2Promela toolkit has many advantages over existing toolkits that address the same goal. In fact, Trio2Promela always derives a Promela code whose size is linear in the size of the original TRIO formula (which may be substantially smaller than an equivalent LTL formula). This is obtained by avoiding, at least at translation time, the state explosion problem, thanks to the fact that we generate a Promela program that will simulate (at verification time) the alternating automaton. The alternation removal is then left to the model checker, allowing the verification of many temporal logic formulae which could not be translated into Promela by means of other techniques. In fact, if the alternation is removed during the translation phase, as all techniques we know of do, then there are many cases where a translator cannot even build the resulting automaton (where all states are explicitly enumerated). This happens, for instance, when specifications are very large, or use metric temporal operators, or mix past and future temporal operators.

Our experiments also support the conclusion that, even when the translation of an LTL formula is possible with other tools, the performance of model checking in Trio2Promela leads to comparable performance results. Hence, current experimental evidence shows that Trio2Promela could be used also for LTL model checking. Future work will be devoted to further optimization of the translation, trying to incorporate other techniques into Trio2Promela.

We have also shown that Trio2Promela can be used for satisfiability checking of large TRIO formulae. Therefore, we plan to assess the merit of satisfiability checking by means of a translation into Spin's Promela (as currently done by Trio2Promela) against the translation into the language of boolean satisfiability solvers, which are well-known for their efficiency in many practical cases.

References

1. Ghezzi, C., Mandrioli, D., Morzenti, A.: TRIO, a logic language for executable specifications of real-time systems. The Journal of Systems and Software **12** (1990) 107–123
2. Morzenti, A., San Pietro, P.: Object-oriented logical specification of time-critical systems. ACM Trans. Softw. Eng. Methodol. **3** (1994) 56–98
3. Morzenti, A., Mandrioli, D., Ghezzi, C.: A model parametric real-time logic. ACM Trans. Program. Lang. Syst. **14** (1992) 521–573
4. Felder, M., Morzenti, A.: Validating real-time systems by history-checking TRIO specifications. ACM Trans. Softw. Eng. Methodol. **3** (1994) 308–339
5. Gargantini, A., Morzenti, A.: Automated deductive requirements analysis of critical systems. ACM Trans. Softw. Eng. Methodol. **10** (2001) 255–307
6. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23** (1997) 279–295
7. Pradella, M., San Pietro, P., Spoletini, P., Morzenti, A.: Practical model checking of LTL with past. In: ATVA03. (2003)
8. Morzenti, A., Pradella, M., San Pietro, P., Spoletini, P.: Model checking TRIO specifications in Spin. In: FME 2003. Volume 2805 of LNCS. (2003) 542–561
9. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: CAV'01. Volume 2102 of LNCS. (2001) 53–65
10. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: CAV'00. Volume 1855 of LNCS. (2000) 248–263
11. Vardi, M.Y.: An automata-theoretic approach to linear temporal logic. In: Logics for concurrency : structure versus automata. Volume 1043 of LNCS. (1996) 238–266
12. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. J. ACM **28** (1981) 114–133
13. Fritz, C.: Constructing Büchi automata from linear temporal logic using simulation relations for alternating Büchi automata. In: CIAA 2003. Volume 2759 of LNCS. (2003) 35–48
14. Heitmeyer, C., Mandrioli, D., eds.: Formal Methods for Real-Time Computing. Volume 5 of Trends in Software. Wiley (1996)
15. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. **5** (1987) 1–11
16. Gastin, P., Oddoux, D.: LTL with past and two-way very-weak alternating automata. In: MFCS'03. Volume 2747 of LNCS. (2003) 439–448