The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications

Julien Brunel
David Chemouil
ONERA/DTIS & Université fédérale de Toulouse
France

Alcino Cunha
Nuno Macedo
INESC TEC & Universidade do Minho
Portugal

ABSTRACT

This paper presents the Electrum Analyzer, a free-software tool to validate and perform model checking of Electrum specifications. Electrum is an extension of Alloy that enriches its relational logic with LTL operators, thus simplifying the specification of dynamic systems. The Analyzer supports both automatic bounded model checking, with an encoding into SAT, and unbounded model checking, with an encoding into SMV. Instance, or counter-example, traces are presented back to the user in a unified visualizer. Features to speed up model checking are offered, including a decomposed parallel solving strategy and the extraction of symbolic bounds.
Source code: https://github.com/haslab/Electrum
Video: https://youtu.be/FbjlpvjgMDA

CCS CONCEPTS

• **Software and its engineering → Model checking**; **Specification languages**;

KEYWORDS

Formal specification language, model checking, model validation

ACM Reference Format:
Julien Brunel, David Chemouil, Alcino Cunha, and Nuno Macedo. 2018. The Electrum Analyzer: Model Checking Relational First-Order Temporal Specifications. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240475

1 INTRODUCTION

Software design validation and verification is a crucial task at early software development stages. For realistic software, different views of the design must be analyzed at different abstraction levels, and taking into consideration a variety of features and configurations. This calls for expressive formal specification languages, ideally supported by automatic reasoning tools that can quickly analyze models and specifications, and provide useful feedback to the user.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240475

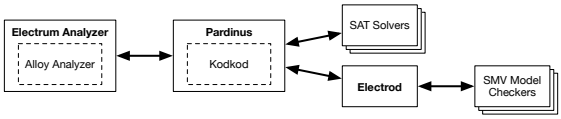

Figure 1: Architecture of the Electrum tool-chain.

When exploring the current panorama of formal methods for software design [7], namely surveying languages and tools that support both rich structural properties — expressed in some first-order logic flavor — and behavioral properties — expressed in temporal logic — we have found that adequate solutions were scarce. This led to the proposal of Electrum [7], an extension of the (purely static) Alloy specification language [5] with dynamic features loosely inspired by the temporal logic of actions of TLA$^+$ [6]. Electrum — a gold and silver alloy — combines the expressiveness of these two popular formal methods, and was conservatively designed with a focus on preserving the flexibility of Alloy while easing the specification (and subsequent analysis) of behavioral properties. Key to the success of these methods is the proper support for both model validation and verification. To this purpose, Electrum is backed by two automatic analysis engines: one for bounded model checking based on the original Alloy encoding into SAT — well-suited to quickly generate and explore alternative scenarios that conform to the model — and another for unbounded model checking through a translation into the SMV language [10] — which provides additional guarantees when checking correctness properties in later stages.

This paper reports recent developments in the Electrum language and presents the new Electrum Analyzer,[1] which fully integrates the bounded and unbounded engines (separate procedures in the first Electrum release), in a unified IDE that extends the Alloy Analyzer. The Alloy Analyzer analyzes and provides visual feedback about an Alloy specification, and is implemented on top of Kodkod [11], a relational model finder built over off-the-shelf SAT solvers. This architecture enables a clear separation of concerns: the Analyzer implements the syntax and semantics of Alloy, providing a text editor and an instance visualizer, while Kodkod focuses on the implementation of the automatic analysis procedures. Electrum preserves this architecture, depicted in Fig. 1, relying on the new stand-alone temporal relational model finder Pardinus[2] — kodkod's sturdier cousin — to support the analysis of problems provided by the Analyzer. This architecture also eases the independent exploration of language [1] and analysis [4, 8, 9] features, like target-oriented model finding, dynamic specifications and temporal formulas, unbounded model

[1] Available under the MIT license at https://github.com/haslab/electrum.
[2] Available under the MIT license at https://github.com/haslab/pardinus.

```
open util/ordering[Id]                    // establish total order on ids

sig Id {}
sig Process {
  id: Id,
  succ: Process,
  var toSend: set Id }
var sig Elected in Process {}

fact ring {                                  // force the ring topology
  id in Process lone → Id
  all p: Process | Process in p.^succ }

fact defElected {              // define the value of Elected at each step
  no Elected and always Elected' =
    { p: Process | after p.id in p.toSend and p.id not in p.toSend } }

pred init { all p: Process | p.toSend = p.id }     // define initial state

pred nop[p: Process] { p.toSend' = p.toSend }                // nop action

pred comm[p: Process] {                            // communication action
  some i: p.toSend {
    p.toSend' = p.toSend - i                // prevs[x] = all elements <  x
    p.succ.toSend' = p.succ.toSend + (i - prevs[p.succ.id]) } }

fact trace {       // traces derived from an initial state through actions
  init and always all p: Process | comm[p] or comm[p.~succ] or nop[p] }

pred Consistent { eventually some Elected }
run Consistent for 2 but 10 Time          // find a trace with an election

assert Safety { always lone Elected }
check Safety for 3 but 10 Time

pred progress { always (some toSend => some p: Process | comm[p]) }

assert Liveness { (some Process and progress) => eventually some Elected }
check Liveness for 3 but 10 Time     // check whether at least one elected
```

Figure 2: Ring leader election in the Electrum language.

finding, symbolic bounds, and a decomposed solving strategy. The unbounded engine relies on Electrod,[3] a stand-alone tool that compiles problems into SMV to be solved by symbolic model checkers (which can also perform bounded analysis). The plug-in architecture, inherited from Kodkod, allows the straightforward integration of future developments in SAT and SMV solving. Every procedure reports back to the Analyzer through the Pardinus API, resulting in a uniform representation of instances and counter-examples.

In summary, the main features of Electrum and its Analyzer, reported in this paper, are: *i*) a lightweight formal specification language with rich structural and behavioral constructs (Section 2); *ii*) integrated scenario exploration functionalities for visualizing and navigating alternative solution traces (Section 3); and *iii*) support for automatic bounded and unbounded model checking of specifications in first-order LTL with past (Section 4).

2 ELECTRUM SPECIFICATIONS

Electrum supports both structural and dynamic constructs, restricted by additional logical constraints, over which arbitrary temporal properties can be checked. The complete syntax can be consulted online,[4] while its formal semantics can be consulted in [7].

Structure is introduced through the declaration of *signatures*, that represent sets of uninterpreted elements, and *fields* of arbitrary arity that relate elements belonging to different signatures. Each signature and field can be declared as *static* (by default) or *variable* (keyword **`var`**): the former have the same valuation throughout the trace, while the latter may change over time. Hierarchy between signatures can be introduced through *extension* or *inclusion*, and signatures can additionally be declared as *abstract*. Finally, signatures and fields may be restricted by simple *multiplicity* constraints. For variable elements, these are applied locally at each state, e.g., two variable signatures extending another one are disjoint in each state, but may swap elements from one state to another.

Additional restrictions can be imposed through *facts*, logical constraints that must hold in every instance of the specification, which may rely on reusable *predicates* and *functions*. Relational expressions are built by composing signatures and fields (and some built-in constants) with common set-theoretic and relational operators like *join* `.` or transitive closure `^`. Expressions can be *primed*, referring to its valuation in the succeeding state. Atomic formulas are defined as inclusion (or equality) tests of relational expressions, which can be combined through the common Boolean operators, first-order quantifications and future and past LTL operators. Although past operators were not initially supported, and despite not actually improving the logical expressiveness, our experiments have shown that they often allow simpler and more succinct specifications.

Writing Electrum specifications tends to follow a pattern that builds on the dichotomy between static and variable elements, as exemplified by the model in Fig. 2 of a leader election algorithm for ring topologies.[5] The first step usually consists of the definition of the static structure (the network topology, established by field `succ`, that arranges the different `Process` elements identified by totally ordered `Id` elements), as well as facts that restrict what is considered a valid valuation for these elements (like `ring`, forcing the ring topology and unique identifiers). We usually refer to these as representing the *configurations* of the system, all of which will be explored by the analysis procedures but that remain frozen through a particular trace. Then, the variable elements are introduced (here, which processes are effectively considered `Elected`, represented by a variable sub-set of `Process`, and the tokens that are to be sent between processes, represented by variable field `toSend`). These can also be bound by temporal facts (like `defElected`, that forces `Elected` to be initially empty and defines it in each succeeding instant). To restrict the evolution of the system, state formulas (formulas without temporal operators or primed expressions, like `init`) and actions that restrict the system evolution (predicates that relate values from the current state with those from the succeeding one through primed variables, like `comm` and `nop`) can be defined. More complex actions may use arbitrary LTL operators. A fact (like `trace`) can then enforce that, globally, every state is reached by the application of actions starting at some initial predicate.

3 MODEL VALIDATION

Lightweight formal specifications are often used to validate design decisions prior to the employment of more advanced methods. In Electrum, analyses can be performed by *run* and *check* commands: the former instruct the Analyzer to search for instances that satisfy

[3] Available under the MPL 2.0 license at https://github.com/grayswandyr/electrod.
[4] https://github.com/haslab/Electrum/wiki/Language

[5] A tutorial with the step-by-step development of this example is available at https://github.com/haslab/Electrum/wiki/Tutorial.

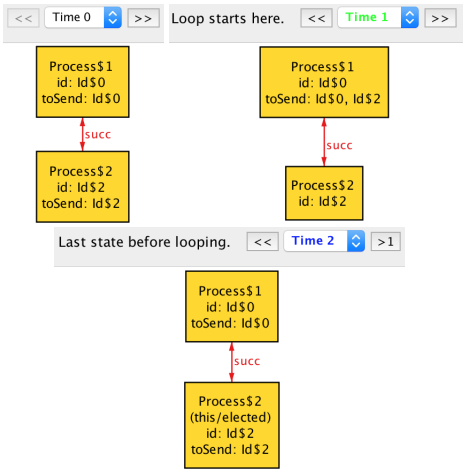

Figure 3: Example trace in the Electrum visualizer.

certain properties, the latter for counter-examples that break desirable properties. Run commands are typically defined as a first step towards the validation of the model by generating instances with interesting properties, such as `Consistent` in Fig. 2, that asks for a trace where a process is eventually elected leader using an LTL formula. Commands are always restricted by *scopes* that determine the maximum (or *exactly* the) number of elements of each signature that will be considered by the analyses. Bounded model checking is particularly useful at this stage due to its performance, providing quick feedback to the user; it is employed if a SAT solver or an SMV model checker with the bounded option is selected through the GUI. A protected scope **`Time`** restricts the size of the traces to be analyzed in this case. The `Consistent` command will search for instances with at most 2 elements per signature, but 10 **`Time`** steps. Due to the nature of the Electrum translation into LTL, all instances found by the commands encode infinite traces through looping states.

Computer-aided support for scenario exploration is critical to facilitate the detection of problematic instances or illustrative counter-examples of the desired properties. To ease this process, the bounded engine of Electrum allows the user to control the range of trace lengths that will be considered. By default, the **`Time`** scope determines the maximum length `n` that will be analyzed, resulting in a (seamless) iterative process that starts from length 1 up to `n`. The search can however be restricted to consider **`exactly`** a length `n` or even a range `n..m`, reducing the effort to find a relevant instance.

The Analyzer provides a visualizer that graphically presents instances found by run and check commands. This allows for the logic-agnostic, uniform visualization of traces generated by the bounded and unbounded engines, making them understandable for stakeholders without expertise in formal specification. It is essentially an adaptation of Alloy's visualizer (resembling the visualization of projected instances), with additional support for navigating infinite traces through looping states. The support for visualization themes is also preserved. Figure 3 depicts an instance found for the `Consistent` run command with 3 states looping back into the 2nd, as highlighted in the GUI. The evaluator, which performs queries over instances, has also been adapted to consider the selected state.

Once an instance is found, it is often useful to navigate through alternative instances that satisfy (or break) the specified property. The Alloy Analyzer, through Kodkod, supports basic navigation by relying on incremental SAT solvers and a powerful symmetry breaking algorithm, quickly generating unique instances in an arbitrary order. In Electrum, the SAT engine generates instances ensuring that the current target trace length will not be increased until that length is exhausted, providing a certain degree of predictability. At that point, seamlessly to the user, the length is increased (if it is still within the **`Time`** scope). The SMV engine also allows iteration, but limited to at most one instance per configuration. More advanced scenario exploration techniques are currently being studied.

4 MODEL VERIFICATION

After validation, the next natural step is to verify the correctness of the specification by model checking safety or liveness properties. Electrum supports *assertions* specified as arbitrary LTL properties, that can be verified, within a certain scope, by check commands. Such is the case of `Safety` in Fig. 2, that checks whether at most one process is elected leader at any given time, and `Liveness`, that checks whether a process will eventually be elected, for a universe with up to 3 processes. The latter will actually produce a counter-example unless an additional fact forces `progress`, again specified as arbitrary LTL formulas. Once properties are shown to hold by bounded analysis, the (less scalable) unbounded procedures can provide more confidence regarding the correctness of the model. This is done by selecting either unbounded NuSMV or nuXmv through the GUI. Scopes on **`Time`** will then be ignored as these check for traces with arbitrary length, although finite state spaces are still considered (i.e., still at most 3 processes are considered).

These procedures rely on Electrod, which converts the relational layer into plain first-order logic, which is then expanded into (propositional) LTL, depending on the possible valuations of sets and relations inferred from the Electrum model and the bounds on first-order domains. As in Kodkod, *lower* bounds and symmetries (computed by Pardinus) are leveraged to simplify the end result. The resulting SMV model features a representation of inferred system transitions as `TRANS` sections, and other inferred invariants and initial conditions. The default checking procedure in nuXmv [2] relies on the k-liveness algorithm (experience has shown that it is usually more efficient than NuSMV by at least an order of magnitude).

Several techniques have been studied to improve the scalability of the analyses. Pardinus implements a decomposed model finding strategy with symbolic bounds, that concurrently tries to find instances in tighter search spaces [9]. This strategy is particularly relevant for complex problems whose result is expected to be satisfiable, since often the concurrent smaller problems will find a solution faster. Symmetry breaking is properly preserved. This strategy can be selected from the GUI, either purely parallel or in a hybrid mode where the original analysis runs concurrently. Symbolic bounds are automatically inferred from the signature hierarchy and bounding expressions in field declarations. Currently, the problem is automatically partitioned between the structural and dynamic elements of the specification, although criteria based on the dependencies entailed by the symbolic bounds have shown to be more efficient in some scenarios [9], which we intend to implement on the Analyzer.

Table 1: Summary of the performed tests (time in seconds).

Spec/Command	S	n	MA	MH	BA	BH	XA	XH
spantree(1)	S	5	0.15	0.32	2.54	0.49	4.96	0.58
spantree(2)	U	5	1.76	4.66	4.30	4.53	30.05	31.07
spantree(3)	U	5	7.82	13.37	13.49	14.37	25.15	28.23
spantree(1)	S	6	0.18	0.32	11.28	0.59	15.77	0.71
spantree(2)	U	6	5.19	6.75	41.03	42.22	344.40	363.20
spantree(3)	U	6	45.30	47.72	130.09	133.53	512.55	502.01
ring(1)	S	4	0.14	0.26	0.48	0.37	3.38	0.43
ring(2)	U	4	1.02	1.70	0.82	0.92	5.80	5.82
ring(3)	U	4	2.36	2.95	3.10	3.06	210.63	204.87
ring(1)	S	5	0.15	0.34	1.01	0.40	3.82	0.49
ring(2)	U	5	2.38	4.17	6.53	6.81	40.07	39.96
ring(3)	U	5	8.75	10.04	26.90	26.46	t/o	t/o
javatypes(1)	S	3	0.18	0.38	0.76	0.72	3.75	0.86
javatypes(2)	U	3	3.21	6.23	t/o	t/o	157.60	121.85
javatypes(1)	S	4	0.19	0.48	3.26	0.78	7.67	0.82
javatypes(2)	U	4	109.69	109.85	t/o	t/o	t/o	t/o
hotel(1)	S	4	0.21	0.73	1.76	0.57	5.23	0.71
hotel(2)	S	4	0.42	1.41	1.55	1.70	8.61	8.79
hotel(3)	U	4	3.01	5.66	22.68	23.07	t/o	t/o
hotel(1)	S	5	0.27	1.25	11.45	0.61	22.18	0.75
hotel(2)	S	5	0.98	1.69	8.89	8.88	90.11	73.47
hotel(3)	U	5	13.09	17.48	t/o	t/o	t/o	t/o
firewire(1)	S	4	0.25	0.57	0.86	0.99	3.89	3.92
firewire(2)	U	4	5.91	8.64	4.60	4.49	t/o	t/o
firewire(3)	U	4	4.49	6.99	64.03	34.60	136.43	136.70
firewire(1)	S	5	0.29	0.44	2.11	2.15	6.56	6.21
firewire(2)	U	5	9.02	13.35	11.59	11.73	t/o	t/o
firewire(3)	U	5	9.15	10.83	28.79	340.57	423.59	179.86
dijkstra(1)	S	4	0.11	0.17	0.30	0.30	0.44	0.36
dijkstra(2)	U	4	0.62	0.97	0.45	0.45	1.00	0.95
dijkstra(3)	U	4	214.29	140.92	33.86	32.73	2.39	2.24
dijkstra(1)	S	5	0.12	0.15	0.30	0.32	0.48	0.36
dijkstra(2)	U	5	0.81	1.25	0.57	0.57	2.23	2.20
dijkstra(3)	U	5	t/o	t/o	t/o	t/o	7.59	7.44
ertms(1)	S	2	0.44	0.54	3.43	0.94	5.46	1.49
ertms(2)	S	2	85.65	53.39	t/o	243.60	t/o	217.89
ertms(3)	S	2	91.17	56.60	t/o	251.60	t/o	222.30
ertms(4)	U	2	34.94	35.16	79.30	79.01	72.93	71.28
ertms(1)	S	3	1.00	0.82	80.73	1.44	77.75	2.90
ertms(4)	U	3	291.63	307.22	t/o	t/o	t/o	t/o

5 EVALUATION

Electrum and associated tools are in active development and publicly available in open-source. Its plug-in architecture allows us to quickly support future improvements on SAT and SMV solvers. A repository of examples is available,[6] which includes the conversion of most dynamic Alloy models from the official distribution. Our successful answer [3] to the ABZ18 call for case study contributions pushed Electrum to its limits and is a fine testimony to the power of the language and tools. Electrum is also being used in an industrial collaboration to formally analyze some safety requirements of a platform screen door system for subways. The approach has allowed us to consider various implementation solutions, actually used in different countries, encompassed by a single model.

Specification languages as expressive as Electrum are scarce. Languages with support for first-order logic and dynamic behavior include TLA$^+$, the B method, and other dynamic extensions to Alloy (at varied stages of robustness). These formal methods are in general less flexible than Electrum, either failing to support rich structural constraints, declarative actions, or expressive temporal properties with future and past operators, resulting in a more cumbersome and verbose modeling process (see [7] for a detailed analysis).

[6] https://github.com/haslab/Electrum/wiki/Examples

Table 1 depicts the performance of the Analyzer for a set of models with both satisfiable (S) and unsatisfiable (U) commands (i.e., with and without solutions) and varying scope (n). They are available at the example repository, and the 1st column identifies the commands within each model. A scope 10 on **`Time`** was set in all bounded tests. All tests were run on a dual Intel Xeon E5-2699 with 36 threads and 512GB RAM, SAT-based ones using MiniSAT 2.2 (M) and SMV-based ones using nuXmv 1.1 in bounded (B) and unbounded (X) modes. Tests were run without (A) and with (H) the hybrid decomposed strategy with automatic decomposition on variable elements and symbolic bound extraction. Timeout was set at 10m. In non-decomposed SAT-based runs (MA), models are eventually converted into plain Kodkod with explicit state, as would pure Alloy specifications. Thus, they allow us to compare the performance of the Electrum Analyzer with that of the Alloy Analyzer.

As expected, unbounded procedures, in general, scale worse than bounded ones (despite some exceptions, like in `dijkstra(3)`), but also provide additional correctness guarantees. Bounded SMV procedures perform worse than SAT-based ones (again, despite the exception in `dijkstra(3)`). Regarding the employment of the decomposed strategy, most gains occur at SAT problems, as expected [9]. It is particularly relevant in unbounded procedures, many of which would timeout otherwise (like `ertms`). Also note that, even when the hybrid approach is outperformed, it is never by a factor higher than 1.5 for problems not solved in a few seconds.

Currently we are exploring advanced scenario exploration procedures tailored for temporal traces, and how to improve the performance of the analysis procedures, namely by extracting tighter symbolic bounds and implementing alternative partition criteria.

ACKNOWLEDGMENTS

Work financed by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation (COMPETE2020) and by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT) within project POCI-01-0145-FEDER-016826, and the French Research Agency project FORMEDICIS ANR-16-CE25-0007.

REFERENCES

[1] J. Brunel, D. Chemouil, A. Cunha, T. Hujsa, N. Macedo, and J. Tawa. 2018. Proposition of an Action Layer for Electrum. In *ABZ (LNCS)*, Vol. 10817. Springer, 397–402.
[2] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. 2014. The nuXmv Symbolic Model Checker. In *CAV (LNCS)*, Vol. 8559. Springer, 334–342.
[3] A. Cunha and N. Macedo. 2018. Validating the Hybrid ERTMS/ETCS Level 3 Concept with Electrum. In *ABZ (LNCS)*, Vol. 10817. Springer, 307–321.
[4] A. Cunha, N. Macedo, and T. Guimarães. 2014. Target Oriented Relational Model Finding. In *FASE (LNCS)*, Vol. 8411. Springer, 17–31.
[5] D. Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis* (revised ed.). MIT Press.
[6] L. Lamport. 2002. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley.
[7] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg. 2016. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *SIGSOFT FSE*. ACM, 373–383.
[8] N. Macedo, A. Cunha, and T. Guimarães. 2015. Exploring Scenario Exploration. In *FASE (LNCS)*, Vol. 9033. Springer, 301–315.
[9] N. Macedo, A. Cunha, and E. Pessoa. 2017. Exploiting Partial Knowledge for Efficient Model Analysis. In *ATVA (LNCS)*, Vol. 10482. Springer, 344–362.
[10] K. L. McMillan. 1993. *Symbolic Model Checking.* Kluwer.
[11] E. Torlak and D. Jackson. 2007. Kodkod: A Relational Model Finder. In *TACAS (LNCS)*, Vol. 4424. Springer, 632–647.