Proposition of an Action Layer for Electrum*

Julien Brunel[1], David Chemouil[1], Alcino Cunha[2],
Thomas Hujsa[1], Nuno Macedo[2], and Jeanne Tawa[1]

[1] ONERA/DTIS & Université Fédérale Toulouse Midi-Pyrénées, France
[2] INESC TEC & Universidade do Minho, Portugal

Abstract. Electrum is an extension of Alloy that adds (1) mutable signatures and fields to the modeling layer; and (2) connectives from linear temporal logic (with past) and primed variables à la TLA^+ to the constraint language. The analysis of models can then be translated into a SAT-based bounded model-checking problem, or to an LTL-based unbounded model-checking problem. Electrum has proved to be useful to model and verify dynamic systems with rich configurations. However, when specifying events, the tedious and sometimes error-prone handling of traces and frame conditions (similarly as in Alloy) remained necessary. In this paper, we introduce an extension of Electrum with a so-called "action" layer that addresses these questions.

1 Introduction

The specification and verification of software and systems are crucial tasks at early development phases. Indeed, the later the detection of an error happens in the development cycle, the more costly it is. This calls for expressive formal specification languages, ideally supported by automatic verification tools. Then, an important issue is the trade-off between the expressiveness of the specification language and the automation degree of the verification. Alloy [4], one of the main propositions in *lightweight* formal methods, does not favor one concern over the other. Instead, it gives up on the completeness of the verification: it performs an exhaustive exploration of the system states up to a user-specified depth.

Alloy is based on an extension of first-order logic and offers a rich way to specify structural properties over a system. In [5], we proposed Electrum, an extension of Alloy with support for dynamic features based upon linear temporal logic (LTL). Electrum preserves the flexibility of Alloy while easing the specification of behavioral properties and enabling verification over an unbounded temporal horizon. With Electrum, the system behavior is specified using FOLTL formulas. Electrum thus preserves the fully declarative feature of Alloy: there is no "constructive" description of the system, but only the constraints that the

* This work is financed by the ERDF - European Regional Development Fund - through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 - and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826, and the French Research Agency project FORMEDICIS ANR-16-CE25-0007.

system satisfies. However, in practice, it is often convenient to specify the basic *actions* of the system (which needs little expressiveness in terms of temporal logic) separately from other behavioral requirements, such as the way these actions are ordered (which may need the full expressive power of temporal logic). Relying on this kind of idioms can have several advantages: (1) some part of the behavior, such as the frame conditions or the time model, can be specified in a systematic way; (2) such a description of the evolution of the system is more likely to be exploited by a verification procedure that relies on a model checker.

Thus, still pursuing the goal of allowing the straightforward specification and verification of models featuring rich structure and behavior, we propose here an extension of Electrum with an action layer.

The remainder of the article is organized as follows. In § 2 we present the Electrum framework. In § 3, we define the syntax and semantics of the action layer and illustrate it on an example.

2 Electrum

Following Alloy, structure in an Electrum specification is introduced through the declaration of *signatures*, which represent sets of uninterpreted atoms, and *fields* of arbitrary finite arity, which relate atoms belonging to different signatures. Each of these signatures and fields can be declared as *static* (by default) or *variable* (keyword `var`): the former have the same valuation throughout a given time trace, while the latter are mutable and hence may evolve in time. Hierarchy between signatures (which can additionally be declared as `abstract`) can be introduced through *extension* (`extends` keyword) or *inclusion* (`in`). Finally, both signatures and fields may be restricted by simple *multiplicity* constraints. Notice that for variable elements, these restrictions are applied globally in time.

Additional restrictions can be imposed through *facts*, axioms that every instance of the specification is required to conform to. Those 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 operators and relational operators like *join* `.` or transitive closure `^`.

Every relational expression can be *primed*, referring to its valuation in the succeeding state. Atomic formulas are then built as inclusion (or equality) tests of relational expressions, which can be composed through the common Boolean operators, first-order quantifications and future and past LTL operators.

Execution instructions consist of `run` and `check` commands restricted by *scopes* that determine the maximum (or `exactly` the) number of atoms of each signature that will be considered by the analyses: (1) `run` instructs the Analyzer to search for an instance satisfying a given constraint; (2) `check` instructs the Analyzer to prove a given assertion valid (in practice: by checking that it cannot find a counter-example). A protected keyword `Time` restricts the size of the traces when analysis is performed by *bounded* model checking (BMC). Note that *unbounded* model checking (UMC) is still bounded on the atoms in the valuations of signatures. The complete semantics of Electrum can be consulted in [5].

```
open util/ordering[Key]
sig Key {}
sig Room {
  keys: set Key,
  var current: one keys }
fact DisjointKeySets {
  Room<:keys in Room lone→ Key }
one sig Desk {
  var lastKey: Room → lone Key,
  var occupant: Room → Guest }
sig Guest { var gkeys: set Key }
...
fun nextKey[k: Key, ks: set Key] : set Key {
  min[nexts[k] & ks] }

act checkin[g: Guest, r: Room, k: Key]
  modifies gkeys, occupant, lastKey {
  no r.(Desk.occupant)
  k = nextKey[r.(Desk.lastKey), r.keys]
  gkeys' = gkeys + g → k
  Desk.occupant' = Desk.occupant + r→g
  Desk.lastKey' = Desk.lastKey ++ r→k }
act checkout[g: Guest] modifies occupant {
  some Desk.occupant.g
  Desk.occupant' = Desk.occupant − Room→g }
...

fact init {
  no Guest.gkeys
  no Desk.occupant
  all r: Room | r.(Desk.lastKey) = r.current }

pred consistent {}
run consistent for 4 but 10 Time
assert BadSafety  {
  always { all r: Room, g: Guest, k: Key |
    entry[g, r, k] and
    some r.(Desk.occupant)
      ⇒ g in r.(Desk.occupant) } }
check BadSafety for 4 but 10 Time
```

Fig. 1. Hotel example in Electrum with actions (syntax additions are underlined).

3 Extending Electrum with Actions

In this section, we present the syntax and semantics of the action layer. The layer is actually syntactic sugar on top of plain Electrum, therefore the semantics is defined by translation into Electrum. For the sake of readability, we illustrate this translation over an example (Fig. 1) inspired by the classic Alloy Hotel example. The latter specifies a system handling entries in the rooms of a hotel with disposable key-cards carrying cryptographic keys that must match other keys stored in room-door locks to release these and open the rooms.

Specifying behavior in Electrum is completely unrestricted. However, *in practice* (and as the Hotel example shows), many models are specified using actions (represented as predicates or using the event idiom [4]) that only relate two consecutive instants. Said otherwise, a large class of Electrum models does not rely on the full power of LTL to specify the valid traces: this logic is mainly useful when specifying additional facts (*e.g.* fairness properties) or stating properties to evaluate on a model using a `run` or a `check` command. Besides, the frame conditions and the time model could be described in a systematic way. Their generation could be automated, depending only on a few parameters (*e.g.* allowing, or not, simultaneous actions).

In practice, we add to plain Electrum an action syntactic sugar that is optional but committing: if no action is present in a model, then its semantics is fully unrestricted, as usual, but as soon as an action is present, the semantics associated with actions applies. The sugar thus introduces a notion of action. Frame conditions are automatically generated out of a specific parameter of actions. Traces are automatically generated, forcing a specific time model. Finally, the occurrence of an action can be referred to in the syntax of constraints.

Actions and time model We add an `act` keyword that introduces a named action, possibly with parameters. Parameters may only be singletons (*i.e.*, no set may be passed as an argument: if this is needed, then a new signature pointing to the said set should be introduced first and then passed as an argument). An action executes *atomically* and relates two consecutive instants, therefore the only temporal constructs allowed are the `after` keyword and the prime operator. As in plain Electrum, formulas (from the action body) relating to the "current" instant represent a *guard* (necessary condition) for the action to occur, and the ones talking about the next instant stand for the post-condition.

The most important semantic constraint in our action layer is that the time model imposes an *interleaving semantics*: *exactly one action is executed at every instant*. Also, it does not feature stuttering steps by default: if this is needed, the user may define an *ad hoc* action: `act skip {}` (with an empty `modifies` clause).

Actions are translated into a structure of signatures and fields encoding the possible *events* (action occurrences). We introduce first an `_Action` enumeration for all action *names*. Then we add a relation encoding all possible events by taking the union of all possible valuations of actions (as actions may differ in arity, we pad them to the highest arity with a dummy signature). Simultaneously, we specify the time model by forcing exactly one event to occur at every instant. Finally, a fact states the effect of every action when it is `fired` (cf. p. 4):

```
enum _Action { checkin, checkout, ... } // action names
one sig _Dummy {}
one sig _E { // simply an enclosing signature for _event
  var _event:   (checkin → Guest → Room → Key)
              + (checkout → Guest → _Dummy → _Dummy)
              + ... // other possible events
  } { one _event } // time model
fact { always { // effect of every action when it is fired:
  all g : Guest, r : Room, k : Key {
    fired[checkin, g, r, k] implies ... /* checkin body */ }
  ... /* the same for other actions */ } }
```

Frame Conditions An action can specify, using a `modifies` clause, which variable signatures and fields it *controls*. In practice, this allows the automatic generation of frame conditions under a simple rule saying that *any variable signature or field that is not controlled by an action is left unchanged by this action*. *E.g.*, the `checkout` action (see Fig. 1 l. 23–25) controls the `occupant` field only, inducing that `current`, `lastKey` and `gkeys` do not change when this action fires. On the other hand, notice every action is responsible for handling the frame conditions for the variable constructs declared in its `modifies` clause.

Referring to Actions in Constraints Any occurrence of an action can be referred to in a constraint, with actual parameters (*e.g.* as `entry[g,r,k]` in Fig. 1 l. 37) or without (in which case, there is an implicit existential quantification over all parameters). For instance, `after checkout` actually means:

`after (some g: Guest | checkout[g])`. To allow this, we generate a `fired` predicate saying whether an action is indeed fired. As actions may take parameters of different types, the `fired` predicate profile accepts arguments in the union of all these types. Again, its arity is the highest arity for actions.

```
var sig _Arg = _Dummy + Guest + Room + Key {} // union of all types
pred fired [a : _Action, x1, x2, x3 : _Arg] { // if max arity = 3
    a→x1→x2→x3 in _E._event }
```

This way, `entry[g,r,k]` (Fig. 1 l. 37) translates to `fired[entry, g, r, k]`.

4 Related Work and Conclusion

TLA$^+$ inspired Electrum in general, and its action layer in particular. However, significant differences between TLA$^+$ and plain Electrum have already been pointed out in [5]. Moreover, our proposition slightly differs as Electrum is stuttering sensitive and the time model is forced. The enhancement of Alloy with behavior [3,1,6,7,2] has been widely studied. Among these propositions, DynAlloy [3] defines a syntax for actions similar to ours, but the semantics differs in the time model and in the firing of actions. Besides, all these frameworks propose in the end a translation into plain Alloy and thus, they only offer verification over a bounded temporal horizon. In our experience, using the Electrum action layer makes the behavior specification both easier (specifying the actions, and reasoning about their occurrence, is quite natural) and less error-prone because part of the behavior specification is automatically generated. We benchmarked (not shown due to lack of space) the action layer on examples coming from the Alloy literature: w.r.t. plain Electrum, the efficiency of analyses is often reduced for valid properties, but still acceptably. In the future, we intend to assess several new compilation strategies (and perhaps semantics) to improve the efficiency.

References

1. Chang, F.S., Jackson, D.: Symbolic model checking of declarative relational models. In: ICSE 2006. pp. 312–320. ACM (2006). DOI: 10.1145/1134329
2. Cunha, A.: Bounded model checking of temporal formulas with Alloy. In: ABZ 2014. LNCS, vol. 8477, pp. 303–308. Springer (2014). DOI: 10.1007/978-3-662-43652-3_29
3. Frias, M.F., Galeotti, J.P., Pombo, C.L., Aguirre, N.: DynAlloy: upgrading Alloy with actions. In: ICSE 2005. pp. 442–451. ACM (2005). DOI: 10.1145/1062455.1062535
4. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, revised edn. (2012)
5. Macedo, N., Brunel, J., Chemouil, D., Cunha, A., Kuperberg, D.: Lightweight specification and analysis of dynamic systems with rich configurations. In: SIGSOFT FSE. pp. 373–383. ACM (2016). DOI: 10.1145/2950290.2950318
6. Near, J.P., Jackson, D.: An imperative extension to alloy. In: ABZ 2010. LNCS, vol. 5977, pp. 118–131. Springer (2010). DOI: 10.1007/978-3-642-11811-1_10
7. Vakili, A., Day, N.A.: Temporal logic model checking in Alloy. In: ABZ 2012. LNCS, vol. 7316, pp. 150–163. Springer (2012). DOI: 10.1007/978-3-642-30885-7_11