Alloy meets TLA$^+$: An exploratory study

Alcino Cunha and Nuno Macedo

HASLab — High Assurance Software Laboratory
INESC TEC & Universidade do Minho, Braga, Portugal

Abstract. Alloy and TLA$^+$ are two formal specification languages that have gained increased acceptance due to their simplicity and flexibility, as well as the effectiveness of their companion model checkers, the Alloy Analyzer and TLC, respectively. Nonetheless, while TLA$^+$ focuses on temporal properties, Alloy is better suited to handle structural properties, reasoning about temporal properties requiring ad hoc mechanisms. This imposes limitation on the analysis of systems rich in both structural and behavioral properties. This paper explores the pros and cons of these two techniques when handling this class of systems.

1 Introduction

Software specification is crucial at early development phases since it encourages the developer to reason about the system and its properties, promoting the detection of design errors. Although a variety of frameworks has been proposed to aid the developer in this task, the most successful are characterized by two features: they provide a simple yet expressive and flexible formal language—allowing the user to specify different classes of systems and properties at different abstraction levels—and are accompanied by tools that automate their analysis—providing quick feedback regarding the correctness of the specification.

Since the verification of properties is one of the main goals when specifying a system, it is important to be aware of the class of properties that are expressible in the specification language and manageable by the provided tools. Two classes of properties are particularly important at the specification stage: *structural* properties, typically defined in first-order logic, that define when the system is considered well-formed, and *behavioral* properties, typically defined in temporal logic, that define how the system is allowed to evolve. Although not necessarily in equal measure, most interesting systems will require the analysis of properties from these two classes: these [*Structural/Dynamic*] problems are the main focus of this study[1].

Alloy [4] is a lightweight formal specification language with an object-oriented flavor, which, paired with its Analyzer that provides support for both bounded model checking and model finding, has been increasingly adopted by software engineering practitioners. Alloy's underlying formalism is *relational logic*, first-order logic enhanced with transitive closures operations, that render the definition

[1] [Need to find a suitable name to represent this class of problems.]

of structural properties extremely simple. Alloy is inherently static, thus the verification of behavioral properties usually relies on well-known idioms that have emerged due to the language's flexibility. Nonetheless, such *ad hoc* specification is error-prone, and forces the developer to be concerned with particularities of the idiom rather than with the properties that he actually wishes to verify. As a consequence, considerable research has been dedicated to enhance Alloy with dynamic behavior [3,1,6,7,2]. The main drawback of these approaches is that they compromise the flexibility that the Alloy users are accustomed to, introducing syntactic extensions that force them into following specific idioms.

In contrast, temporal model checkers are developed with that specific goal in mind. Among the most successful formalisms is the *temporal logic of actions* (TLA) [5], a variant of temporal logic that introduces the notion of *action* to model the evolution of the system. Actions are essentially predicates that relate two consecutive states, and can be used to model the acceptable steps that allow the system to evolve. The TLA$^+$ specification language, built over this notion, has proven to be well suited to specify such class of systems. Moreover, TLA$^+$ is accompanied by a set of effective tools, including TLC, a model checker that has proven effective on the verification of complex TLA$^+$ systems. While TLA$^+$ does support the specification of first-order logic properties, these are not the main focus of the framework, giving rise to some limitations that would be trivially addressed in Alloy.

Thus, although the popularity of both Alloy and TLA$^+$ arose due to similar factors—powerful but simple languages associated with effective and automated tools—they excel in the verification of different classes of properties. Since in this paper we aimed at specifying [*Structural/Dynamic*] systems, with rich structural and behavioral properties, none of these two frameworks rises as the ideal solution. We advocate that identifying the similarities and disparities of these two frameworks and whether they can benefit from one another would help developers specify and verify this kind of systems. Technically, since TLC, in contrast to the Analyzer, supports unbounded model checking, the Alloy user would benefit from an automatic translation of the specifications into TLA$^+$. Thus, in this preliminary work we explore the potential of embedding dynamic Alloy specifications in TLA$^+$. Since this study is developed from the perspective of an Alloy user exploring the TLA$^+$'s capabilities, basic knowledge of the Alloy language is expected from the reader, but not of TLA$^+$ and TLC.

The remainder of this paper is structured as follows. Section 2 presents the Alloy example that will be used throughout the paper. Section 3 presents our embedding of this example in TLA$^+$ and our experience with TLC. Section 4 presents a preliminary comparison of the Analyzer and TLC performance for this example. Finally, Section 5 draws conclusions regarding this exploratory study.

2 An Alloy Specification: Hotel Room Locking System

The example that will be used throughout this paper models a hotel room locking system, which was initially presented by Jackson [4, p. 187]. This system is built

on the assumption that the key management system at the front desk of the hotel is disconnected from the locking systems of the room doors. The door locking system only opens the door for the currently registered key, unless it detects a more recent one—issued by the front desk—at which point the older keys are rendered obsolete. Structural properties in this example regards the consistent state of the front desk and room doors systems, while behavioral properties model how these states evolve as guests check in and out. The remainder of this section explores a specification of this system in Alloy, which is depicted in Fig. 1. This specification is similar to the original one from [4], apart from small styling changes.

2.1 Structural Properties

In Alloy, the basic structure of the specification is defined by *signatures* and *fields* contained within. Each hotel locking system instance consists of a set of disposable keys (signature `Key`), rooms (signature `Room`) and guests (signature `Guest`). Each room has a pool of keys assigned to it (field `keys`), and keeps track of the last valid key that was used to open the door (field `currentKey`), which must belong to the pool. A front desk (signature `FD`) keeps track of each room occupants (field `occupant`) and the last key delivered for each room (field `lastKey`).

Signature and field declarations may not able to completely define a system, and thus additional *facts*—constraints that must hold for every valid instance—may be defined. In this case, fact `DisjointKeySets` forces each key to belong to the key pool of a single room. Finally, in order for the locking systems to recognize fresh keys, an order must be imposed over such elements. In Alloy this can be performed by setting atoms of `Key` as totally ordered. Function `nextKey` can then retrieve a fresh key from the pool of available ones.

2.2 Behavioral Properties

Since Alloy does not natively support temporal logic, some well-known idioms have been developed for modeling dynamic behavior. This example follows the *local state idiom*. Here, a totally ordered signature `Time` is introduced whose elements denote instants in time. Fields that are expected to vary in time (and signatures, although there are no such signatures in this example) are appended with a `Time` element that denotes its valuation in each moment. In this example, fields `currentKey`, `lastKey`, `occupant` and `gKeys` are expected to be variable. To access the valuation of a field in an instant `t` from `Time`, one simply composes the field with that element, e.g., `currentKey.t`.

Since in this example all the signatures are static, no additional facts are required to manage the temporal consistency of the fields. That is not the case in general. For instance, if signature `Key` was variable (modeling the fact that keys are only known to the system as they are assigned to guests), one would need to guarantee that the valuation at each instant of variable fields relating keys only related keys that actually exist in that same instant.

```
open util/ordering[Time] as to
open util/ordering[Key] as ko

sig Key {}
sig Time {}

sig Room {
  keys: set Key,
  currentKey: keys one -> Time }

fact DisjointKeySets {
  keys in Room lone -> Key }

one sig FD {
  lastKey: (Room -> lone Key) -> Time,
  occupant: (Room -> Guest) -> Time }

sig Guest {
  gKeys: Key -> Time }

fun nextKey [k: Key, ks: set Key]: set Key {
  min [k.nexts & ks] }

pred init [t: Time] {
  no Guest.gKeys.t
  no FD.occupant.t
  all r: Room | FD.lastKey.t [r] = r.currentKey.t }

pred entry [t, t': Time, g: Guest, r: Room, k: Key] {
  k in g.gKeys.t
  k = r.currentKey.t or k = nextKey[r.currentKey.t, r.keys]
  r.currentKey.t' = k
  all r: Room - r | r.currentKey.t = r.currentKey.t'
  all g: Guest | g.gKeys.t = g.gKeys.t'
  FD.lastKey.t = FD.lastKey.t'
  FD.occupant.t = FD.occupant.t' }

pred checkout [t, t': Time, g: Guest] {
  some FD.occupant.t.g
  FD.occupant.t' = FD.occupant.t - Room -> g
  FD.lastKey.t = FD.lastKey.t'
  all r: Room | r.currentKey.t = r.currentKey.t'
  all g: Guest | g.gKeys.t = g.gKeys.t' }

pred checkin [t, t': Time, g: Guest, r: Room, k: Key] {
  g.gKeys.t' = g.gKeys.t + k
  no FD.occupant.t [r]
  FD.occupant.t' = FD.occupant.t + r -> g
  FD.lastKey.t' = FD.lastKey.t ++ r -> k
  k = nextKey [FD.lastKey.t [r], r.keys]
  all r: Room | r.currentKey.t = r.currentKey.t'
  all g: Guest - g | g.gKeys.t = g.gKeys.t' }

fact traces {
  init [first]
  all t: Time, t' : t.next | some g: Guest, r: Room, k: Key |
      entry [t, t', g, r, k] or checkin [t, t', g, r, k] or checkout [t, t', g] }

assert NoBadEntry {
  all t: Time, t': t.next, r: Room, g: Guest, k: Key |
    entry [t, t', g, r, k] and some FD.occupant.t[r] => g in FD.occupant.t[r] }

fact NoIntervening {
  all t: Time, t': t.next, t'': t'.next, g: Guest, r: Room, k: Key |
    checkin [t, t', g, r, k] => (entry [t', t'', g, r, k] or no t'') }

check NoBadEntry for 3 but 5 Time
```

Fig. 1: Hotel room locking system under standard Alloy.

Next, one must specify the valid actions under which the state of the system may evolve, which in this case includes guests checking in and out, and entering the assigned rooms. Actions are specified as predicates between two instants of time (and possibly other parameters), much like TLA$^+$ actions. When a guest checks in, the next key for the chosen room is given, and the guest is added to the room's occupants (predicate `checkin`). The room locking system is unaware of such assignment: only when a more recent key than the one currently known is used to open the door is the system updated, rendering older keys obsolete in the process (predicate `entry`). The owner of the current key is also allowed to enter the room. Checking out vacates the room but allows the guest to keep the room key (predicate `checkout`). Valid systems states evolve under these actions from an initial state where all rooms are vacant (predicate `init`).

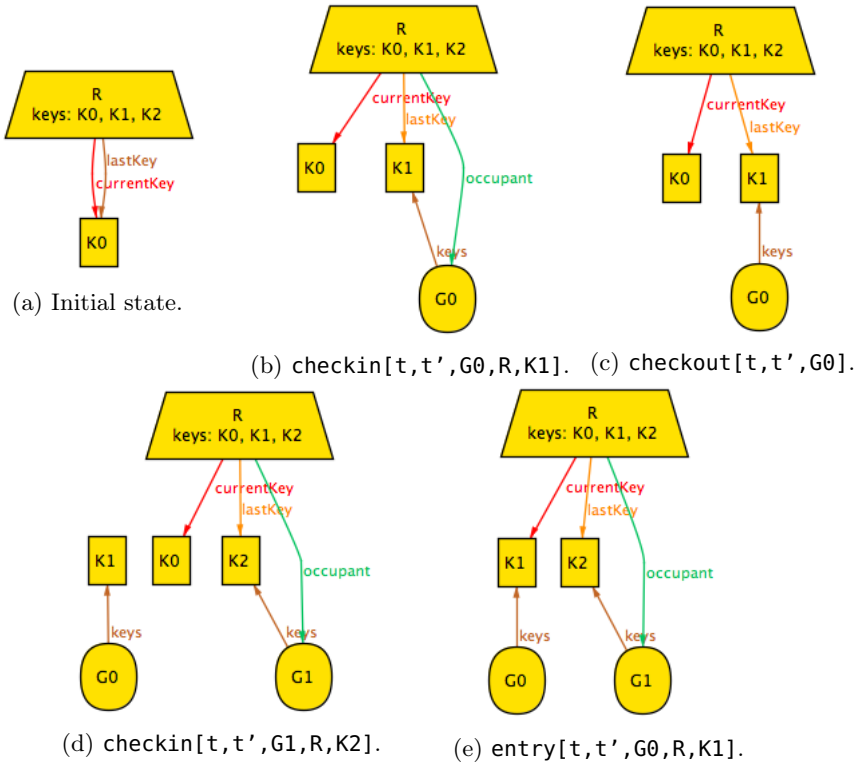
To actually force the system to behave according to the defined predicates, the fact `trace` must be introduced that forces `init` to hold in the first instant and every succeeding state to be derived from the action predicates. Once this fact is introduced, every instance generated by the Analyzer will consist of a trace over `Time` elements, whose steps arise from action application.

2.3 Safety Assertion

To validate the specification, the Alloy Analyzer can be instructed to check assertions or find instances that preserve a certain property. In this case, we are interested in checking whether a guest is able to enter a room after another guest has checked in into that room. This is modeled by the `NoBadEntry` assertion, which states that a guest entering a non-vacant room must be the room's occupant. Once the assertion is defined, the Analyzer is instructed to check it under a predefined scope, in this case for 3 elements of each signature and a trace with length 5. Since the initial version of this specification does not guarantee `NoBadEntry`, the Analyzer will quickly generate counter-examples. One such instance is depicted in Fig. 2 (atoms names are abbreviated to their initials for readability purposes).

It is important to be aware that with the Alloy Analyzer one can only perform bounded model checking, since we must impose a limit on the size of traces, which may lead to unpredictable behaviors. The most obvious one, is that the counter-examples may not be found within the provided scope. For instance, in this example, a scope of less than 5 instants would not flag the inconsistency, leading the user to a false sense of safety. A more subtle problem occurs when checking liveness properties—in contrast with safety properties like `NoBadEntry`. In such cases, the finite trace may lead to the detection of false positives. The turnaround to this issue is to force traces to simulate infinite traces by introducing a loop, disregarding non-infinite traces [2]. This guarantees that, at least within the provided bounds, there are no false positives.

Another particularity of this specification is that, since a total order is imposed on `Time`, the number of `Time` atoms is always exact to the defined scope. This means that the check command from Fig. 1 will only consider models with exactly 5 `Time` atoms, rather than models with *up to* 5 `Time` atoms. Since there is no



(a) Initial state.

(b) `checkin[t,t',G0,R,K1]`. (c) `checkout[t,t',G0]`.

(d) `checkin[t,t',G1,R,K2]`.

(e) `entry[t,t',G0,R,K1]`.

Fig. 2: Alloy counter-example for `NoBadEntry`.

guarantee that counter-examples occurring in smaller traces will reappear in larger ones, the user would have to manually check the system for increasing `Time` scopes. Alternatively, the user could define his own notion of total order that did not force the exact scope. However, this would have great impact on the performance of the Analyzer, since the native operation is highly optimized internally (which justifies the imposition of the exact scope).

Looking at Fig. 2, the `NoBadEntry` predicate breaks because guest $G0$ checks out without having ever entered the room, and thus the door locking system regards his key $K1$ as the next valid key. When guest $G1$ checks in into that room, $G0$ is still be able to enter the room with $K1$, breaking `NoBadEntry`. A possible fix for this issue is to force guests to enter the room right after checking in: in the counter-example, $G1$ would immediately enter the room with his $K2$ key, rendering $K1$ from $G0$ invalid. Such constraint was encoded as the `NoIntervening` fact, that forces the step succeeding a check in to be an entry action. Under this specification, the Analyzer is no longer able to find a counter-example for `NoBadEntry`.

3 TLA$^+$ Embedding

While TLA$^+$ is an extremely expressive specification language, the TLC model checker is not able to process arbitrary TLA$^+$ specifications and imposes restrictions to the language [5, p. 230]. Thus, there are two classes of issues that must be addressed throughout this embedding: first, the mismatch between the Alloy and the TLA$^+$ languages; second, the additional restrictions imposed by TLC for checking properties. A possible embedding of the hotel locking system in TLA$^+$ is depicted in Fig. 3. This specification is TLC-compatible, and can be deployed under the configuration presented in Fig. 4[2]. This section thoroughly explores this embedding. While the required TLA$^+$ concepts are presented when required, the interested user is redirected to [5] for a thorough presentation.

3.1 Module Parameters

Alloy signatures and fields take the shape of the module's parameters in TLA$^+$, which can be either set as *constant* or *variable*. This differs from the object-oriented flavor of Alloy, where fields are always associated to a parent signature. As a consequence, the `currentKey` and `occupant` fields no longer need to belong to the placeholder signature `FD`, which is not specified in the TLA$^+$ version.

It could be expected that the non-variable components of the specification would be represented by constants in the TLA$^+$ specification, which in this example would include the signatures `Key`, `Room` and `Guest` as well as the field `keys`. However, when TLA$^+$ modules are processed by TLC, the user is expected to assign a fixed valuation to constant parameters prior to the deployment. This notion differs fundamentally from the notion of non-variable fields in Alloy: although fields like `keys` do not evolve in time, the Alloy Analyzer is free to explore their valuation at the initial state. To model this behavior in TLC, field `keys` must still be specified as a variable parameter, whose valuation is always preserved by the actions.

This requirement to bind constant parameters *a priori* may lead to other unpredictable behaviors, which are manifest in the hotel locking system example. The specification entails that there is always at least a key assigned to each room's pool `keys`; the counter-example for the `NoBadEntry` assertion (Fig. 2) arose from the successive checking in of two guests in the same room, to whom two fresh keys were assigned. This means that the problem only arises in universes where there are at least two extra keys. In general, this is not problematic in Alloy, because executing `check NoBadEntry for 3` only indicates that the maximum of atoms per signature is 3 (unless the user defines an exact scope or defines a total order over them, as in signature `Key`)[3]. Thus, it will still check for scenarios

[2] The TLA tools are currently deployed as the *TLA Toolbox* (`http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html`), where the TLC parameters are defined through the GUI rather than as a configuration file.

[3] Technically, all the atoms are created in the underlying Kodkod, but may not be assigned to the signatures.

MODULE *HotelExactScope*

EXTENDS *Naturals*
CONSTANT *Key*, *Room*, *Guest*
ASSUME $Key \in \text{SUBSET } Nat$
VARIABLE *keys*, *currentKey*, *lastKey*, *occupant*, *gKeys*

$$
\begin{aligned}
TypeInv \triangleq\ & \land keys \in [Room \to \text{SUBSET } Key] \\
& \land currentKey \in [Room \to Key] \\
& \land \forall\, r \in Room : currentKey[r] \in keys[r] \\
& \land \forall\, r1,\, r2 \in Room : (keys[r1] \cap keys[r2]) \neq \{\} \Rightarrow r1 = r2 \\
& \land lastKey \in [Room \to Key] \\
& \land occupant \in [Room \to \text{SUBSET } Guest] \\
& \land gKeys \in [Guest \to \text{SUBSET } Key]
\end{aligned}
$$

$$
\begin{aligned}
Init \triangleq\ & \land keys \in [Room \to \text{SUBSET } Key] \\
& \land currentKey \in [Room \to Key] \\
& \land \forall\, r \in Room : currentKey[r] \in keys[r] \\
& \land \forall\, r1,\, r2 \in Room : (keys[r1] \cap keys[r2]) \neq \{\} \Rightarrow r1 = r2 \\
& \land lastKey = currentKey \\
& \land occupant = [r \in Room \mapsto \{\}] \\
& \land gKeys = [g \in Guest \mapsto \{\}]
\end{aligned}
$$

$$vars \triangleq \langle keys,\, currentKey,\, lastKey,\, occupant,\, gKeys \rangle$$

$$
\begin{aligned}
nextKey[k \in Key,\, ks \quad \in \text{SUBSET } Key] \triangleq\ & \text{LET } nexts \triangleq ks \setminus (0\,..\,k) \\
& \text{IN} \quad \{x \in nexts : \forall\, y \in nexts : x \leq y\}
\end{aligned}
$$

$$
\begin{aligned}
Entry(g,\, r,\, k) \triangleq\ & \land (k = currentKey[r] \lor \{k\} = nextKey[currentKey[r],\, keys[r]]) \\
& \land k \in gKeys[g] \\
& \land currentKey' = [currentKey \text{ EXCEPT } ![r] = k] \\
& \land \text{UNCHANGED } \langle keys,\, lastKey,\, occupant,\, gKeys \rangle
\end{aligned}
$$

$$
\begin{aligned}
Checkout(g) \triangleq\ & \land \exists\, r \in Room : g \in occupant[r] \\
& \land occupant' = [r \in \text{DOMAIN } occupant \mapsto occupant[r] \setminus \{g\}] \\
& \land \text{UNCHANGED } \langle keys,\, lastKey,\, currentKey,\, gKeys \rangle
\end{aligned}
$$

$$
\begin{aligned}
Checkin(g,\, r,\, k) \triangleq\ & \land occupant[r] = \{\} \\
& \land \{k\} = nextKey[lastKey[r],\, keys[r]] \\
& \land occupant' = [occupant \text{ EXCEPT } ![r] = @ \cup \{g\}] \\
& \land gKeys' = [gKeys \text{ EXCEPT } ![g] = @ \cup \{k\}] \\
& \land lastKey' = [lastKey \text{ EXCEPT } ![r] = k] \\
& \land \text{UNCHANGED } \langle keys,\, currentKey \rangle
\end{aligned}
$$

$$Next \triangleq \exists\, g \in Guest : Checkout(g) \lor \exists\, r \in Room,\, k \in Key : Entry(g,\, r,\, k) \lor Checkin(g,\, r,\, k)$$

$$Spec \triangleq Init \land [Next]_{vars}$$

$$
\begin{aligned}
& NoBadEntry \triangleq \Box[\forall\, g \in Guest,\, r \in Room,\, k \in Key : \\
& \quad Entry(g,\, r,\, k) \land occupant[r] = \{\} \Rightarrow g \in occupant[r]]_{vars}
\end{aligned}
$$

$$
\begin{aligned}
& NoIntervening \triangleq [\forall\, g \in Guest,\, k \in Key,\, r \in Room : \\
& \quad (g \in occupant[r] \land k \in gKeys[g] \land lastKey[r] = k \land currentKey[r] \neq k) \Rightarrow Entry(g,\, r,\, k)]_{vars}
\end{aligned}
$$

Fig. 3: The TLA$^+$ embedding of the hotel locking system.

```
INIT Init
NEXT Next
CONSTRAINT TypeInv
PROPERTY NoBadEntry
ACTION_CONSTRAINT NoIntervening
CONSTANTS Key = {1,2,3,4,5,6}
          Guest = {g1,g2,g3}
          Room = {"r1","r2","r3"}
```

Fig. 4: TLC configuration file.

with 1 room and 3 keys, finding the counter-example. In TLC however, the user assigns the exact valuation to the constant parameters. This means that, while the configuration from Fig. 4 does find the counter-example, a more distracted user could have specified `Key = 1,2,3` and `Room = r1,r2,r3`, in which case TLC would not find any counter-example.

To circumvent this issue, the user would have to manually specify additional variable parameters *key*, *guest* and *room*, and force them to be within the respective constants in the initial predicate as

$$\begin{aligned} &\wedge\, key \in \text{SUBSET } Key \\ &\wedge\, guest \in \text{SUBSET } Guest \\ &\wedge\, room \in \text{SUBSET } Room \end{aligned}$$

and then have the actions preserve their valuation (by appending these variables to the UNCHANGED expression). This will lead TLC to assign them an arbitrary valuation in the initial state while remaining constant throughout the trace (like the field `keys`). We refer to such modified module as `HotelVarScope`. As should be expected, such modification will have a great impact on the performance of TLC (Section 4).

3.2 Structural Properties

Although TLA$^+$ is untyped, it is considered good practice to always declare a type invariant over the module's parameters that is expected to hold in every state [5]. This is performed through the definition of a *state predicate*—first-order logic formulas without primed variables—as the *TypeInv* predicate in Fig. 3. We opted to encode the variables corresponding to Alloy fields as functions, because these are more manageable in TLA$^+$ and the Alloy specification does not rely on relational operators. Expression SUBSET A denotes the power-set of A, and is used to "type" many-valued functions. For simplicity, `lastKey` was implemented as a total function from rooms to keys instead of a partial function (in fact, from `init` and the defined actions, `lastKey` is always defined for every room). Besides the type declarations, the fact that the `keys` assignments are disjoint and that the `currentKey` is selected from these pools is also defined in *TypeInv*.

Since the valuation of constant parameters is defined externally, it makes little sense to define a state invariant over them. Instead, TLA$^+$ provides an ASSUME keyword that should be used for assumptions over constant parameters [5, p. 42]. Such instruction does not affect in any way the meaning of the specification, but can be used as hypothesis when trying to verify properties. Moreover, the first step of a TLC execution is testing whether the assignments of the constant parameters satisfy every ASSUME [5, p. 241]. In our example, in order to impose a total order over keys, they are assumed to be naturals (the instruction EXTENDS *Naturals* imports the required module). The main difference arises in the definition of `nextKey`, that relies on natural intervals to retrieve the next key.

Although it is not the case of hotel locking system, Alloy specifications that rely heavily on relational operators would not result in an embedding as clean as this one, since TLA$^+$ does not support relational operations like the converse nor transitive closure operations[4]. In such cases this would require the conversion of the transitive closure into a recursive definition, which is supported by TLA$^+$. Such is the case, for instance, in the ring election example from [4, p. 171] whose embedding in TLA$^+$ was also attempted.

Much like in Alloy, defining a predicate like *TypeInv* does not affect the specification of the system by itself. Nonetheless, the way that type invariants is typically handled by Alloy and TLA$^+$ renders evident a difference in the methodologies. In TLA$^+$, type invariants are typically not enforced in specification, but instead used to check the correctness of the defined actions [5, p. 26]. This reflects the prominent role of the actions in TLA$^+$ specifications, that, along with the initial state predicate, entail the set of acceptable states. In fact, TLC compatible actions must completely specify the succeeding state [5, p. 238]. In contrast, actions in Alloy are defined by regular declarative predicates and do not by themselves entail the set of valid states: structural properties are typically enforced by facts that restrict the states related by the actions. This allows the user to separate the concerns between the structural and behavioral components of the specification, and obtain simpler action definitions.

In this study, in order to be as faithful as possible to the Alloy specification, *TypeInv* was specified as a `CONSTRAINT` of the specification, as depicted in Fig. 4. This instructs TLC to ignore states where *TypeInv* does not hold [5, p. 241]. Nonetheless, we also experimented with *TypeInv* set as an invariant to be checked, as advocated in [5]. TLC is instructed to check a property P through `PROPERTY P` instructions, which in this case should take the shape `PROPERTY []TypeInv`. However, since invariants are so common, TLC provides the abbreviation `INVARIANT P` for such properties. Instruction `INVARIANT TypeInv` will thus instruct TLC to check whether *TypeInv* holds in every state, and throw an error in case it is broken. Interestingly, TLC proves that *TypeInv* does hold for every acceptable state under this specification. Thus, setting it as a constraint does not affect the behavior of TLC. Our experiments show that in this particular

[4] [As far as I am aware.]

example, having *TypeInv* set as a `CONSTRAINT`, an `INVARIANT` or not present at all, does not affect the performance of TLC significantly[5].

3.3 Behavioral Properties

Specifications in TLA$^+$ are expected to follow the shape $Init \wedge \Box[Next]_v \wedge Temporal$, where *Init* is a state predicate restricting the initial state, *Next* denotes valid *actions* and *Temporal* is a temporal formula specifying liveness conditions. An action is essentially a first-order logic formula with primed variables referring to their valuation in the succeeding state. Stuttering steps are intrinsic in TLA$^+$, and $[Next]_v$ denotes the fact that either *Next* is applied or a stuttering step is performed with $v' = v$. Since this example has no fairness properties defined, the overall specification is simply defined by the *Spec* predicate in Fig. 3.

Predicate *Init* follows the semantics of the corresponding Alloy predicate. Parameters *occupant* and *gKeys* are assumed to be initially empty, and *lastKey* to have the same valuation as *currentKey*. As for the *keys* parameter, it is free to take any valuation that results in a disjoint set and *currentKey* selected from those. Note that the two first conjuncts that bound these parameters to be functions are required by TLC to provide an upper bound to their valuation.

The *Next* predicate is comprised by the *Entry*, *Checkin* and *Checkout* actions, that are encoded as first-order logic formulas with primed variables. TLA$^+$ has also some syntactic sugar to denote data that remains unchanged: an UNCHANGED x expression is a simple abbreviation for $x' = x$ and for sequences or functions, the user may rely on EXCEPT expressions, where $f' = [f \text{ EXCEPT } ![x] = e]$ means f remains unchanged except for its valuation at x, which is updated to e (expression e may refer to the previous $f[x]$ value through the operator @). The specification from Fig. 3 relies on such operators, although it is not clear whether they can be derived by the original Alloy specification.

In order to be TLC compatible however, both the initial state predicate and the action predicates must follow some stricter rules that allow TLC to calculate their valuations. This means that the conjuncts must be ordered in such a way that guarantees that the occurring variables are already bound. For instance, in *Init*, the expression $\forall r \in Room : currentKey[r] \in keys[r]$ cannot occur before the expressions $keys \in [Room \rightarrow \text{SUBSET } Key]$ and $currentKey \in [Room \rightarrow Key]$, which impose the upper bounds for `keys` and `currentKey`. Moreover, in the action definitions TLC must be able to derive the valuation of every variable parameter in the succeeding state from the actions' definitions. In general, this imposes a set of instructions of the shape $x' = e$ for every variable x. This contrasts with the definition of actions in Alloy that are purely declarative predicates, which may not bound the next state completely: either their valuations are restricted by other facts of the specification, or their behavior is non-deterministic. Thus, it is not clear how actions TLC compatible actions can be derived from Alloy predicates.

[5] [This means that they may also be removed from the Alloy spec. What would be the impact?]

Name	Value
▼ <Initial predicate>	State (num = 1)
▶ occupant	(r1 :> { })
▶ lastKey	(r1 :> 1)
▶ currentKey	(r1 :> 1)
▶ keys	(r1 :> {1, 2})
▶ gKeys	(g1 :> { } @@ g2 :> { })
▼ <Action line 36, col 19 to lin...	State (num = 2)
▶ occupant	(r1 :> {g1})
▶ lastKey	(r1 :> 2)
▶ currentKey	(r1 :> 1)
▶ keys	(r1 :> {1, 2})
▶ gKeys	(g1 :> {2} @@ g2 :> { })
▼ <Action line 32, col 16 to lin...	State (num = 3)
▶ occupant	(r1 :> { })
▶ lastKey	(r1 :> 2)
▶ currentKey	(r1 :> 1)
▶ keys	(r1 :> {1, 2})
▶ gKeys	(g1 :> {2} @@ g2 :> { })
▼ <Action line 27, col 17 to lin...	State (num = 4)
▶ occupant	(r1 :> { })
▶ lastKey	(r1 :> 2)
▶ currentKey	(r1 :> 2)
▶ keys	(r1 :> {1, 2})
▶ gKeys	(g1 :> {2} @@ g2 :> { })

Fig. 5: TLC counter-example for *NoBadEntry*.

3.4 Assertions

In order to check the validity of `NoBadEntry`, the predicate must be converted into valid a TLA$^+$ formula that TLC is able to check. Unfortunately, the class of formulas that TLC is able to check is even more limited than the those used to specify the system [5, p. 236]: they must either be state predicates P, invariance predicates $\Box P$, box-action formulas $\Box[A]_v$ or simple temporal formulas, i.e., boolean combinations of temporal state formulas (the combination of state predicates with $\Box$, $\Diamond$ and $\rightsquigarrow$) and simple action formulas (expressions $\mathrm{WF}(A)$, $\mathrm{SF}(A)$, $\Box\Diamond\langle A\rangle_v$ and $\Diamond\Box[A]_v$). This contrasts with Alloy where any predicate supported by the language can be checked by the Analyzer.

Luckily, the `NoBadEntry` can be translated into a TLC compatible TLA$^+$ formula in a straightforward manner in the shape of a box-action formula $\Box[A]_v$:

$$\Box[\forall\, g \in \mathit{Guest},\, r \in \mathit{Room},\, k \in \mathit{Key} : \mathit{Entry}(g,\, r,\, k) \wedge \mathit{occupant}[r] \neq \{\} \Rightarrow g \in \mathit{occupant}[r]]_{\mathit{vars}}$$

Meaning that at every state, and *Entry* action in non-empty rooms should only be performed by its occupants. As already mentioned, TLC is instructed to check properties through `PROPERTY` instructions, as in the configuration file from Fig. 4. As expected, TLC finds a counter-example for this property, which is depicted in Fig. 5.

The fact that *NoBadEntry* must defined as a box-action does raise some issues regarding the enforcement of stuttering steps. Concretely, when a guest enters the room for the second time—i.e., when his key has already been registered in the room—the *Entry* action does not update state of any variable. Thus, TLC

will never recognize this step, identifying instead a stuttering step. In fact, the following property, stating that there are no entries in a room with an updated key, holds in TLC:

$$\Box[\forall g \in Guest, r \in Room, k \in Key : \\ \neg(Entry(g, r, k) \wedge currentKey[r] = lastKey[r])]_{vars}$$

3.5 Behavior Constraints

To guarantee that *NoBadEntry* holds, a `NoIntervening` predicate was specified that forces guests that check in to enter the room in the following step. Unfortunately, `NoIntervening` is not a valid formula in the TLA$^+$ language (it is *not* a limitation of TLC) because variables cannot be doubly primed. This renders the expression `entry[t',t",g,r,k]` not expressible in TLA$^+$. In this case, *NoIntervening* was adapted to force an *Entry* action to occur whenever there is a guest whose key is not registered in the room, which is the post-condition of the *Checkin* action[6]:

$$[\forall\, g \in Guest,\, k \in Key,\, r \in Room : \\ (g \in occupant[r] \wedge k \in gKeys[g] \wedge lastKey[r] = k \wedge currentKey[r] \neq k) \Rightarrow \\ Entry(g,\, r,\, k)]_{vars}$$

Similar to `CONSTRAINT` instructions for state predicates, TLC can also process `ACTION-CONSTRAINT` instructions for action formulas, that removes from the search space steps where the defined formula does not hold. This was followed in Fig. 4 for *NoIntervening*. Under this configuration, TLC does not find any counter-example for *NoBadEntry*, as expected.

One could also wonder whether it would not suffice to check property *NoIntervening* $\Rightarrow$ *NoBadEntry*. Unfortunately, this is not a valid TLC property, since it is not a simple temporal formula (it is not a boolean combination of temporal state formulas and simple action formulas).

4 Evaluation

The underlying mechanism through which the Analyzer and TLC check properties is fundamentally different. The Analyzer embeds the specification into a boolean predicate which are subsequently fed to a SAT solver. As a consequence, the underlying process is unaware of the different components of the Alloy specification that gave rise to the SAT problem and obliviously search for a valid solution. This also renders the procedure bounded, thus the properties are only ever checked for traces with limited length. In contrast, TLC analyzes the TLA$^+$ specification using an explicit-state model checker and derives from it the procedure through which the state space is searched. This allows TLC to have a finer control on how

[6] [It is possible that there are better ways to define this...]

the states are explored, allowing breadth-first search procedures that generate counter-examples with minimal trace length. However, this happens to be a hinderance in [*Structural/Dynamic*] systems: since they possess rich structural properties, not enforcing the initial state to a concrete valuation, the generation of every initial state *a priori* may unnecessarily encumber the model checker. There are some comparison points regarding the performance of these two tools that may be studied: this section provides a preliminary evaluation. All tests were performed multiple times on an 1,8 GHz Intel Core i5 with 4 GB memory running OS X 10.10.

All the comparisons made in this section are divided into two main orthogonal axes: first, whether the `NoIntervening` constraint is enforced or not; second, whether the scope of the parameters is fixed or variable (i.e., module `HotelExactScope` vs. `HotelVarScope`). Module `HotelVarScope` was tested for different scopes n such that $\#Key = \#Guest = \#Room = n$, while module `HotelExactScope` was tested for different scopes n for $(\#Key) + 2 = \#Guest = \#Room = n$ (since detecting the inconsistency requires two extra keys). Note that as a consequence, executions of `HotelVarScope` and `HotelExactScope` for the same n are not comparable.

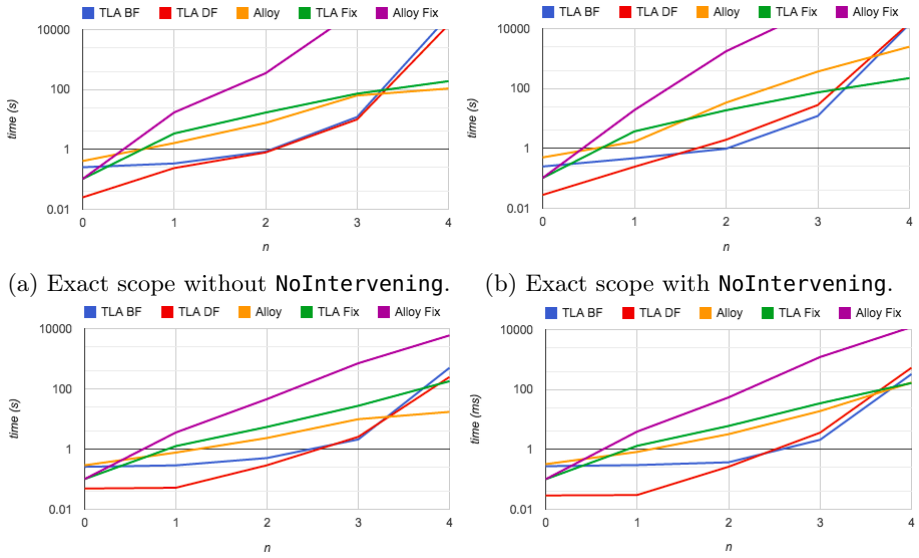
Although TLC's default model is breadth-first (BF), it also supports depth-first (DF) searches, which were also tested (for a maximum depth of 100). TLC was also instructed not to flag deadlocks as errors, as this system is expected to halt once all keys have been used. The Alloy Analyzer was run over the MiniSat solver. Its bounded nature requires that a maximum trace length t be set. Moreover, it must be run up to that maximum so that every trace length is tested: the timing for testing a property for trace length t aggregates the timing of the previous $t - 1$ runs. In Alloy the exact scope version is attained by simply setting the scope of check command to be exact, e.g., for $n = 3$:

```
check NoBadEntry
  for 3 but exactly 3 Guest, exactly 3 Room, exactly 6 Key, 30 Time
```

Figure 6 compares the performance of the approaches for a fixed trace length $t = 30$ and increasing scope n (the t value is only relevant for the Alloy-based bounded techniques, as in TLC the checking is unbounded). In general, although TLC fares better than the Analyzer for smaller scopes, the Analyzer largely outperforms TLC for larger scopes. This is due to the fact that TLC always generates every possible initial state as a first step, even if performing in DF (at $n = 4$ for `HotelVarScope`, there are already 18,960 initial states). Interestingly, TLC is actually faster with `NoIntervening` enabled than without it. Our experiments show that it finds rather quickly that there is a counter-example for the specification, but then spends considerable time trying to effectively generate it[7].

Figure 7 compares instead the performance of the approaches for a fixed scope $n = 4$ with increasing trace length t. Length t is only relevant for the

[7] [I browsed TLC's source code and found the bottleneck. Apparently, when generating the counter-example, it retrieves every initial state again in order to find the one that originated the counter-example.]



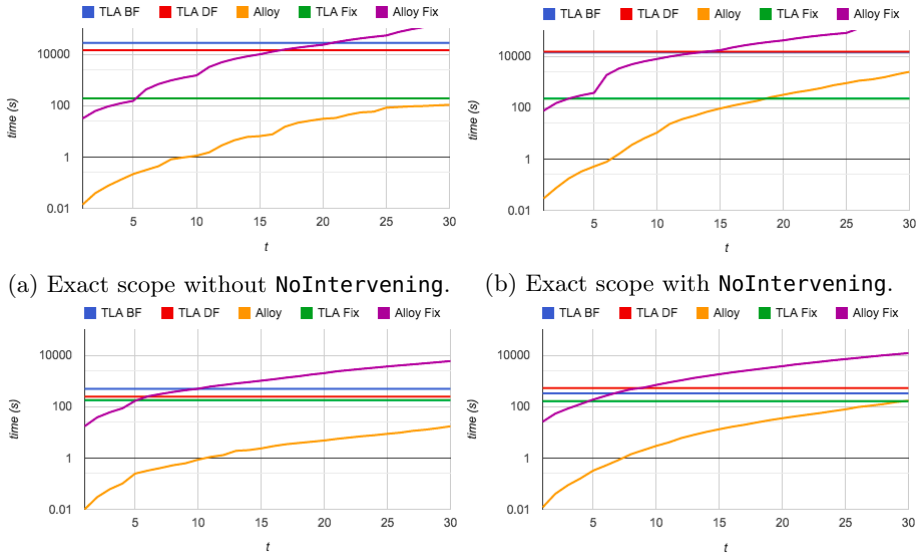
(a) Exact scope without `NoIntervening`. (b) Exact scope with `NoIntervening`.

(c) Variable scope without `NoIntervening`. (d) Variable scope with `NoIntervening`.

Fig. 6: Performance of different approaches for increasing n under $t = 30$ (for bounded techniques).

Alloy bounded procedures: since TLC is unbounded, verifying the property for every trace, it appears as a constant function on t. As was just seen, the Analyzer outperforms for such n even at $t = 30$. However, when `NoIntervening` is disabled, it would actually detect the counter-example at $t = 5$ in less than a second for both `HotelVarScope` and `HotelExactScope`, in contrast to TLC's 498 and 27970 seconds, respectively.

Our experiments show that TLC spends a great deal of time generating every initial state as a first step. For $n = 4$, there are 9000 and 18,960 distinct initial states for the `HotelExactScope` and `HotelVarScope` specifications respectively, arising from the attribution of keys to the rooms and the selection of the current key from those pools. Table 1 provides an overview of the generated initial states for a scope of 1 room and 3 keys. Generating every initial state *a priori* is essential to obtain a breadth-first search procedure, but for reasons unknown to us, the depth-first algorithm also performs this generation. Since in Alloy the Analyzer is expected to be deployed with increasingly larger trace lengths starting at $t = 1$, its search procedure is also naturally breadth-first. In some contexts, TLC's breadth-first algorithm could pay off due to its ability to detect equivalent states when exploring the state search space. However, in this example (and in fact, in [*Structural/Dynamic*] problems in general, with variable fields that remain fixed for the entire trace), state sharing is limited: the `keys` variable is fixed for the entire execution; sharing only occurs as the `currentKey` evolves. In Table 1, the initial states reachable through other states are colored gray.



(a) Exact scope without `NoIntervening`. (b) Exact scope with `NoIntervening`.

(c) Variable scope without `NoIntervening`. (d) Variable scope with `NoIntervening`.

Fig. 7: Performance of different approaches for increasing t (for bounded techniques) under $n = 4$.

The overhead imposed on TLC is in part due to the rich first-order properties that are imposed over the initial state in [*Structural/Dynamic*] problems, which TLC is not able to process efficiently. Thus, we explored a different approach, where the Alloy Analyzer is used to generate every initial state, and then TLC is deployed over a fixed initial state. The advantage of this two-phased technique is that it allows the partitioning of the problem into as many problems as there are initial states. This helps avoiding executions running out of memory for large scopes that tackle every possible state at once. The tradeoff is that it will no longer guarantee that the detected counter-example is one with minimal trace length since it checks initial states in an arbitrary order. Also, TLC will no longer be able to benefit from state sharing.

The Alloy specification that generates such states can be easily derived from the original one by, first, removing every predicate that considers two instants of time, and second, remove every reference to the `Time` signature (adapting the multiplicities of the fields accordingly). The result is depicted in Fig. 8 for the hotel locking system example. A factor that must be taken into consideration, is that while TLC detects, e.g., 18960 initial states for $n = 4$ in `HotelVarScope`, Alloy with symmetry breaking enabled only generates 520 unique states, which it does in less than 2 seconds (with symmetry breaking disabled, the Analyzer does produce the 18960 states). While TLC also enables the definition of the model elements as symmetric, in this particular example this does not seem to affect the generation of the initial states in any way.

```
open util/ordering[Time] as to

sig Key {}

sig Room {
  keys: set Key,
  currentKey: one keys }

fact DisjointKeySets {
  keys in Room lone -> Key }

one sig FD {
  lastKey: Room -> lone Key,
  occupant: Room -> Guest }

sig Guest {
  gKeys: set Key }

pred init {
  no Guest.gKeys
  no FD.occupant
  all r: Room | FD.lastKey [r] = r.currentKey }

fact traces {
  init }

run { } for 3 but exactly 3 Room, exactly 3 Guest, exactly 6 Key
```

Fig. 8: Alloy specification for the generation of initial states.

Given the Alloy specification for the initial states, we timed both the generation of these states by Alloy[8] and the execution of both TLC and the Analyzer[9] for a fixed initial state. The results regarding this two-phased technique are presented under "TLC Fix" in and "Alloy Fix" in Figs. 6 and 7. The performance of this technique when there are counter-examples to be found (i.e., without `NoIntervening` in our example) is dependent on from how many initial states can the counter-example be detected. Just to give an idea, in Table 1, initial states that lead to a `NoBadEntry` counter-example are colored green, which for this scope is a single state[10]. Thus, for the experiments with `NoIntervening` disabled, the timing of the fixed execution was only multiplied by a portion of the initial states; for the experiments with `NoIntervening` enabled, they were multiplied by every initial state.

TLC with fixed initial states seems to outperform regular TLC executions, in some cases quite significantly, although its gains over pure Alloy are not so evident. Alloy with fixed initial states fares worse than all the other techniques.

[8] The execution times were actually calculated over Kodkod directly.

[9] [Ideally these should be run directly in Kodkod so that exact bounds can be imposed on the relations. This should improve the performance of "Alloy Fix" considerably.]

[10] [With Kodkod's symmetry breaking, I cannot infer the ratio]

keys[R1]	currentKey[R1]	keys[R1]	currentKey[R1]
{K1,K2,K3}	K1	{K2,K3}	K3
{K1,K2,K3}	K2	{K1,K2}	K1
{K1,K2,K3}	K3	{K1,K2}	K2
{K1,K3}	K1	{K1}	K1
{K1,K3}	K3	{K2}	K2
{K2,K3}	K2	{K3}	K3

Table 1: Initial states for `Key = {K1,K2,K3}` and `Room = {R1}`. Green states lead to inconsistencies; gray states are reachable through others.

5 Discussion and Future Work

Throughout this paper some pros and cons of using Alloy and TLA$^+$ to specify [*Structural/Dynamic*] problems were identified. Alloy has two main limitations: first, dynamic behavior must be explicitly modeled by the user, which is a cumbersome and error-prone task, even if following well-known idioms; second, the Analyzer only supports bounded model checking, which hinders sound verification of liveness properties. The fact that each execution of the Analyzer only verifies the properties for a specific trace length that is specified by the user renders this process even more cumbersome.

The major advantage of Alloy lies in its expressive language that is fully supported by the Analyzer, in contrast with TLC that imposes restrictions over TLA$^+$ specifications. This is patent in the definition of the initial state predicate and the action predicates, which in Alloy are purely declarative specifications but that must completely specify the state in TLA$^+$ if they are to be processed by TLC. Besides the specification of the system, the properties that TLC is able to verify are also restricted. Since dynamism in Alloy is not native, it actually renders it more expressive than native temporal model checkers, patent in the fact that doubly primed variables are not allowed in TLA$^+$. Finally, the management of non-variable arbitrary artifacts, including signature population, common in [*Structural/Dynamic*] problems are better manageable in Alloy than in TLA$^+$, where constants are assigned a valuation externally to the checking procedure.

Regarding the possible embedding of Alloy specifications into TLA$^+$, some mismatches have also been identified that would hinder this process. First, the translation of specifications rich in relational operators would not be straightforward. This is especially problematic with transitive closure operations, that must be converted to recursive definitions in TLA$^+$. Second, it is not clear if deriving TLC compatible specifications (initial state predicate and actions) and properties to be verified from arbitrary Alloy predicates would be possible. It is also not clear how enforcing stuttering steps in TLA$^+$ would affect the semantics of actions derived from Alloy where stuttering steps are not enforced.

As for performance, the Analyzer seems to outperform TLC with larger scopes, but is of course performing in a bounded search space. The alternative technique explored where the initial state of TLA$^+$ specifications is fixed shows promise,

but since the initial states are being generated by Alloy, the mismatches between the languages would have to be tackled in order to implement it effectively.

[TODO: compare with previous work on dynamic Alloy [3,1,6,7,2].]

References

1. F. S. Chang and D. Jackson. Symbolic model checking of declarative relational models. In *ICSE 2006*, pages 312–320, 2006.
2. A. Cunha. Bounded model checking of temporal formulas with alloy. In *ABZ 2014*, pages 303–308, 2014.
3. M. F. Frias, J. P. Galeotti, C. L. Pombo, and N. Aguirre. DynAlloy: upgrading alloy with actions. In *ICSE 2005*, pages 442–451, 2005.
4. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.
5. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
6. J. P. Near and D. Jackson. An imperative extension to Alloy. In *ABZ 2010*, pages 118–131, 2010.
7. A. Vakili and N. A. Day. Temporal logic model checking in alloy. In *ABZ 2012*, pages 150–163, 2012.