Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations*

Nuno Macedo
HASLab, INESC TEC and Universidade do Minho
nfmmacedo@di.uminho.pt

Julien Brunel
DTIM, UFTMiP, ONERA
julien.brunel@onera.fr

David Chemouil
DTIM, UFTMiP, ONERA
david.chemouil@onera.fr

Alcino Cunha
HASLab, INESC TEC and Universidade do Minho
alcino@di.uminho.pt

Denis Kuperberg
TU Munich
denis.kuperberg@gmail.com

ABSTRACT

Model-checking is increasingly popular in the early phases of the software development process. To establish the correctness of a software design one must usually verify both *structural* and *behavioral* (or temporal) properties. Unfortunately, most specification languages, and accompanying model-checkers, excel only in analyzing either one or the other kind. This limits their ability to verify dynamic systems with rich *configurations*: systems whose state space is characterized by rich structural properties, but whose evolution is also expected to satisfy certain temporal properties.

To address this problem, we first propose Electrum, an extension of the Alloy specification language with temporal logic operators, where both rich configurations and expressive temporal properties can easily be defined. Two alternative model-checking techniques are then proposed, one bounded and the other unbounded, to verify systems expressed in this language, namely to verify that every desirable temporal property holds for every possible configuration.

CCS Concepts

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

Keywords

Model-checking, formal specification language

*Work funded by the European Regional Development Fund (ERDF) through the Operational Programme for Competitiveness and Internationalisation (COMPETE 2020) and by National Funds through the Portuguese funding agency, Fundação para a Ciência e a Tecnologia (FCT) within project POCI-01-0145-FEDER-016826. Research partly funded by DGA/ANR project Cx (ref. ANR-13-ASTR-0006) and *fondation STAE* (IFSE, BRIefcaSE).

1. INTRODUCTION

Software specification and verification is crucial at early development phases, since it allows the developer to reason about the system and its properties, and timely detect design errors. Although a variety of frameworks has been proposed to aid the developer in this task, the most successful ones are *lightweight*, in the sense that 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. In fact, such frameworks have already reached a level of maturity that enables their application in complex real world scenarios [20].

Two classes of properties are particularly important to consider: *structural* (or *static*) properties, typically expressed in some variant of first-order logic, that address the well-formedness of the system state, and *behavioral* (or *dynamic*) properties, typically expressed in a temporal logic, that address the evolution of the system state. Although not necessarily in equal measure, most interesting systems will require the specification and analysis of properties from both classes. The analysis of distributed computing algorithms is a paradigmatic class, whose behavioral properties are expected to be checked for arbitrary network topologies, within a range specified by particular structural properties. We denote such components of the system state, that are initially arbitrary, but remain unchanged as the system evolves, as *configurations*. Another relevant class is that of software product lines (SPLs), where each valid software product of a family, specified by simple structural properties, amounts to a different configuration, each of which should be checked for the behavioral properties.

Dynamic systems with rich configurations are the focus of this work, and concretely, this class of systems exhibits the following requirements:

R1 A clear distinction between the specification of the system configuration and the system evolution;

R2 Configurations constrained by rich structural properties (like inheritance, complex relationships between entities, or reachability properties);

R3 A declarative specification of the system evolution (the possible actions affecting the state), possibly under

different idioms;

R4 The need to verify (temporal) safety and liveness properties about the specified system.

Thus, to be suitable to address this kind of problems, a specification language should be sufficiently rich and flexible to support the definition of both structural and behavioral properties, while still promoting the separation of concerns. Moreover, it should be accompanied by effective tool support, to allow the automatic model-checking of the desired temporal properties for every valid configuration.

1.1 Motivating Examples

To further clarify the class of problems we intend to address, this section presents two motivating examples where all the above characteristics are manifest.

Hotel room locking system.

This example regards the specification of a hotel room locking system that uses disposable electronic keys [14], that relies on recodable locks that either unlock the door for the currently coded key, or for its successor, at which point the lock is recoded, rendering the previous key obsolete. The front desk issues new keys for the appropriate room when guests check-in. The front desk and the locking systems are stand-alone (no communication between them): the system works properly because keys are generated using the same pseudo-random generator, with the initial seed of each room lock being synchronized with the front desk *a priori*.

To abstract away the details, keys are interpreted as a totally ordered set (*e.g.*, a set of natural numbers), and each room is assigned a (disjoint) subset of such keys *a priori*: given the currently coded key, the next valid one is the smallest among its successors in this subset. The available rooms, keys, and possible guests, together with a valid assignment of keys to the rooms, constitutes a configuration of this system. These remain constant as the system evolves, contrasting with the dynamic components of the system (*e.g.*, the keys currently coded in each lock) (**R1**). Moreover, a valid configuration is not arbitrary, but characterized by a precise set of constraints (**R2**). The dynamic components evolve as guests check in and out, or enter a room with a fresh key, updating the currently coded key of that room lock. These actions can easily be specified in a declarative manner, for example relying on pre- and post-conditions (**R3**).

A safety property that is expected to hold (**R4**) is that guests cannot enter rooms in which they are not currently registered. This property does not hold in some configurations, as depicted in Fig. 1. Each state depicts the rooms (square elements), guests (rounded corners) and the front desk (the lower faceted rectangle). Bold typeface values are part of the system configuration; the others change as the system evolves, with the values that are modified at each step underlined. Although this particular configuration leads to a counter-example, the problem may go unnoticed if one is required to perform the analysis for a specific (user-picked) configuration (*e.g.*, where $R1$ was assigned $K1$ and $K2$ and $R2$ the remaining keys).

Distributed spanning tree algorithm.

This example concerns a simple distributed spanning tree algorithm, that runs on an arbitrary (but connected) network topology, building on the one proposed in [22]. Here, a distinguished root node (possibly elected beforehand) starts by assigning itself level 0. Nodes with assigned levels (*i.e.*, already in the spanning tree) broadcast their level to the neighbors. When a node not yet in the spanning tree receives one such message, it sets its level to one plus the level of the sender, and records it as its parent node. The details of message passing are abstracted away by allowing the system to evolve by selecting an arbitrary node to act, among those not in the spanning tree but with neighbors already so, and arbitrarily choosing one of the latter as parent.

Here, a configuration consists of a set of nodes, a root node among them, and a possible network topology, which may take the shape of an arbitrary undirected connected graph, while the dynamic aspects encompass the level and parent structures of the tree being computed (**R1**). The specification of a valid topology requires a reachability constraint (**R2**). In general, the same topology may lead to several different spanning trees, and this non-determinism can be captured by declarative operations, where the selection of the process to act, as well as its parent, is arbitrary within the stated constraints (**R3**). Both a safety and a liveness properties are expected to hold for every network topology, *i.e.*, system configuration (**R4**): the algorithm never introduces a cycle in the parent structure, and a spanning tree of the whole network is eventually computed, respectively.

Figure 2 depicts a possible execution trace of the algorithm for a specific configuration with four nodes ($P1$ to $P4$), root node $P2$ (bold circle), and network topology depicted with dashed lines. The changing elements of the specification, namely the level and parent of each node, are represented in the lower-half of the respective circle and by solid arrows leaving from them, respectively. Again, the values updated in each step are underlined.

Elevator system SPL.

This example models an elevator system SPL inspired by the one proposed in [21] and extended in [7]. The model consists of an elevator and a set of floors; at each floor there is a button that calls the elevator, and inside it there is a button for each of the floors. The base system answers button calls giving priority to the current direction: it only changes direction when there are no more calls for the succeeding floors. This behavior is however modified as additional features are selected. For instance, a parking feature moves the elevator to the first floor when there are no button calls; an idle feature forces the elevator to open the door when there are no button calls; an executive floor feature prioritizes calls to one of the floors over the others. Multiple interfering features, under some restrictions, may be selected.

Here each configuration represents a valid product from the SPL, that is, a selection of features, while the dynamic component models the evolution of the system taking into consideration those features (**R1**). The selection of these features is restricted by a feature diagram that defines simple dependencies/conflicts between the features, which can be encoded as structural properties (**R2**). For instance, since the idle and the parking features have conflicting behavior their choice could be forced to be exclusive. The operations must be sufficiently expressive to encode the behavior of the system taking into consideration the selected features (**R3**).

There are several safety and liveness properties that should be checked about this specification (**R4**). For instance, the most basic liveness property states that a pressed button

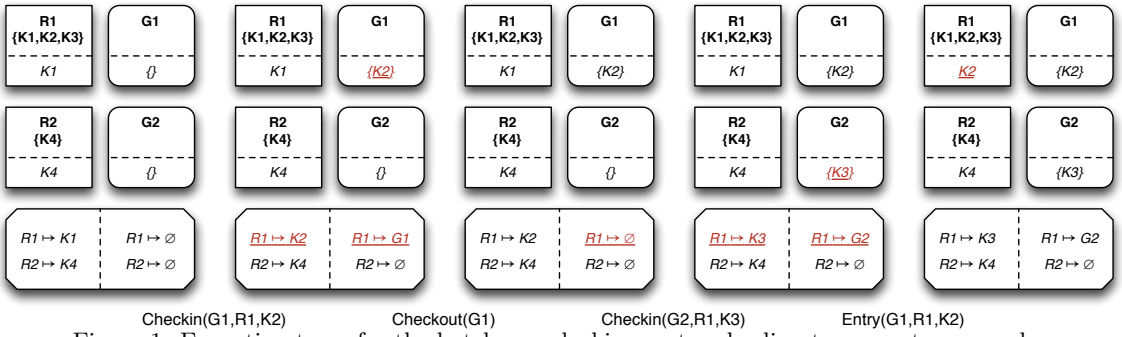


Figure 1: Execution trace for the hotel room locking system leading to a counter-example.

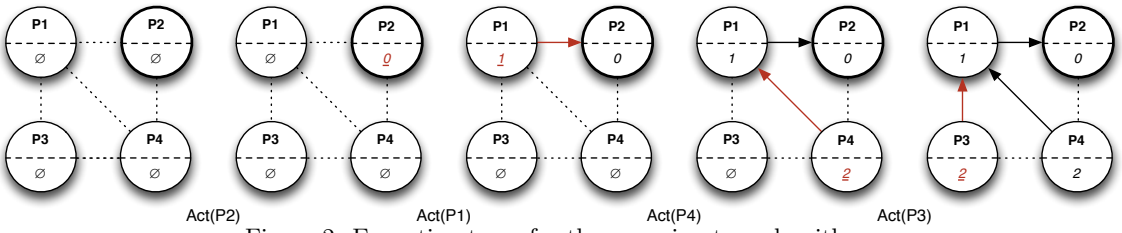


Figure 2: Execution trace for the spanning tree algorithm.

will eventually be answered. These must be checked over every possible feature combination. While some of these are expected to always hold, some fail under certain feature configurations. For instance, the above property will fail with the executive floor feature, as calls to those floors will be prioritized.

1.2 Contributions

Unfortunately, most formal specification languages (and accompanying model-checkers) are not designed nor optimized to analyze problems such as these. For example, most standard model-checkers only perform well with fixed configurations, while languages more geared towards the analysis of structural properties, usually without native support for some sort of temporal logic, require the user to verify behavioral properties through *ad hoc* mechanisms.

This paper aims precisely to fill this niche, and proposes a language and model-checker tailored for the lightweight analysis of dynamic systems with rich configurations. Concretely, we propose:

- A formal specification language, inspired by Alloy [14] and TLA$^+$ [15] (two of the most popular specification languages nowadays), that simplifies the specification of systems exhibiting all the requirements defined above;
- Two model-checking techniques, one bounded and the other unbounded, to verify systems expressed in such language, namely to verify that every desirable temporal property holds for every possible configuration.

The remainder of the paper is organized as follows. Section 2 explores related languages and techniques, and justifies why they fall short when analyzing this class of problems. Section 3 presents the proposed language, its semantics, and the proposed model-checking techniques. Their performance is then evaluated in Section 4. Finally, Section 5 draws conclusions and points directions for future work.

2. RELATED WORK

There are numerous approaches to the specification and model-checking of systems. Here we focus on those whose level of expressiveness and tool support come close to that needed to handle systems with rich configurations, *i.e.*, that address some of the four requirements defined in Section 1.

Alloy [14, **?**, **?**, 5] is a lightweight formal specification language with an object-oriented flavor, which, paired with its Analyzer, that provides support for automatic bounded verification, has been increasingly adopted by software engineering practitioners. The underlying formalism of Alloy is *relational logic*, first-order logic enhanced with transitive closure operations, that render the definition of structural properties extremely simple. Thus, Alloy is naturally well-suited to handle the **R2** requirement.

However, Alloy is inherently static, thus the verification of behavioral properties usually relies on well-known idioms that have emerged due to the language flexibility. Such *ad hoc* specification is error-prone and verbose, and forces developers to be concerned with particularities of the idiom rather than with the properties that they actually wish to verify (see for instance [**?**, **?**]), and as a consequence regular Alloy is not well-suited to address **R4**. To overcome this limitation, considerable research has been dedicated to enhance Alloy with dynamic behavior [11, 5, 18, 24, 8]. 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 to adhere to specific idioms, and consequently breaking the **R3** requirement. That is the case, for instance, of DynAlloy [11], an Alloy variant that resorts to dynamic logic to specify behavior. Liveness properties, which comprise a large class of behavioral properties, are also not expressible in DynAlloy [12], and thus **R4** is not effectively addressed. Although expressible in regular Alloy (via said idioms), verifying such properties with the Analyzer requires some insightfulness and care from the user, to avoid the spurious counter-examples that usually occur with naive encodings of bounded model-checking techniques [24,

8]. The technique from [18] enhances Alloy with imperative constructs, again undermining **R3**. In contrast, the technique proposed in [5] extends the relational logic of Alloy with CTL temporal logic. Unfortunately, the system actions must be specified with a fixed idiom with an imperative-flavor, and thus falls short on the **R3** requirement. Moreover, it disregards the rich structural properties introduced by the signature type system from regular Alloy, undermining its ability to address **R2**. Finally, even though all these works attempt to enhance Alloy with dynamic properties, besides [18] that distinguishes between static and variable fields, none properly address **R1**, since they do not distinguish between the system configuration and the dynamic components that evolve over time.

In contrast, temporal model-checkers are developed precisely to address the **R4** requirement. Among the most successful formalisms is the *temporal logic of actions* (TLA) [15], 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, specifying the acceptable steps that allow the system to evolve. Thus, the TLA$^+$ specification language, built over this logic, naturally handles **R3**, and has proven to be well suited to specify systems with rich temporal properties. 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. However, in order to be manageable by TLC, some additional restrictions are imposed over the TLA$^+$ language, namely over the action predicates, reducing its compliance with the **R3** requirement.

Unlike most model-checkers, TLA$^+$ does support the specification of first-order logic properties, addressing **R2** to some extent. However, unlike Alloy [**?**], it lacks a type system with inheritance, that greatly simplifies the specification of complex entities and their relationships. Moreover, due to the nature of the model-checking procedure, rich structural properties may severely hinder the performance of TLC since the first step of the procedure involves calculating every possible initial state. While TLA$^+$ does provide a simple mechanism to separate the system configuration from its evolution (by allowing the distinction between variable and constant parameters), TLC requires constant parameters to be fixed *a priori* by the user, limiting its ability to automatically explore all possible configurations and thus to fully address **R1**. To circumvent this problem, configurations must be encoded in regular variable parameters and then (artificially) constrained in the specification to remain constant throughout the system evolution.

Thus, although both Alloy and TLA$^+$ exhibit powerful but simple languages associated with effective and automated tools, they excel in the verification of different classes of systems, and none addresses all four requirements identified above[1]. The number of available model-checking languages and tools is too large to allow for an exhaustive comparison here, but in general they are not better than Alloy or TLA$^+$ at dealing with dynamic systems with rich configurations. Most, like SPIN or the various SMV variants, do not even support first-order logic, making it very cumbersome to specify complex structural properties. This related work focuses on Alloy and TLA$^+$ because we believe they are quite close to excel at handling such systems, and, in fact, the language here proposed combines their best aspects.

An alternative perspective to the problem of model-checking multiple configurations arises from the area of SPLs, a set of software products that share common base functionalities. The variability between the products is defined through the selection of features; acceptable feature combinations (i.e., products) are defined through feature diagrams that impose simple dependencies and conflicts between them. Model-checking an SPL involves checking the properties for every acceptable product. Under our perspective, each of these products represents a configuration of the system, restricted by the (rather basic) structural properties entailed by the feature diagram, that are to be checked for the temporal properties. Several techniques are able to model-check SPLs. Most, however, are not accompanied by high-level specification languages that allow both the modeling of the SPL and the feature diagram. One such language, with support for model-checking, is *fSMV* [21, 7], an extension to SMV. It follows a compositional approach, in the sense that each feature is implemented through modifications to the base system. *FeatureAlloy*, an extension to Alloy following a similar approach, has also been proposed [1]. In contrast, in annotative approaches the SPL is represented by a single system whose behavior is defined by guards over the selected features. One such language is *fPromela* [6], an extension to the Promela language of SPIN. These approaches suffer from the expressiveness problems of the languages they extend that have already been exposed previously in this section.

3. THE ELECTRUM FRAMEWORK

Considering the presented state-of-the-art, this work proposes an extension of the Alloy specification language with temporal logic operators – denoted *Electrum* – and associated model-checking tools, that address all four characteristics identified in Section 1.

3.1 Language

This section describes the proposed Electrum language and its formal semantics in several steps, starting with an informal overview in Section 3.1.1. To ease the presentation of the semantics, Section 3.1.2 introduces an abstract syntax for a representative subset of Electrum, whose semantics is expressed in terms of a translation to a first-order temporal logic. For the sake of readability, this logic is described in Section 3.1.3, before the translation itself in Section 3.1.4.

3.1.1 Overview

This section introduces Electrum, whose concrete syntax is presented in Fig. 3. The language is inspired both by Alloy, for its structural concepts, and by TLA$^+$, for its ability to freely define actions as predicates with primed variables. The specification of the examples in Electrum, as well as their Alloy and TLA$^+$ versions, can be found here [**?**].

Likewise Alloy, structure in Electrum is introduced through the declaration of *signatures* which specify sets of uninterpreted atoms. Hierarchical signatures can be introduced by *extension*, in which case the sub-signatures must be disjoint, or through *inclusion*, in which case sub-signatures may overlap with each other. *Abstract* signatures are comprised only of the atoms of their sub-signatures. Finally, signatures may be attached with *multiplicities* that restrict the num-

[1]An in-depth comparison of TLA$^+$ and Alloy, using the hotel room locking system as running example, can be consulted in [16]. The specification of the running examples of this paper in both those languages can also be found at [**?**].

ber of atoms that they may contain. Signature declarations may also introduce *fields* with arbitrary, finite arity, that represent relations between the various signatures. These constructs are essentially those provided by the standard Alloy language.

Contrary to Alloy, however, both signatures and fields may be additionally tagged as *variable*, meaning that their valuation may evolve in time. In contrast, non-variable signatures and fields are assumed to be static, meaning that their valuation remains fixed throughout the evolution of the system. Thus, Electrum provides a clear distinction between the configuration of the system (static constructs) and its evolution (variable constructs) (**R1**).

Additional restrictions are introduced through the definition of *paragraphs*: *facts* (axioms) impose restrictions on the specifications and *assertions* denote properties that are to be checked over the specification. *Predicates* and *functions* are essentially reusable formulas and expressions, respectively. Signatures may also be annotated with local facts, that apply to every atom of the signature in every instant of time.

All these paragraphs are comprised of logical formulas that borrow their expressiveness from Alloy (supporting universal and existential quantifications, as well as transitive closure operations) and from TLA$^+$ (supporting classical temporal operators[2] as well as primed expressions), and thus allow both the specification of rich structural properties (**R2**), the definition of actions in a flexible manner (**R3**).

Verification commands (to be model-checked) are integrated in the specification file (again, inspired by Alloy), allowing the verification of rich temporal properties (**R4**). *Check* commands are passed an assertion and scopes for the (static and variable) signatures and instruct the model-checker to try to prove the assertion, while *run* commands instruct the model-checker to yield an example instance of the specification if there is one (this way, it also shows whether the specification is indeed consistent). The scopes bound the maximum number of elements that a top-level signature will at least contain and are described in more detail in Section 3.2.1. Remark that the model-checking techniques are expected to explore every valid valuation of signatures and fields *up to the given bound*[3]. It should be noticed that, for a variable signature, the scope bounds the total number of its instances over the complete life of the modeled system[4].

Figure 4 presents the hotel room locking system specified in Electrum. Rooms, guests and keys are introduced by static signatures, meaning that their valuation remains fixed along the system evolution. There is also a singleton signature `FD` representing the front desk. There is only one static field `keys`, that represents the set of keys assigned to each room; this distribution is constrained to be a partition by fact `DisjointKeys`. Every other field, denoting the key currently coded for a room, and the registry of occupants and assigned keys at the front desk, is variable. Actions are declared as regular predicates that refer to the post-state by priming

[2]The keyword **`after`** was chosen to stand for the **X** (read: "next") temporal operator because, in Alloy, `next` is a predicate from a commonly used standard library for ordering, and we wanted to preserve upward compatibility with Alloy.
[3]This contrasts with TLC configuration files which assign concrete valuations to the constant parameters.
[4]As time may be unbounded, this is a way to retain decidability while remaining faithful to the successful bounded verification practice of Alloy.

```
spec ::= module qualName [ [ name,+ ] ] import* paragraph*
import ::= open qualName [ [ qualName,+ ] ] [ as name ]
paragraph ::= sigDecl | factDecl | funDecl | predDecl
    | assertDecl | checkCmd
sigDecl ::= [ var ] [ abstract ] [ mult ] sig name,+
                [ sigExt ] { varDecl,* } [ block ]
sigExt ::= extends qualName | in qualName [ + qualName ]*
mult ::= lone | some | one
decl ::= [ disj ] name,+ : [ disj ] expr
varDecl ::= [ var ] decl
factDecl ::= fact [ name ] block
assertDecl ::= assert [ name ] block
funDecl ::= fun name [ [ decl,* ] ] : expr { expr }
predDecl ::= pred name [ [ decl,* ] ] block
expr ::= const | qualName | @name | this | unOp expr
    | expr binOp expr | expr arrowOp expr | expr [ expr,* ]
    | expr [ ! | not ] compareOp expr
    | expr ( => | implies ) expr else expr
    | quant decl,+ blockOrBar | ( expr ) | block
    | { decl,+ blockOrBar } | expr'
const ::= none | univ | iden
unOp ::= ! | not | no | mult | set | ~ | * | ^
    | eventually | always | after
binOp ::= || | or | && | and | <=> | iff | => | implies
    | & | + | - | ++ | <: | :> | . | until | release
arrowOp ::= [ mult | set ] -> [ mult | set ]
compareOp ::= in | =
letDecl ::= name = expr
block ::= { expr* }
blockOrBar ::= block | | expr
quant ::= all | no | mult
checkCmd ::= check qualName [ scope ]
scope ::= for number [ but typescope,+ ] | for typescope,+
typescope ::= [ exactly ] number qualName
qualName ::= [ this/ ] ( name/ )* name
```

Figure 3: Concrete syntax of the Electrum language (additions w.r.t. the Alloy syntax are underlined).

variable expressions. Finally, the fact `traces` restricts the acceptable states of the system: a state is either the initial one or obtained from the application of the actions. Over this specification, the assertion `NoBadEntry` verifies whether there are unwanted room entries. This assertion is referred by the check command immediately below that instructs the model-checker to test the assertion for a scope of at most 3 elements per signature (it also defines the scope for time instants if the analysis is performed by the bounded model-checker, as will be described in Section 3.2).

The flexibility of the language allows the adoption of varied specification idioms that could be cumbersome in more rigid specification languages. For instance, in the event idiom, actions are embodied by model elements rather than by predicates. This allows the developer to define a hierarchy of actions, allowing the sharing of parameters (*e.g.*, in the room locking system, every action has a guest parameter) and of constraints (*e.g.*, in the front desk actions, the frame condition on the room coded keys is shared), resulting in simpler and more manageable specifications.

Figure 5 depicts an excerpt of the hotel room locking system specified with the event idiom in Electrum, where each action is embodied by a variable signature: the presence of an event atom in an instant denotes the occurrence of that action. This excerpt is an alternative to the check-in action predicate specified in Fig. 4. When defining these event signatures, one must be aware of the bounded nature of the universe in order to not restrict the application of actions. Here, the multiplicity **`one`** of the abstract `Event` signature, from which the concrete events inherit, forces the existence

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

sig Key {}

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

fact DisjointKeySets {
  keys in Room lone → Key }

one sig FD {
  var lastKey: Room → lone Key,
  var occupant: Room → Guest }

sig Guest {
  var gKeys: Key }

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

pred checkin[g: Guest, r: Room, k: Key] {
  g.gKeys' = g.gKeys + k
  no FD.occupant[r]
  FD.occupant' = FD.occupant + r → g
  FD.lastKey' = FD.lastKey ++ r → k
  k = nextKey[FD.lastKey[r], r.keys]
  all gg: Guest - g | gg.gKeys' = gg.gKeys
  all r: Room | r.currentKey' = r.currentKey }

...

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

fact traces {
  init
  always | some g: Guest, r: Room, k: Key |
    entry[g, r, k] or checkin[g, r, k] or checkout[g] }

assert NoBadEntry {
  always | all r: Room, g: Guest, k: Key |
    entry[g, r, k] and some FD.occupant[r] =>
      g in FD.occupant[r] }

check NoBadEntry for 3
  but 5 Time // Time scope only for bounded verification
```

Figure 4: Hotel room locking system under Electrum.

```
one var abstract sig Event {
  g: Guest }

var abstract sig FDEvent extends Event { } {
  currentKey' = currentKey }

var sig Checkin extends FDEvent {
  r: Room,
  k: Key } {
  g.gKeys' = g.gKeys + k
  no FD.occupant[r]
  FD.occupant' = FD.occupant + r → g
  FD.lastKey' = FD.lastKey ++ r → k
  k = nextKey[FD.lastKey[r], r.keys]
  all gg: Guest - g | gg.gKeys' = gg.gKeys }
```

Figure 5: Excerpt of the hotel room locking system under Electrum in the event idiom.

```
abstract sig Feature {}
one sig FIdle, FExecutive, FPark extends Feature {}

sig Product in Feature {} {
  FIdle + FPark not in this }

sig Floor {} {
  one b: LandingButton | b.floor = this
  one b: LiftButton | b.floor = this }

abstract sig Button { floor: one Floor }
sig LandingButton, LiftButton extends Button {}

var one sig Current in Floor {}
var lone sig Open, Up {}
var sig Pressed in Button {}

...

pred prop {
  always { all f: Floor | floor.f&LiftButton in Pressed =>
    eventually { current = f && some Open } } }

check { FIdle = Product => prop } for 6 but 10 Time
```

Figure 6: Excerpt of the of the elevator SPL under Electrum.

of exactly one event at each instant, although the concrete event signature to which it belongs may vary in time (this also simplifies the fact `traces`, that will only be required to enforce the `init` constraint). Parameters of the actions are embodied by the event-signature fields: since all the actions in the hotel room locking system have a guest parameter, this field is defined at the top-level event signature. Another abstract signature `FDEvent` represents every action that occurs at the front desk, enforcing the frame condition on the door locks' coded keys. The concrete event signatures then define the specific constraints that restrict their occurrence at each instant. Note how, in the check-in action presented in Fig. 5, the constraint is identical to the one defined in the predicate idiom in Fig. 4 (modulo the frame condition), and thus no additional burden was imposed on the developer. The remaining operations would be defined in a similar manner.

Figure 6 depicts an excerpt of a possible specification of the elevator SPL in Electrum to illustrate its potential to handle systems with variability (the actions are omitted). Here, features are simply declared as static signatures, a product being simply a subset of those features. Conflicts between features are enforced through constraints over products. The floors and the respective buttons – one landing and one elevator button per floor – are also static and defined by structural constraints. The remaining signatures depict the variable components of the model. At each instant, one floor marks the current position of the elevator; "lone" variable signatures act as temporal Boolean variables, that denote whether the elevator is open and moving in the upward direction; finally, a set of buttons is selected as pressed at each moment. Properties can then be checked for arbitrary or particular products, *e.g.*, the check command in the excerpt checks whether calls from elevator buttons are eventually answered for products that only implement the idle feature.

3.1.2 *Electrum Kernel*

Following the approach of [14, App. C], we simplify the presentation of the semantics of our framework by considering a stripped-down language, dubbed Electrum Kernel, focusing only on formulas and relational terms. The abstract syntax of Electrum Kernel is shown in Fig. 7. For constraints and relational expressions, the translation from Electrum is relatively straightforward and follows that of Alloy kernel (specifically, in formulas, the dual logical operators and connectives may be defined in the obvious way).

In Electrum Kernel, the main concept is that of relation. We assume the existence of a set $\mathcal{R}$ of *variable* relations, which

```
formula ::= not formula | after formula
          | always formula | eventually formula
          | formula until formula | formula and formula
          | term in term | all decl | formula
   term ::= x ∈ Var | r ∈ ℛ | ^term | ~term
          | term & term | term × term | term . term
          | term' | { decl+ | formula }
   decl ::= x : term
```

Figure 7: Electrum Kernel abstract syntax.

are declared with their arity. Signatures and fields declared in Electrum are translated into Electrum Kernel relations (unary relations in the case of signatures). We also assume the existence of a set *Var* of first-order variables. Additional information expressed in signature and field declarations in Electrum (multiplicities, the fact that some signatures and fields are static, signature hierarchy and local facts) must be specified by formulas in Electrum Kernel (in our prototypes, more efficient encodings are implemented).

In order to illustrate the translation from Electrum to Electrum Kernel, let us consider the following example:

```
abstract sig A { r: some A }
var sig B,C extends A {}
```

In the corresponding Electrum Kernel specification, three unary relations and one binary relation are declared.

```
Relations: A(1), B(1), C(1), r(2)
```

The fact that `A` is not a variable signature is expressed by the formula `always A' = A`. The fact that `B` and `C` extend `A`, which is abstract, can be expressed by the formula `always (A = B + C and no B & C)`.

Now, the typing and the multiplicity constraint related to the field `r` are expressed as follows:

```
always r in A → A
always all a: A | some a.r
```

Local facts are surrounded by an `always` operator after translation, forcing them to hold in every instant of time.

3.1.3 FOLTL

The semantics of Electrum Kernel is expressed *via* a translation into First-Order Linear Temporal Logic (FOLTL) [13, 3]. Here we briefly describe its syntax and semantics.

Definition 1. Given mutually-disjoint sets $\mathcal{V}$ and $\mathcal{P}$ of (resp.) variables and predicates (with their arity), the syntax of FOLTL formulas is given as follows[5]:

$$\varphi ::= P(x_1, \ldots, x_k) \mid x_1 \doteq x_2 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x.\varphi \mid \mathbf{X}\,\varphi \mid \mathbf{G}\,\varphi \mid \mathbf{F}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$$

with $x_i \in \mathcal{V}$ and $P \in \mathcal{P}$ (of course, $P(x_1, \ldots, x_k)$ is a formula only if the arity of P is k).

Derived constructs ($\exists$, $\vee$, $\Rightarrow$) can be defined in the obvious way. $\mathbf{X}\,\varphi$ (read "next φ") means that φ is true in the *next* instant, $\mathbf{G}\,\varphi$ (read "always φ") means that φ will *always* be true, $\mathbf{F}\,\varphi$ (read "eventually φ") means that φ will eventually be true and $\varphi\,\mathbf{U}\,\psi$ (read "φ until ψ") means that φ is true and remains true until ψ becomes true. The "release"' operator can be derived from $\varphi\,\mathbf{U}\,\psi$ in the usual way.

[5]The symbol $\doteq$ stands for equality in FOLTL in order to avoid notation clashes.

FOLTL is provided with both unbounded and bounded semantics. For the unbounded semantics, time is interpreted over the set $\mathbb{N}$ of non-negative integers. Each first-order variable is interpreted over the domain $\mathcal{D}$.

Definition 2. A *model* for FOLTL is a pair $\mathcal{M} = (\mathcal{D}, \rho)$ where: (1) the set $\mathcal{D}$ is the *domain* of first-order variables and (2) ρ maps each predicate $P \in \mathcal{P}$ at each instant $i \in \mathbb{N}$ to a relation $\rho(P, i) \subseteq \mathcal{D}^k$, where k is the arity of P.

The satisfaction of a formula by a model is then defined as follows.

Definition 3. Given a model $\mathcal{M}$, a formula φ, an instant $i \in \mathbb{N}$, and an environment σ, mapping each free variable x to an element in the domain $\mathcal{D}$, the satisfaction relation $\mathcal{M}, \sigma, i \models \varphi$ is defined inductively as follows.

$\mathcal{M}, \sigma, i \models x \doteq y$ if $\sigma(x) = \sigma(y)$
$\mathcal{M}, \sigma, i \models P(x_1, \ldots, x_n)$ if $(\sigma(x_1), \ldots, \sigma(x_n)) \in \rho(P, i)$
$\mathcal{M}, \sigma, i \models \neg\varphi$ if $\mathcal{M}, \sigma, i \not\models \varphi$
$\mathcal{M}, \sigma, i \models \varphi \vee \psi$ if $\mathcal{M}, \sigma, i \models \varphi$ or $\mathcal{M}, \sigma, i \models \psi$
$\mathcal{M}, \sigma, i \models \forall x.\varphi$ if for all $a \in \mathcal{D}, \mathcal{M}, \sigma[x \mapsto a], i \models \varphi$
$\mathcal{M}, \sigma, i \models \mathbf{X}\,\varphi$ if $\mathcal{M}, \sigma, i+1 \models \varphi$
$\mathcal{M}, \sigma, i \models \mathbf{G}\,\varphi$ if for each $j \geq i, \mathcal{M}, \sigma, j \models \varphi$
$\mathcal{M}, \sigma, i \models \mathbf{F}\,\varphi$ if there is $j \geq i$ s.t. $\mathcal{M}, \sigma, j \models \varphi$
$\mathcal{M}, \sigma, i \models \varphi\,\mathbf{U}\,\psi$ if there exists $j \geq i$ s.t. $\mathcal{M}, \sigma, j \models \psi$, and for all $i \leq k < j, \mathcal{M}, \sigma, k \models \varphi$.

A formula φ without free variables is satisfiable if and only if there exists a model $\mathcal{M}$ such that $\mathcal{M}, \emptyset, 0 \models \varphi$, which is simply denoted by $\mathcal{M} \models \varphi$.

The bounded semantics of FOLTL can be derived from the unbounded one following the standard technique described in [2]. In (bounded) models of size k (denoting traces with k states), ρ is partial function, with domain $\{0 \ldots k-1\}$. If such trace has a loop to time $0 \leq l < k-1$, *i.e.*, the value of $\rho(s, k-1)$ is equal to the value of $\rho(s, l)$ for all s, then the semantics is the same as in the unbounded case, after unrolling the model to be defined over $\mathbb{N}$. If the trace has no loop, then the semantics has to be slightly adjusted, as such traces cannot be considered valid models of (invariant) formulas of type $\mathbf{G}\,\varphi$ (as there could be a state after $k-1$ that violates φ), and only of formulas of type $\mathbf{F}\,\varphi$ (as discussed in [2], $\mathbf{G}$ and $\mathbf{F}$ are no longer duals in a bounded semantics, hence the inclusion of both in the language kernel).

Note that we do not follow the temporal semantics from TLA$^+$, which makes formulas invariant under stuttering (for compositionality purposes), but use instead the classic semantics of temporal logic, which allows to specify behaviors that may not be invariant under stuttering.

3.1.4 From Electrum Kernel to FOLTL

The essence of the interpretation of Electrum Kernel into FOLTL boils down to removing relational terms so that we end up with formulas only. This is a standard approach when embedding the relational logic of Alloy into FOL [10, 9]. Thus, the main operation consists in removing all membership and inclusion statements present in Electrum Kernel formulas and replacing them with corresponding FOLTL subformulas. Hence the semantic map (denoted $[\![-]\!]$) relies on a function $[- \in -]$ which, given a pair of a tuple of variables and a term, yields a formula stating that the former is a member of the latter. For the sake of readability, tuples are written

$$
\begin{aligned}
[\![\texttt{not } f]\!] &= \neg[\![f]\!] \\
[\![\texttt{after } f]\!] &= \mathbf{X}[\![f]\!] \\
[\![\texttt{always } f]\!] &= \mathbf{G}[\![f]\!] \\
[\![\texttt{eventually } f]\!] &= \mathbf{F}[\![f]\!] \\
[\![f_1 \texttt{ until } f_2]\!] &= [\![f_1]\!] \,\mathbf{U}\, [\![f_2]\!] \\
[\![f_1 \texttt{ and } f_2]\!] &= [\![f_1]\!] \wedge [\![f_2]\!] \\
[\![t_1 \texttt{ in } t_2]\!] &= \forall \vec{x}.[\vec{x} \in t_1] \Rightarrow [\vec{x} \in t_2] \\
&\quad \text{where } \vec{x} \text{ are } \textit{fresh} \text{ variables;} \\
[\![\texttt{all } x : t \mid f]\!] &= \forall x.[x \in t] \Rightarrow [\![f]\!] \\
[x \in y] &= x \doteq y \\
[\vec{x} \in r] &= \overline{r}(\vec{x}) \\
[\langle x_1, x_2\rangle \in \hat{}t] &= \text{there are } y_1, \ldots, y_n \text{ such that} \\
&\quad y_1 \doteq x_1 \wedge y_n \doteq x_2 \wedge \bigwedge_{i<n} [\langle y_i, y_{i+1}\rangle \in t] \\
[\langle x_1, x_2\rangle \in \sim t] &= [\langle x_2, x_1\rangle \in t] \\
[\vec{x} \in t_1 \,\&\, t_2] &= [\vec{x} \in t_1] \wedge [\vec{x} \in t_2] \\
[\vec{x} \in t_1 \times t_2] &= [\vec{y} \in t_1] \wedge [\vec{z} \in t_2] \\
&\quad \text{with } \vec{x} = \vec{y}\vec{z} \\
[\vec{x} \in t_1.t_2] &= \exists u.[\vec{y}u \in t_1] \wedge [u\vec{z} \in t_2] \\
&\quad \text{where } \vec{x} = \vec{y}\vec{z}, \text{ and } u \text{ is a } \textit{fresh} \text{ variable,} \\
[\vec{x} \in t'] &= \mathbf{X}[\vec{x} \in t] \\
[\vec{x} \in \{\vec{y} : \vec{t} \mid f\}] &= \Big(\bigwedge_{1 \leqslant i \leqslant |\vec{x}|} [x_i \in t_i] \Big) \wedge [\![f\{\vec{y} \leftarrow \vec{x}\}]\!] \\
&\quad \text{where } f\{\vec{y} \leftarrow \vec{x}\} \text{ is the usual substitution.}
\end{aligned}
$$

Figure 8: From Electrum Kernel to FOLTL (cf.. Def. 4).

as vectors, their concatenation is denoted by juxtaposition and $|\cdot|$ stands for their length.

Definition 4. The formal semantics of Electrum Kernel formulas into FOLTL formulas is defined by structural induction according to Fig. 8.

3.2 Verification

While Electrum supports the definition of both rich specifications and properties to be checked over them, it is only useful if accompanied by effective model-checking techniques. Fitting the dual nature of the problem at hand, two distinct approaches to the model-checking of Electrum specifications were explored: one bounded and another unbounded. Before detailing these, we first describe the commands provided by Electrum for formal analysis.

3.2.1 Commands and scopes

As presented in Section 3.1.1, an Electrum specification integrates execution commands for the model-checker. The difference between both verification approaches lies in the way time is handled. In the bounded approach, time is internally handled as a signature and is thus bounded the same way as others, and in the other time is left unbounded. The maximum number of time instants considered by the bounded approach is also defined in the scope of the commands (which is simply ignored by the unbounded approach).

Then, given an Electrum model and a scope, every signature or field is instantiated depending on the bound of signatures. The model and the "run" (or "check") command give rise to a formula φ_M and a formula φ_r (or φ_c), respectively. Finally, if the command is a "run", the formula $\varphi_M \wedge \varphi_r$ is checked for *satisfiability*; otherwise we check whether $\varphi_M \Rightarrow \varphi_c$ is *valid*. In the following, we detail how these verification problems translate in practice.

3.2.2 Bounded model-checking

The bounded semantics of FOLTL described in Section 3.1.3 can be directly encoded into Alloy itself, as described in [8], by explicitly introducing a time signature with a total order imposed over it to represent the trace; potential loops are represented by a relation from the last time atom to a previous one. Our bounded model-checker is implemented using this alternative encoding, and deployed as a new version of the Alloy Analyzer, to minimize the adoption time by Alloy practitioners. This Analyzer not only generates a single counter-example (depicted visually), but allows the user to iterate over all possible counter-examples (within the specified bounds) that broke the specified properties, thus providing the user with a wider perception of what may be the problems of the specification. Note that this bounded model-checking procedure is iterative, checking the properties for increasing trace sizes up to the specified scope on time, stopping along the way if a counter-example is found.

3.2.3 Unbounded model-checking

This technique is implemented in a prototype called the Electrum Analyzer. It relies on a direct embedding into the nuXmv tool[6] which implements various algorithms performing unbounded model-checking [4] (we currently rely on the so-called "k-liveness" algorithm). Its free-software predecessor, NuSMV, can also be used but it is far less efficient on the examples we have studied so far. The principle of the translation proceeds as described before, chaining a translation from Electrum to Electrum Kernel, then to FOLTL and finally to LTL. Compared to the semantics presented in this paper, several simple optimizations are also implemented (smarter optimizations are left for future work).

Now, nuXmv expects a description of a transition system and a formula to check on the latter, whereas an Electrum model is essentially a formula specifying *a set of* transition systems (that satisfy it). Hence the generated SMV model does not contain an explicit transition system: (i) signatures and fields give rise to "frozen" or plain variables depending on their status (static or variable); (ii) various formulas related to the typing and inclusion of signatures and fields are combined to form an "invariant" section in the file. This is important as it allows to constrain and reduce the size of the state space. A possible improvement would be to infer a (non-deterministic) transition system and add a corresponding SMV "assign" section, restricting the state space further. Then, a so-called SMV "LTL specification" is produced that represents the formula to be verified[7].

Finally, it should be remarked that the present approach allows to perform verification on an unbounded time horizon, but it does not allow to perform scenario exploration, namely

[6] Available at https://nuxmv.fbk.eu.

[7] This formula is in practice dualized (w.r.t. the description in Section 3.2.1) as nuXmv expects specifications expressed as validity problems rather than as satisfiability ones.

iterate over counter-examples, the same way the bounded approach does. Therefore, regardless of practical performance results, both approaches are complementary.

4. EVALUATION

This section presents the empirical evaluation of our language and the proposed model-checking techniques. Concretely, we aim to assess how the performance of the proposed bounded and unbounded checking techniques compare with each other and with other existing, similar approaches.

To answer these questions, a detailed evaluation of the proposed techniques under the examples from Section 1.1 is presented, as well as a summary of the results for an additional specification with rich configurations.

Regarding the hotel room locking system, two versions are considered: `Hotel (1)` checks the desired safety property and thus leads to counter-examples, and `Hotel (2)` checks the same property, but with an additional constraint that prohibits any other action to occur between a guest checking in and entering a room, and, as a result, is correct. Recall nonetheless that the counter-example in `Hotel (1)` does not occur with every configuration. In these examples, the size of the model n denotes the number of keys, rooms and guests available in the universe. For the spanning tree algorithm, we consider the following verification goals: `Span (1)` checks the liveness property without enforcing fairness, thus producing counter-examples; `Span (2)` checks for the liveness property but with fairness enforced, and thus is correct; and `Span (3)` checks for the safety property and does not generate counter-examples. Here, model size n denotes the number of processes and tree levels in the universe. For the SPL example we consider two check commands: `Elevator (1)` tests the liveness property for products implementing only the idle feature, which holds; `Elevator (2)` tests the property for arbitrary products which does not hold. Here n denotes the exact number of floors.

Additionally we explore another distributed algorithm over arbitrary topologies that is packaged with the Alloy Analyzer, whose specification was quite simplified with Electrum. Concretely, we consider a distributed algorithm for the election of a leader in a network with ring topology, inspired by the specification presented in [14]. This specification is checked for two temporal properties: that at least one leader is eventually elected (a liveness property) and that at most one leader is elected (a safety property). Different versions of the specification were considered: `Ring (1)` checks the specification for liveness without fairness enforced, `Ring (2)` checks for liveness with fairness, and `Ring (3)` checks for safety, which holds.

As has already been stated in Section 2, the two existing techniques that we believe are best suited to model-check specifications with rich configurations are Alloy and TLA$^+$. Since our bounded technique actually relies on Alloy, we will focus on comparing the performance of our two techniques with that of TLA$^+$/TLC.

All tests were run multiple times using Alloy 4.2 with the MiniSat solver and nuXmv 1.0.1, on a 1,8 GHz Intel Core i5 with 4 GB memory running OS X 10.10. TLC 2.05 was used for the TLA$^+$ tests. Note that, for the unbounded approach, we pre-process Electrum files to replace every command by all possible combinations of the said command with *exact scopes*. Then we generate as many corresponding SMV files, and run nuXmv in parallel on all CPU cores using *GNU parallel* [23], starting with the smallest scopes and stopping immediately if a property is refuted.

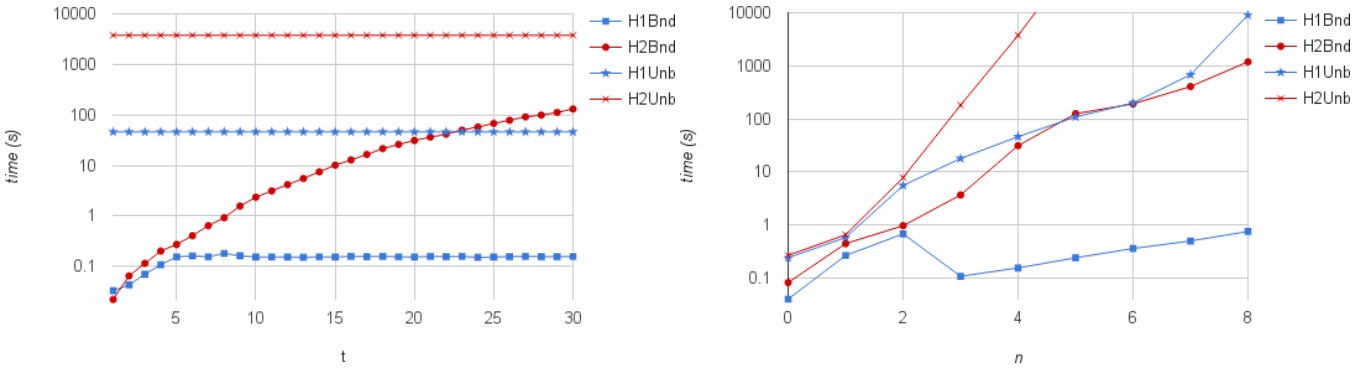
4.1 Results

Figures 9a and 10a present the performance for the hotel room locking system and the spanning tree algorithm for a fixed size $n = 4$ and increasing trace length t for the bounded scenario. Labels `H`n and `S`n stand for the `Hotel (`n`)` and `Span (`n`)` scenarios respectively, and `Bnd` and `Ubd` stand for the bounded and unbounded techniques.

In the unbounded technique the properties are checked for arbitrary trace lengths, and thus their performance is depicted by a constant function in these graphs. Note also that in the bounded scenario, each trace length aggregates the time spent verifying the smaller traces, *i.e.*, the time spent to assess that a property holds for $t = 4$ includes checking the property for lengths 1, 2 and 3. In the scenarios where there are counter-examples to be found, the performance of the bounded technique stabilizes at the t value where that counter-example finally occurs (*e.g.*, 5 for `Hotel (1)`). This is due to the iterative nature of the technique: once a counter-example is found, the procedure is stopped and not run for larger t values. In contrast, when no counter-examples occur, the technique must be run for every trace length value and keeps increasing with t. The performance of the unbounded approach is also better when there are counter-examples to be found due to the parallelization of the procedure that stops as soon as a counter-example is found. For the scenarios without counter-examples the procedure must check the properties for every launched process. The bounded technique outperforms the unbounded one for smaller trace lengths, but their performance starts to converge as the t value increases. This reinforces the common policy of relying on bounded techniques to quickly discard trivial counter-examples, and move on to unbounded techniques only when the confidence level is high enough.

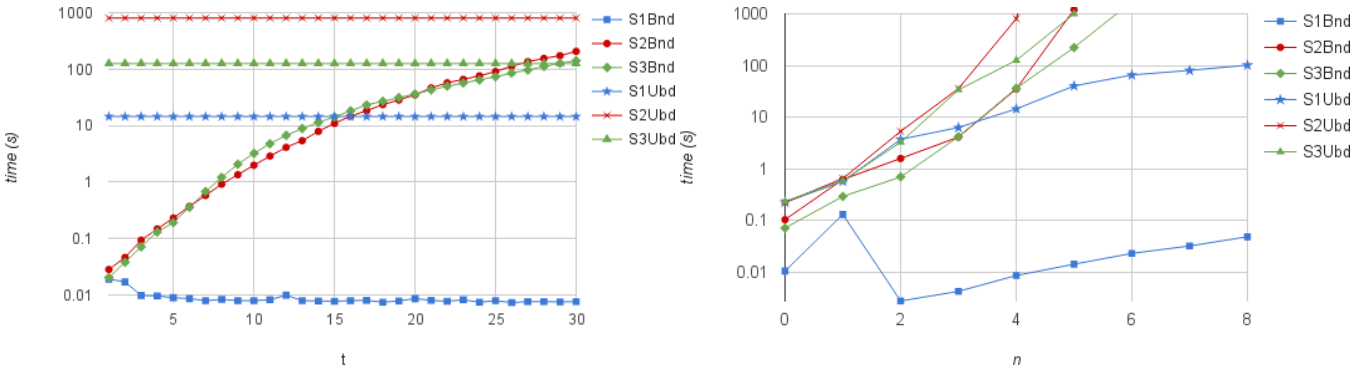
In contrast, Figs. 9b and 10b present the performance of the model-checking techniques for increasing model size n and fixed trace length $t = 20$ (for the bounded technique). Again, for the bounded scenario the results aggregate the time spent for trace lengths up to 20. At such t value, the performance of the two model-checkers is almost similar for the considered examples, although the bounded technique still outperforms the unbounded one. For `Hotel (1)` and `Span (1)`, the performance of the bounded technique improves at $n = 3$ and $n = 2$, respectively, because these are the sizes where counter-examples first appear, and for smaller sizes the properties must be checked for every trace length.

Table 1 summarizes the evaluation for the explored examples and reinforces the conclusions discussed above. It also presents the number of valid configurations for each of the scenarios. Note how the hotel room locking system already has 18960 valid configurations for $n = 4$. The Alloy Analyzer has a powerful *symmetry breaking* algorithm that infers an equivalence class within the search space, highly reducing the number of explored models. For example, in the same example, the number of non-symmetric configurations is only 520. By performing the bounded model-checking of Electrum via an embedding into Alloy we take advantage of this mechanism, hence the quite positive results for this technique. This also points for interesting possible optimizations of the unbounded model-checking technique, namely its parallelization based on unique non-symmetric configurations.



(a) Fixed $n = 4$ and increasing t. (b) Fixed $t = 20$ and increasing n.

Figure 9: Performance tests for `Hotel (1)` and `Hotel (2)`.



(a) Fixed $n = 4$ and increasing t. (b) Fixed $t = 20$ and increasing n.

Figure 10: Performance tests for `Span (1)`, `Span (2)` and `Span (3)`.

Spec.	n	#Cfg (sym)	Type	Holds	Bnd (s)	Ubd (s)
Hotel (1)	4	18960 (520)	S	×	0.2	47.5
Hotel (2)	4	18960 (520)	S	✓	31.2	3844.2
Span (1)	4	216 (16)	L	×	0.0	14.5
Span (2)	4	216 (16)	L	✓	35.1	803.9
Span (3)	4	216 (16)	S	✓	36.4	125.8
Ring (1)	3	40 (9)	L	×	0.0	3.4
Ring (2)	3	40 (9)	L	✓	2.0	156.9
Ring (3)	3	40 (9)	S	✓	31.9	35.2
Elevator (1)	3	1 (1)	L	×	0.0	12.3
Elevator (2)	3	48 (48)	L	✓	21.1	2246.8

Table 1: Summary of the performed tests (for $t = 20$ for the bounded scenarios); S is for "safety", L is for "liveness".

For the comparison with TLC, we encoded the considered examples in TLA$^+$. In general, our model-checkers outperform TLC when there are counter-examples to be found. For instance, for `Hotel (1)` with $n = 4$, TLC takes 545.2 seconds to generate a counter-example. In contrast, TLC outperforms our unbounded technique when there are no counter-examples. For instance, for `Hotel (2)` TLC takes 256.6 seconds to check that the specification is correct. Note that our unbounded technique, unlike TLC which imposes constraints on how actions can be specified, can handle actions specified in a very liberal declarative style. Nonetheless, our experiments show that TLC is heavily affected by the number of valid configurations (worsened by the fact that it is not able to explore symmetry), since these affect the number of initial states that must be explored. In fact, in the hotel scenarios, for sizes higher than $n = 4$, TLC runs out of memory when calculating the initial states. Our unbounded model-checker is still able to terminate in such scenarios.

5. CONCLUSION

This work proposed a language, Electrum, mixing the best aspects of both Alloy and TLA$^+$, currently two of the most popular formal specification languages, and that we believe hits the sweet spot for the specification of dynamic systems requiring rich declarative specifications (both of structural aspects, namely configurations, and dynamics). Two model-checking tools for this language were also developed, one bounded and the other unbounded, that our preliminary evaluation already showed to be competitive, performance wise, with existing model-checkers, in particular TLC. The bounded model-checker is useful at early analysis stages, namely excelling at founding and iterating over counter-examples, while the unbounded one, naturally slower, should be used afterwards for further confirmation of the results.

In the future we intend to improve the Electrum framework in two key aspects: first, building on previous work on scenario exploration [19, **?**, 17], we intend to improve the counter-example generation and iteration features, by allowing the user to parameterize the tool to prioritize certain aspects, for example, showing first counter-examples similar to those found in previous versions of the specification; second, we intend to improve the efficiency of the unbounded model-checking technique in order to take advantage of symmetry breaking, namely explore a "hybrid" verification approach where a bounded analyzer would compute in advance (bundles of) non-symmetric static configurations, to be used to optimize the LTL specifications passed to each (parallel) unbounded checking process.

6. REFERENCES

[1] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *ISSRE 2010*, pages 161–170. IEEE, 2010.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS 1999*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.

[3] T. Braüner and S. Ghilardi. First-order modal logic. *Handbook of modal logic*, 3:549–620, 2007.

[4] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv symbolic model checker. In *CAV 2014*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

[5] F. S. Chang and D. Jackson. Symbolic model checking of declarative relational models. In *ICSE 2006*, pages 312–320, 2006.

[6] A. Classen, M. Cordy, P. Heymans, A. Legay, and P. Schobbens. Model checking software product lines with SNIP. *STTT*, 14(5):589–612, 2012.

[7] A. Classen, M. Cordy, P. Heymans, A. Legay, and P. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Sci. Comput. Program.*, 80:416–439, 2014.

[8] A. Cunha. Bounded model checking of temporal formulas with Alloy. In *ABZ 2014*, pages 303–308, 2014.

[9] A. Cunha, A. Garis, and D. Riesco. Translating between Alloy specifications and UML class diagrams annotated with OCL. *Software & Systems Modeling*, 14(1):5–25, 2015.

[10] A. A. El Ghazi and M. Taghdiri. Relational reasoning via SMT solving. In *FM 2011*, pages 133–148, June 2011.

[11] M. F. Frias, J. P. Galeotti, C. L. Pombo, and N. Aguirre. DynAlloy: upgrading Alloy with actions. In *ICSE 2005*, pages 442–451, 2005.

[12] M. F. Frias, C. L. Pombo, J. P. Galeotti, and N. Aguirre. Efficient analysis of DynAlloy specifications. *ACM Trans. Softw. Eng. Methodol.*, 17(1), 2007.

[13] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106(1–3):85 – 134, 2000.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, revised edition, 2012.

[15] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[16] N. Macedo and A. Cunha. Alloy meets TLA$^+$: An exploratory study. Manuscript available at http://alfa.di.uminho.pt/~nfmmacedo/publications/tlalloy15.pdf, 2015.

[17] N. Macedo, A. Cunha, and T. Guimarães. Exploring scenario exploration. In *FASE 2015*, volume 9033 of *LNCS*, pages 301–315. Springer, 2015.

[18] J. P. Near and D. Jackson. An imperative extension to Alloy. In *ABZ 2010*, pages 118–131, 2010.

[19] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. Aluminum: principled scenario exploration through minimality. In *ICSE 2013*, pages 232–241. IEEE/ACM, 2013.

[20] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services uses formal methods. *Commun. ACM*, 58(4):66–73, 2015.

[21] M. Plath and M. Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001.

[22] I. Shlyakhter, M. Sridharan, and D. Jackson. Analyzing distributed algorithms with first-order logic. available at http://sdg.csail.mit.edu/pubs/alloy-distalg.pdf, 2002.

[23] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[24] A. Vakili and N. A. Day. Temporal logic model checking in Alloy. In *ABZ 2012*, pages 150–163, 2012.