A Study of the Electrum and DynAlloy Dynamic Behavior Notations

César Cornejo, Germán E. Regis, Nazareno Aguirre, and Marcelo F. Frias

***Abstract*—Alloy is a formal specification language, which despite featuring a simple syntax and relational semantics, is very expressive and supports efficient automated specification analysis, based on SAT solving. While the language is sufficiently expressive to accommodate both *static* and *dynamic* properties of systems within specifications, the latter kind of properties require intricate, ad-hoc, constructions to encode system executions. Thus, extensions to the language have been proposed, that internalize these encodings and provide analysis techniques, specifically tailored to properties of executions. In this paper we study two particular extensions to Alloy that incorporate elements for the specification of properties of executions. These are DynAlloy, whose syntax and semantics are inspired by dynamic logic, and Electrum, based on linear-time temporal logic and inspired by languages such as TLA+. We analyze and compare the syntactic characteristics of the languages, their corresponding expressiveness, and the effectiveness and efficiency of their associated analysis tools. The comparison is based on a set of Alloy specifications that are taken from the literature and demand dynamic behavior analysis, including an Alloy model of the Chord ring-maintenance protocol, that drives our qualitative comparison of the notations.**

***Index Terms*—Formal methods, software specification, alloy, automated analysis.**

I. Introduction

SOFTWARE modeling is an important activity of many software development processes. The reason is simple: by building models of the software to be developed (and its environment), engineers can anticipate potential flaws in their designs, through various activities. Software models enable, among other tasks, the early communication and discussion of design decisions, and the identification of assumed constraints of the problem domain, at a higher level of abstraction than that provided by the software's source code, and before such code is even produced [7], [28]. This importance is recognized by the broad availability of software modeling notations, among which *formal* notations have a distinctive position [8]. The relevance of formal approaches to modeling lies, among various reasons, in the fact that notations with a formal semantics lead to specifications with unambiguous interpretations, that are better suited for analysis [32]. This is generally achieved at the expense of diminishing specification understandability (compared to informal notations), since formal specifications tend to be more detailed, and require a knowledge of the logical or mathematical foundations behind the notations [63]. Thus, finding appropriate abstractions for capturing software behavior and its intended properties, and internalizing them into simple (but precise) notations or notation patterns, is an important task in the design of formal specification languages.

A particular example of the above-described situation arises in the context of Alloy [38]. Alloy is a formal specification language that features a simple syntax and relational semantics, incorporates abstractions similar to common concepts from object oriented design, and is designed with an emphasis in *automated* analysis, provided through a SAT-based instance finding mechanism [37]. The versatility of Alloy, and its fully automated analysis mechanism, has led to its use in a wide range of applications [6], [16], [29], [40]. The notation is very expressive, allowing one to straightforwardly capture, in a formal way, the typical static modeling constructions for modules, module operations and properties. Moreover, the language is sufficiently expressive to also characterize *dynamic behavioral properties*, i.e., properties of system executions [39]. The ability of encoding dynamic behavioral properties into the language enables one to use Alloy's automated analysis support to analyze dynamic properties too, without the need for additional analysis tools and techniques for this kind of properties.

Capturing dynamic behavior and properties of execution traces is highly relevant in the context of Alloy. For various application domains, many properties of interest involve system executions [11], [16], [26], and some tools and notations that use Alloy for analysis require expressing and analyzing such properties. Some relevant examples are tools for bounded verification of annotated code [16], [26]; languages for class and feature modeling with support for dynamism [53]; encodings of notations for dynamic behavior, such as activity diagrams or business process modeling, into Alloy [43], [62]; among others.

Although, as we mentioned, Alloy is sufficiently expressive to capture properties of system executions, such properties require intricate *ad-hoc* characterizations, that reduce model understandability. The issue here has to do with having to explicitly

Manuscript received 1 March 2021; revised 6 July 2023; accepted 18 September 2023. Date of publication 29 September 2023; date of current version 14 November 2023. This work was supported in part by ANPCyT PICTs 2017-2622, 2019-2050, 2020-2896, an Amazon Research Award, and in part by EU's Marie Sklodowska-Curie under Grant 101008233 (MISSION). Recommended for acceptance by P. Spoletini. *(Corresponding author: César Cornejo.)*

César Cornejo, Germán E. Regis, and Nazareno Aguirre are with CONICET, Buenos Aires 1425, Argentina, and also with the Department of Computer Science, FCEFQyN, University of Río Cuarto, Río Cuarto 5800, Argentina (e-mail: ccornejo@dc.exa.unrc.edu.ar)

Marcelo F. Frias is with the Department of Computer Science, The University of Texas at El Paso, TX 79968 USA.

Digital Object Identifier 10.1109/TSE.2023.3320625

capture execution traces as part of an Alloy formal model (i.e., capture traces "statically"), as opposed to these being *implied* by the semantics of trace-related constructs in the language. In an attempt to improve this situation, some extensions to Alloy have been proposed, to allow modelers to better capture system execution properties. Two particular notations that extend Alloy to characterize dynamic properties of systems are DynAlloy [21] and Electrum [44]. While these notations are similar in spirit, they also have some substantial differences, that may make them particularly appropriate for different application domains, for specifications with different characteristics, or for different analysis purposes. This motivates the present article, where we perform a study of these notations, with the aim of understanding the languages' characteristics, and providing modelers and developers of other tools and techniques employing Alloy with guidelines that may make one choose the most appropriate notation, according to their specific needs, or characteristics of their modeling intentions. Thus, in this paper we perform a detailed comparison of DynAlloy and Electrum, their corresponding expressive powers, and general language characteristics. As we previously mentioned, model analyzability is a primary concern in the context of Alloy. Both languages acknowledge this fact, and are accompanied by tool support for automated model analysis, that resort to SAT-based analysis through Alloy via different encodings (and in the case of Electrum, can also exploit alternative verification engines [44]). Then, we also assess the effectiveness and efficiency of these analysis tools. The comparison is driven by a set of well-known Alloy specifications demanding dynamic behavior analysis, taken from the literature.

II. ALLOY AND THE MODELING OF DYNAMIC BEHAVIOR

Alloy is a model-oriented formal specification language, whose underlying formalism is of a relational nature, and is called *relational logic* [38]. While, technically, Alloy is not object-oriented, it features a number of modeling constructions that resemble abstractions known to developers, making its syntax intelligible and elegant, while at the same time having a precise meaning. Specifications in Alloy are expressed around *signatures*, which define data domains, *fields*, that are associated with specific signatures and define relations, *predicates*, parameterized formulas in relational logic, and *functions*, parameterized relational expressions. Finally, formulas can be used to establish *facts*, constraints that are assumed valid in specifications (and thus constrain models), and *assertions*, intended properties of the model that need to be analyzed for validity (typically, only in bounded contexts).

As a very simple model illustrating Alloy's syntax, consider the (partial) specification of the river crossing puzzle [54], in Fig. 1. This is a classic puzzle, where a farmer carrying three objects, a fox, a chicken, and a sack of grain, needs to cross a river using a boat that can hold the farmer and at most one other object. The farmer cannot leave the fox and chicken alone, because the fox would eat the chicken; nor the chicken alone with the grain, since the chicken would eat the grain. How can the farmer take the objects one at a time, guaranteeing that he will reach the other side of the river with all the objects? In the Alloy specification for the puzzle, an `Object` data domain (captured through an *abstract* signature, i.e., a signature whose only associated elements are those of its extending signatures) is composed of 4 particular elements, namely `Farmer`, `Fox`, `Chicken` and `Grain` (each captured through a *one* signature, i.e., a signature forced to contain just a single element, and thus used to represent particular elements through singleton sets). A fact in this model defines the `eats` field of `Object` (itself a binary relation between objects). Finally, predicate `crossRiver` can be understood as an *operation* of the model, allowing it to change state. The state is characterized by parameters of the predicate. Primed parameters (in this case double-primed parameters, to distinguish them from primed variables in Electrum and Alloy) are used as a convention to model "post-states". The parameters of this predicate indicate that the state of the system is composed of two sets of objects (the objects on each side of the river). According to the predicate's definition, to cross from side `from` to side `to`, the farmer needs to be present in the former; when the farmer crosses the river to the `to` side, he can take an item with him, and items in the resulting "unsupervised" side eat each other (according to what `eats` prescribes). As it can be seen from this example, signatures can be used to capture state, and predicates to capture operations, as well as properties, of a formal model in Alloy. In particular, predicates can be used to characterize *state changing operations* (i.e., to capture operations that modify the system state), as is the case with `crossRiver`.

```
abstract sig Object {
    eats: set Object
}

one sig Farmer, Fox, Chicken, Grain extends Object { }

fact { eats = Fox->Chicken + Chicken->Grain }

pred crossRiver[from, from'', to, to'': set Object] {
    one x: from | {
      from'' = from - x - Farmer - from''.eats
      to'' = to + x + Farmer }
}
```

Fig. 1. (Partial) Alloy specification for the river crossing puzzle.

Through the above described constructions, Alloy models can be equipped with a wide variety of *static* properties of systems (i.e., properties that refer to the system description but not its implied execution traces). These can be captured as predicates, as constraints of models (facts), or assertions (intended properties), exploiting the expressive power of relational logic. It is worth remarking that relational logic is strictly more expressive than first-order logic (essentially, first-order logic with transitive closure) [38]. Moreover, Alloy has, from its inception, put an emphasis in automated analysis: predicates can be checked for bounded satisfiability, and assertions for validity within bounded contexts, by resorting to SAT solving [37]. Of course, Alloy's expressiveness makes the analysis of specifications based on SAT solving necessarily incomplete: one may find counterexamples of intended properties and instances of specified models in *bounded* scenarios, but the absence of such counterexamples or instances does not imply their nonexistence in a larger or unbounded scenario [35] (i.e., the

SAT-based decision procedures for relational logic satisfiability and validity that Alloy Analyzer implements are incomplete, according to the terminology in [41]).

Using predicates and simple combinations of these, one can capture single operations as well as operation compositions (e.g., sequential composition), and state properties regarding their execution. Moreover, Alloy is also sufficiently expressive to capture *dynamic* properties of systems, i.e., properties regarding system executions, i.e., traces of successive state changes achieved by system operations [38], [39]. This is achieved through an explicit model of execution traces, as shown in Fig. 2. This model extends the model presented in Fig. 1 with a definition of signature `State`, that makes explicit the fact that there are two sets of objects, one at each side of the river, a model of sequences of these states (`ordering`[1], imported from a library), and constraints indicating how the initial state of the system is configured (all objects on the `near` side of the river), and how successive states in every trace are related (through predicate `crossRiver`). Finally, a predicate defines a property of the last state of a trace, asking it to have all objects on the `far` side of the river, so that querying for the satisfiability of the predicate makes the solver to "solve" the puzzle (i.e., to produce a trace that satisfies all constraints, and leads all objects to the `far` side of the river).

While this mechanism to capture dynamic properties is undeniably powerful, it involves ad-hoc characterizations of state and state change, realized through explicit models of execution traces. This issue reduces model understandability, due to the fact that execution traces need to be manually modeled as part of the specification. These "manual" models of traces can also have, in many cases, a significant impact in analyzability, calling for intricate model optimizations that may reduce their readability even further [21], [22], [23].

These issues had led to proposals for extensions to Alloy, to better capture dynamic behavior. We describe these extensions in the next section.

III. Extensions to Support Dynamic Behavior in Alloy

The need to capture and analyze properties regarding system execution traces is a recurrent issue for Alloy users. This situation led to the emergence of extensions to the language, to support the specification and analysis of dynamic properties of systems. Two particular extensions are Electrum and DynAlloy. We describe these extensions through an example, using the river crossing puzzle as a means for comparison.

A. Electrum

Electrum [44] is an extension of Alloy, which enriches Alloy's syntax to allow for the specification of execution traces and their properties, maintaining Alloy's declarative style. More specifically, Electrum incorporates temporal logic operators into Alloy, in order to prescribe the system behavior as well as intended system properties. Electrum is inspired by the *temporal logic of actions* (TLA) [42], and in order to indicate which signatures or fields of a signature are "mutable", i.e., can change over time, it decorates these with a specific `var` modifier. Electrum captures dynamic behavior through an *implicit* notion of time, associated with trace states, where `var` expressions may receive different values.

```
open util/ordering[State]
...

sig State {
    near, far: set Object
}

fact { first.near = Object && no first.far }

fact { all s: State, s'' : s.next | {
  Farmer in s.near =>
              crossRiver[s.near, s''.near, s.far, s''.far]
         else crossRiver[s.far, s''.far, s.near, s''.near]
  }
}

pred solvePuzzle {
    last.far = Object
}

run solvePuzzle for 8 State
```

Fig. 2. Alloy specification of execution traces for the river crossing puzzle.

In order to define how a model may evolve over time, users can write formulas and enforce them in facts, using linear-time temporal logic (LTL) (all the typical operators are supported), and resorting to the use of variables and "primed variables", to refer to values of fields or signatures before and after executing a transition, respectively.

The analysis of Electrum specifications is enabled by encoding these specifications in a number of different verification tools, including Alloy and nuXmv [12]. As described in [9], [44], the encoding in Alloy essentially uses the "local state" idiom [15], a generalization of the approach to capturing dynamic behavior within Alloy introduced in the previous section [38]. The "local state" idiom introduces an additional `Time` (or `State`) signature, and a "time" column in relations that change over time, making the values of these relations relative to a time instant. The time signature is constrained to form a total order, leading to traces of time instants, as in the `State` signature in Fig. 2. Since the time signature is used to represent time instants, its scope represents the maximum trace length that is considered for analysis. It is important to remark that, since Electrum supports LTL formulas, including in particular liveness properties, its trace model is adapted to capture *lasso traces* [50]. A lasso trace is a finite state sequence that represents an infinite run, via a loop from the final state to some previous state in the sequence. Although lasso traces do not capture *all* possible infinite state sequences, it is known that invalid LTL formulas necessarily have lasso-shaped counterexamples [50]. In fact, Electrum implements a "complete" LTL bounded model checker (i.e., given a bound n, Electrum will verify an LTL formula α over an Alloy model if and only if no counterexample of size at most n exists for α). The bound considered for the implicit time signature in an Electrum model is expressed using the `steps` keyword, as we describe below.

[1]The `ordering` model defines total orders over a given (finite) domain. Among other ingredients, it defines: `first` as a function that returns the first element in a total order; `last` as a function that returns the last element in a total order (over a finite domain); and `next`, that given an element of the domain, returns its next element in the order.

```
abstract sig Object {
    eats: set Object
}

one sig Farmer, Fox, Chicken, Grain extends Object { }

fact eating { eats = Fox->Chicken + Chicken->Grain }

var sig near, far in Object { }

fact initialState {
    near = Object && no far
}

pred crossRiver[from, from2, to, to2: set Object] {
   (from2 = from - Farmer - from2.eats and
    to2 = to + Farmer)
   or
   (one x : from - Farmer | {
       from2 = from - Farmer - x - from2.eats
       to2 = to + Farmer + x })
}

fact stateTransition {
 always {
  Farmer in near =>  crossRiver[near,near',far,far']
                else crossRiver[far,far',near,near']
 }
}

pred solvePuzzle {
    eventually { far = Object }
}

run solvePuzzle for 8 steps
```

Fig. 3. Electrum specification for the river crossing puzzle.

As an example of the Electrum approach, consider the river crossing puzzle model, this time specified using Electrum, as shown in Fig. 3. Notice how, in this model, signatures `near` and `far` are defined as mutable (through `var`), avoiding the use of an explicit additional signature `State`. Recall that according to the local state idiom, this implies the introduction, in the Alloy model generated from the Electrum model, of a `Time` signature and a time column for relations `near` and `far` (i.e., instead of being unary relations, these become binary relations from `Time` to `Object`). Also, a fact describes state transitions of this system using the linear-time temporal operator `always` (forcing what is essentially the `crossRiver` predicate in the original Alloy specification, to hold in all consecutive states in execution traces). Finally, the solutions to the puzzle are specified using the temporal logic operator `eventually`, characterizing traces where at some future instant in time all objects are on the far side of the river. Notice how, in the analysis command (`run solvePuzzle for 8 steps`), we indicate the lasso trace length to consider for analysis via the scope for `steps`.

B. DynAlloy

The DynAlloy language, originally introduced in [22] and further developed in [21], [23], [24], extends Alloy's syntax with a particular idiom for dynamic behavior, inspired by dynamic logic [31]. DynAlloy borrows from dynamic logic its abstract programming constructions, incorporating *atomic actions* and program composition operators such as nondeterministic choice, sequential composition, and unbounded iteration, to define dynamic behavior as sequences of states (program *runs*) over Alloy models. As opposed to the modal nature of dynamic logic [31] (which features diamond/box modalities for program terms), properties associated with the execution of programs are specified in DynAlloy just using *partial correctness assertions*, defined with pre and postconditions written in relational logic.

```
abstract sig Object {
    eats: set Object
}

one sig Farmer, Fox, Chicken, Grain extends Object { }

fact eating { eats = Fox->Chicken + Chicken->Grain }

act crossRiver[from, to: set Object] {
    pre { Farmer in from }
    post { one x: from |
        from' = from - x - Farmer - from'.eats &&
        to' = to + x + Farmer }
}

program solvePuzzle[near: set Object, far: set Object] {
    assume (Object in near && no far);
    (crossRiver[near, far] + crossRiver[far, near])*;
    [ Object in far ]?
}

run solvePuzzle lurs 7
```

Fig. 4. DynAlloy specification for the river crossing puzzle.

DynAlloy captures the parts of a system model that are state changing (i.e., mutable), through *parameters*. Indeed, both atomic actions and more complex program constructions define the program state they operate on, via the corresponding program's parameters. Similar to Electrum (and contrary to standard Alloy), primed parameters are not a convention, they have an actual semantics associated with state transformation: a primed variable refers to the value of a variable in the "post" state, i.e., the state after the execution of the action or program (primed variables can only appear in assertions, not within programs).

As an example of a DynAlloy model, consider the river crossing puzzle in Fig. 4. The atomic state change captured in Alloy via a predicate `crossRiver` is now captured using an *atomic action*; notice how the `crossRiver` atomic action (defined with the `act` keyword) is defined via pre and postconditions. It is worth observing how primed expressions only participate in postconditions, to refer to "post states". The `crossRiver` atomic action is then employed in a program `solvePuzzle` (defined with the `program` keyword) that sequentially composes three main parts: an assumption characterizing the initial state; an iteration (Kleene star *, denotes the iteration of the corresponding program expression zero or more times) of the non-deterministic choice (denoted by +) between crossing the river from the near to the far side and vice versa; and a "test" action (denoted by `[ ]?`) that allows the execution to continue only if all objects are on the far side of the river.

As opposed to Alloy and Electrum, the analysis of DynAlloy specifications is based on verifying a program against its partial correctness assertion, by computing a bounded version of weakest liberal precondition [18] for the program and its postcondition [21], [52]. Besides the scope for signatures in a model, DynAlloy also requires a bound for iteration. This is indicated in assertion checking commands, via the *lurs* (for loop unrolls) keyword (notice how 7 iterations, i.e., 8 states including the initial one, are sufficient to solve the puzzle).

IV. Comparing the Dynamic Behavior Notations

We now start with a qualitative analysis of the two introduced alternatives, Electrum and DynAlloy, to capture dynamic behavior over Alloy specifications. The comparison is performed around various dimensions, namely, language style in relation to the specification of dynamism (Section IV-C), expressiveness and analysis (Section IV-D), and tool support usability (Section IV-E). The motivation here is to provide a comparison that may be informative to the modeler, and may serve as a guideline towards the most convenient notation for a specific modeling and analysis task.

To better compare the two different approaches, we will consider a more complex case study, namely an Alloy model [64] of the Chord ring-maintenance protocol [57]. Although in [64] Chord is modeled in standard Alloy, it is a model that inherently requires dynamic behavior specification, and thus it is an excellent case to assess how conveniently our compared dynamic extensions can improve specification. Moreover, as we discuss below, some limitations of Alloy in relation to the analysis of dynamic properties required more involved modeling, as well as the use of complementary analysis techniques. These are also interesting to discuss in the context of the dynamic extensions studied in this paper.

Regarding Chord, it is a protocol that allows peer to peer communication via efficient lookup, by maintaining a ring-shaped structure in which each peer has a predecessor and a list of successors. The ring topology information is distributed across the peers (members of the ring); these then need to periodically update their information, to reflect changes in the ring. A non-member node *joins* the ring by contacting an existing member, and building its successor list from the contacted node and its own list of successors. This change spreads across the ring via *stabilization* operations in member nodes (that stabilize local member information) followed by *notifications* to their corresponding successors. Of course, some member nodes may *fail*, losing connection to the network and forcing the remaining members to reconfigure the ring via modifications to their own local information. This is achieved in the protocol via *reconcile*, *update*, and *flush* operations, which are executed periodically by the members of the ring.

A. Zave's Models of Chord

Chord is a very practical protocol, whose simplicity and performance made it the default choice for lookup implementation in peer-to-peer networks. Chord was the subject of study by P. Zave, who modeled the protocol and found subtle flaws in it, contradicting previous claims on the protocol being provable correct [64]. Alloy was a key modeling and analysis tool both in Zave's study of the protocol, and her subsequent proposal for fixes to the protocol [66]. More precisely, Zave's study involved a formalization of the Chord protocol and its intended properties in Alloy, and the analysis of these properties using Alloy Analyzer. In this section, we will focus on Zave's *corrected* version of Chord, specified both in Alloy and Promela (Spin's specification language) [65], [66]. Both specifications are intrinsically *dynamic*. In particular, the Alloy specification captures the protocol operations (fail, join, reconcile, etc.) as *events* associated with network nodes, using the "time" idiom. It is important to remark that the term *event* here is taken from the terminology used by P. Zave in [64] to refer to the network operations. Events may interleave arbitrarily, but the overall analysis of the protocol's main safety property (the network is always in a "valid" state, meaning that there is exactly one ring connecting the ring members, and all the appendages are linked to the ring too) is greatly simplified by the provision of a stronger *inductive* safety property [65]. This inductive invariant allows one to verify the "validity" property by verifying that all protocol events preserve it. Thus, despite the need for dynamism to reason about Chord executions, analysis is limited to traces of size 2 (for pre and post states of each event, whose invariant preservation is checked independently).

Chord also has an important liveness property: if at some point no new node joins the network nor any member node fails, then eventually the network reaches an *ideal* state (meaning that all joined nodes are members of the ring, and the node predecessor/successor pointers are consistent). This property is *not* analyzed in Alloy, but in Spin [34]. Spin's Chord model is truly dynamic, and is used to verify the invariance of the "validity" property (in its original, non-inductive, form) as well as the previously mentioned "eventually ideal" liveness property [65].

B. Our Target Chord Models

We will consider two different models of Chord, both originating from Zave's work, to be captured in Electrum and DynAlloy. Firstly, we will take Zave's elaborate Alloy model from [65], which verifies the inductive invariance of the validity property with respect to Chord's events; the model requires state change, but this is moderately exploited for analysis (with all analysis commands requiring traces of size 2). For easier reference, we will call this model the "inductive model". Secondly, we will take the essential parts of Zave's Promela model from [65], in which the events of each node are nondeterministically chosen, iterated and interleaved to form the system traces, and to verify a non-inductive invariant and a liveness property. We will call this model the "non-inductive model". In each language (DynAlloy and Electrum), the inductive and non-inductive models will share the same core part, where the state of the system is captured, and the individual events are specified. What will change from the inductive case to the non-inductive case is how the system traces are built; in the inductive case, the "system" will be individual events/actions. In the non-inductive case, the system will be, essentially, an infinite iteration of the non-deterministic choice of different actions/events.

C. Language Style Comparison

1) The System State: In Zave's model of Chord, the network consists of a set of nodes (for simplicity, we will not discuss the size-related `base` information). The relevant topological data associated with a node are its pointers to successor nodes (first successor and second successor, as for simplicity, the successor

```
one sig Network { base: set Node } { # base = 3 }

abstract sig Node {
   var succ: lone Node,
   var succ2: lone Node,
   var prdc: lone Node,
   var bestSucc: lone Node,
   ...
}{ -- Signature facts
   -- Best successor definition
   always (
      (Member[succ] && Member[succ2] => bestSucc = succ)
       ...
   )
}
```

Fig. 5. Electrum specification of the Chord network and node state (partial).

```
one sig Network { base: set Node } { # base = 3 }

sig Node { }

program chord[succ: Node -> lone Node,
              succ2: Node -> lone Node,
              prdc: Node -> lone Node
              bestSucc: Node -> lone Node] {
   ...
}
```

Fig. 6. DynAlloy specification for Chord (partial).

list is assumed to be of size at most two), the pointer to its predecessor, and a pointer to the best of the successors (all of these can be "empty" pointers). Since this node information can change during execution, all the corresponding node fields have a "time" component in Alloy, following the time idiom. This is greatly simplified in Electrum, where the mutable portions of the state are straightforwardly declared using the var modifier, as shown in Fig. 5. It is worth remarking here that the best successor is mutable, as it is defined in terms of conditions on the current first and second successors; a *fact* allows one to capture the immediate "update" of the best successor according to how the first and second successors change over time, as shown in the figure.

DynAlloy's definition of the network is the same as for Alloy and Electrum, as this part of the state is unmutable. The state changing part of a system, on the other hand, is given in DynAlloy through the *parameters* of a program. Fig. 6 illustrates this style. The `Node` signature has now no fields (since all its original fields are mutable), and the *program* that represents the system state change declares as parameters the mutable parts. Notice how, in this case, the explicit type of relations `succ`, `succ2`, `prdc` and `bestSucc` is given, reducing readability compared to Electrum. It is also worth remarking that constraints as the one defining the "dynamic" relation `bestSucc` in terms of other dynamic relations, cannot be straightforwardly achieved in DynAlloy; instead, relation `bestSucc` will have to be explicitly updated, whenever one of the relations it depends on changes (we will illustrate this issue later on in the paper).

2) System Events/Actions: Let us now move on to how the basic state changing events, or actions, are defined. Zave's characterization of the Chord protocol defines signatures associated with the protocol's *events* (with signature fields representing the parameters of the event), and facts that indicate how two system states are related, when a given event is the cause of the change.

```
abstract sig RingEvent extends Event { node: Node }
sig Join extends RingEvent { newSucc: Node }
...
fact JoinEvent {
   all j: Join, n: j.node, t: j.pre | {
-- preconditions
      NonMember[n,t]
      Member[j.newSucc,t]
      no b: Network.base | Between[n,b,j.newSucc]
-- postconditions
      n.succ.(j.post) = j.newSucc
      n.succ2.(j.post) = (j.newSucc).succ.t
}  }  -- A Join event always follows a JoinLookup event at
      -- the same node, where
      -- Join.newSucc = JoinLookup.lookup. In the pre-state
      -- of the Join, the joining node makes one query to get
      -- the successor list of Join.newSucc.
}
```

Fig. 7. Alloy specification for Chord's join event (partial).

```
pred JoinEvent[n: Node, newSucc: Node]{
    -- preconditions
    NonMember[n]
    Member[newSucc]
    no b: Network.base | Between[n,b,newSucc]
    -- postconditions
    n.succ' = newSucc
    n.succ2' = newSucc.succ
    n.prdc' = n.prdc -- "local" frame condition
    (all m: Node - n | m.succ' = m.succ &&
                m.succ2' = m.succ2 && m.prdc' = m.prdc)
    -- "global" frame condition
}
```

Fig. 8. Join event in Electrum.

In Electrum, this can be more conveniently captured using predicates (which will also better serve to compose events), in a more declarative way, without the explicit use of `Time`. To better contrast the gain in convenience, consider the Alloy specification of `JoinEvent`, taken verbatim from [65], shown in Fig. 7. The specification of this same event in Electrum, now using predicates, is shown in Fig. 8. Notice how the need for explicit time is eliminated; this simplifies the way one refers to the pre and post states, which in Electrum are referred to via the "unprimed" and "primed" expressions involving mutable fields.

It is also important to notice that specifications involving system state and state-changing actions often require so called *frame conditions*. A frame condition for a system action a is simply a constraint that indicates that the part of the state that does not concern a, will not change when a takes place. Zave's model of Chord describes the frame conditions centered on each part of the mutable state (i.e., for each mutable relation, it indicates via a corresponding fact which actions of the system may change it, or similarly, which actions will not alter the corresponding field). Additionally, since Chord events model node actions, and the mutable relations capture state for the whole system, it is also necessary to indicate a finer grained kind of frame condition: only the state that refers to the involved node changes, but not the rest. These "local" constraints are, in Zave's model, within each event definition. In our Electrum model, both coarser-grained ("global") and finer-grained ("local") frame conditions are defined within each event predicate, as shown at the bottom of Fig. 8.

DynAlloy is, in this respect, slightly different. Basic state changing events can be specified via DynAlloy's *atomic actions*; their relevant mutable state is explicitly mentioned as

```
act JoinEvent[succ: Node -> lone Node,
              succ2: Node -> lone Node,
              prdc: Node -> lone Node,
              n: Node,
              newSucc: Node]
{
pre {
      NonMember[n, succ, succ2, prdc] and
      Member[newSucc, succ] and
      no b: Network.base | Between[n, b, newSucc]
    }
post {
      succ' =  succ ++ (n -> newSucc) and
      succ2' = succ2 ++ (n -> newSucc.succ)
     }
}
```

Fig. 9. Join event via an atomic action in DynAlloy.

the parameters of the corresponding atomic actions, and the changes to the state are specified in the postcondition. The denotational semantics of DynAlloy's actions/programs (as manifested in the weakest precondition computation that DynAlloy Analyzer performs) establishes that the mutable state that is not referred to as part of the post-state (primed expressions in the post-condition) or is not part of the parameters of the action/program, does not change when the corresponding action/program is executed. This characteristic of DynAlloy makes it simpler to define actions in relation to frame conditions, as these conditions do not need to be explicitly stated. This can be seen in the DynAlloy specification of `JoinEvent` shown in Fig. 9 (see in particular the action's postcondition, in contrast with the `JoinEvent` predicate in Electrum). At the same time, as it can be seen in this figure, the explicit mention of the mutable state as action/program parameters, as well as having to refer to mutable state as relational expressions (as opposed to indirectly using mutable fields as is done in Electrum), make DynAlloy actions more verbose. In particular, notice how state updates are expressed in DynAlloy as updates on whole relations, since updates on "projected" parts of the relation would be, in accordance with Alloy's relational semantics, weaker constraints. For instance, expression `n.succ' = newSucc` as the postcondition of `JoinEvent` (as opposed to `succ' = succ + (n -> newSucc)`) would leave relation `succ'` unconstrained for all nodes except `n`.

3) Dynamic Property Specification and Checking: We now turn our attention on how the two languages being compared specify dynamic properties, and how these are checked in the languages. We will see that the languages also have different styles in this respect. As mentioned previously, we will consider two kinds of dynamic properties, for the "inductive" and "non-inductive" models, respectively.

Let us start with dynamic property specification and verification for the *inductive* model. If a property to be verified as an invariant is assumed to be inductive, then the verification is simpler, as one can check it by proving that the initial state of the system satisfies it, and all system events preserve it. As mentioned earlier, this is the case of the main invariant property of Chord, in Zave's specification [65]. This "validity" property states that there is exactly one ring connecting the ring members, and all the appendages are linked to the ring. Assuming that the validity property is captured using an Alloy predicate `Valid` (details on how this is achieved can be found in the accompanying site [1]), one can specify, and check, in Electrum the preservation of validity by the `Join` event, as follows:

```
assert JoinPreservesValidity {
(Valid[] && (some n, newSucc: Node | JoinEvent[n,newSucc])
                                        => after Valid[])
}

check JoinPreservesValidity for 8 but exactly 2 steps
```

The assertion is simple: if validity holds and `JoinEvent` occurs (at the initial state), then validity also holds in the next state. This assertion uses the linear-time temporal logic "next" operator (`after`). Since only two states are necessary (first and second trace states), the verification command can use a larger scope for the remaining signatures (in particular, the number of nodes), but uses exactly two steps, as shown above.

In the case of DynAlloy, assertions on program (including atomic actions) executions are captured using standard partial correctness assertions. The assertion here will capture the fact that the atomic action `JoinEvent` preserves validity, as follows:

```
assertCorrectness JoinPreservesValidity[
        succ: Node -> lone Node,
        succ2: Node -> lone Node,
        prdc: Node -> lone Node,
        bestSucc : Node -> lone Node,
        n: Node,
        newSucc: Node]
{
    pre { Valid[succ, succ2, prdc, bestSucc] }
    program
    {
        JoinEvent[succ, succ2, prdc, n, newSucc];
        bestSuccUpdate[succ, succ2, prdc, bestSucc]
    }
    post { Valid[succ', succ2', prdc', bestSucc'] }
}

check JoinPreservesValidity for 8
```

The notation is rather straightforward. The same issues with state-dependent parameters apply to program correctness assertions. An issue to notice here, that has been mentioned earlier on, is that we cannot use a fact to define a changing state-dependent relation, as is done with Alloy featuring explicit time, and in Electrum, with the `bestSucc` node information. Instead, we need to define an action that "updates" the state of `bestSucc` whenever any of the relations it depends on changes (in this case, relations `succ`, `succ2` and `prdc`). This is achieved via an additional atomic action, `bestSuccUpdate`, that updates the contents of `bestSucc`, and must be explicitly called after all actions affecting the depended-on fields. Finally, the check commands does not need to express a limit for `loop unrolls`, as the program of the partial correctness assertion has no loops.

In the above case, the inductiveness of `Valid` is obtained by introducing an additional, important assumption: at every program state, every node either has a successor, or does not have any successor nor a predecessor (these are called `Member` and `NonMember` properties of a node, respectively, in Zave's model [65]). This can be either added to the definition of `Valid`, or forced through a fact, for analysis.

```
pred runNode[n: Node] {
  ((n.pc = Ready) and (FailEvent[n]) and n.pc' = Ready) ||
  (n.pc = Ready and (JoinLookupEvent[n]) and
     n.pc' = Joining) ||
  (n.pc = Joining and JoinEvent[n] and n.pc' = Ready) ||
  (n.pc = Ready and StabilizeFromNewSuccessorEvent[n] and
       (after StabilizeFromOldSuccessorEvent[n] and
            n.pc' = Rectifying)) ||
  (n.pc = Rectifying and RectifyEvent[n.succ, n] and
       n.pc' = Ready) ||
  (n.pc = Ready and StabilizeFromOldSuccessorEvent[n] and
       n.pc' = Ready)
}

fact main{
  always (some n: Node | runNode[n])
}
```

Fig. 10. Chord system of nodes in Electrum.

Now let us move to the non-inductive model for Chord. The construction of strengthened inductive assertions from non-inductive ones for verification is a valuable approach, but also a very difficult one in many cases. Thus, in many contexts, one often resorts to verifying a given invariant by checking that it holds in all reachable states of the system. this is, in fact, the approach of model checking, that allows one to deal with property verification fully automatically. It is also worth mentioning that liveness properties may be dealt with using inductive reasoning too [46], although typically not in a fully automated way (e.g., variant functions need to be crafted), making the approach also difficult to use for liveness properties (cf. Section 4 in [65] for comments on this issue).

Zave's analysis of Chord, as presented in [65], limits the use of Alloy for the inductive model. The non-inductive model is captured in Promela, the specification language of Spin [34]. We take this model and reproduce it in Electrum and DynAlloy, which contrary to Alloy, provide the syntax to more conveniently capture this non-inductive model.

The non-inductive model of Chord models the system as a collection of *nodes*, that run concurrently, and each of which can perform certain actions, that correspond to the system events (fail, join, reconcile, etc.). However, for each node, these actions have certain dependencies. For instance, before performing a `Join` action, the node needs to perform a `JoinLookup`, which will define the ring node the joining node will append to; these two actions are non-atomic, in the sense that they may interleave with other node actions, and thus are modeled as two separate, but related, events. Similarly, the update of node information is performed in two steps, first a stabilization, "non-atomically" followed by a rectification step. All these event associations need to be modeled as the behavior of the system, and a usual approach is to consider an abstract form of *program counter*, that will maintain the internal mode state of a node, and only enable the corresponding events in each mode. Fig. 10 shows how this is achieved in Electrum. A mutable field `pc` is added to a node, whose value is used as part of the guard for events, and is appropriately updated when an event is triggered, to move the node to its corresponding target mode (using the primed expression notation). For instance, a node needs to be "ready" to perform a join lookup, and after it, it moves to "joining" mode; this in turn enables the "join" event, after which the node returns to a "ready" state.

As shown in Fig. 10, the whole system is constructed using the *always* temporal operator: the system always progresses by performing a step (triggering an event) in one of the nodes of the system. This is enforced with a fact, as it *defines* the behavior of the system. The system prevents two nodes to perform a step at the same time (i.e., to correctly interleave node events) thanks to the frame constraints that are explicitly provided within each node event definition, as described earlier.

The verification of the validity property is, for this system, simply specified as follows:

```
assert validIsInvariant {
    always Valid[]
}
```

We also mentioned before that Chord has an important liveness property associated: if at some point nodes stop failing and no new nodes join the network, then the network will eventually reach a continuous "ideal" situation. To model this property, we need to slightly modify the system, adding a "flag" (in the Spin model, this is called "churnStopped"), that can be arbitrarily enabled, and once enabled, prevents joining and failing from happening:

```
one sig churnStopped {
    var flag: Boolean
}

pred runNode[n: Node]{
  ((n.pc = Ready) and (churnStopped.flag = False) and
   (FailEvent[n]) and n.pc' = Ready) ||
 ...
}

pred setChurnStopped[] {
    churnStopped.flag' = True
}

fact main{
  (churnStopped.flag = False) and
  always ((some n: Node | runNode[n]) || setChurnStopped[])
}
```

Of course, now `churnStopped.flag` needs to be added to the frame constraints of all node events, as it becomes part of the mutable system state. The liveness property can now be captured as follows:

```
assert liveness{
  (eventually churnStopped.flag = True)
       => eventually (always Ideal[])
}

check liveness for 6 but exactly 10 steps
```

The non-inductive model can be captured in DynAlloy using program constructions, in particular test actions (for guards), sequential composition, non-deterministic choice, and iteration. The main difference with the Electrum model is that it has a more programmatic style. Fig. 11 shows how the non-inductive Chord model is captured in DynAlloy. Again, a node progresses by executing one of its events, appropriately guarded, and correspondingly updating the node's mode; sequential composition takes care of the "chaining" of actions for each event. Finally, the system is an unbounded iteration (Kleene star `*`) of nondeterministically choosing a node, and performing an event in it.

The non-inductive verification of the invariance of `Valid` is captured as follows:

```
program runNode[succ: Node -> lone Node,
                succ2: Node -> lone Node,
                prdc: Node -> lone Node,
                n: Node]
                var [succNode: Node]
{
    ([n.pc = Ready]?;
     call JoinLookupEvent[succ, succ2, prdc, n];
     n.pc := Joining
    ) +
    ([n.pc = Joining]?;
     JoinEvent[succ, succ2, prdc, n];
     n.pc := Ready
    ) +
    ([n.pc = Ready]?;
     Fail[succ, succ2, prdc, n]
    ) +
    ([n.pc = Ready]?;
     StabilizeFromNewSuccEvent[succ, succ2, prdc, n];
     call StabilizeFromOldSuccEvent[succ, succ2, prdc, n];
     n.pc := Rectifying
    ) +
    ([n.pc = Rectifying]?;
     succNode := n.succ ;
     RectifyEvent[succ, succ2, prdc, succNode, n];
     n.pc := Ready
    ) +
    ([n.pc = Ready]?;
     call StabilizeFromOldSuccEvent[succ, succ2, prdc, n]
    )
}

program main[succ: Node -> lone Node,
             succ2: Node -> lone Node,
             prdc: Node -> lone Node]
             var [n: Node]
{
  (choose[n] ; runNode[succ, succ2, prdc, n])*
}
```

Fig. 11. Composition of system specification in DynAlloy.

```
assertCorrectness validIsInvariant[succ: Node -> lone Node,
                                   succ2: Node -> lone Node,
                                   prdc: Node -> lone Node,
                                   bestSucc: Node -> lone Node]

    pre { Init[succ, succ2, prdc] }
    program
    {
        main[succ, succ2, prdc];
        bestSuccUpdate[succ, succ2, prdc, bestSucc]
    }
    post { Valid[succ', succ2', prdc', bestSucc'] }
}
```

Notice that, since the Chord events do not need to query the `bestSucc` information (only the `Valid` needs this information), it suffices to update the `bestSucc` mutable state just once, before checking the postcondition. Another important difference with Electrum arises at this point: DynAlloy's partial correctness assertions can only express safety properties. These are insufficiently expressive to capture liveness properties.

D. Expressiveness and Model Analysis Comparison

Both Electrum and DynAlloy have been motivated by an inconvenience in the direct use of Alloy to express properties of executions, rather than on a concrete expressiveness limitation. In fact, from a theoretical point of view, neither Electrum nor DynAlloy are more expressive than Alloy as a logical language: first-order relational logic with closure operators is sufficiently expressive to encode both linear-time temporal logic, and dynamic logic [25], [31]. The expressive power is in fact enhanced by the extensions, only in relation to particular analysis approaches, and this is what we discuss below. We refer to analysis from a qualitative point of view (quantitative analysis is performed later on in this paper), and limit ourselves to capturing properties of executions through the mechanisms proposed by each extension.

In the case of DynAlloy, unbounded finite iteration (Kleene star) leads to state sequences of arbitrary length, that cannot be directly captured using Alloy's finite sequences. This expressiveness enhancement provided by DynAlloy cannot however be exploited when the mechanism for DynAlloy analysis is SAT solving, which in fact makes all iterations and execution traces bounded. It is worth remarking that, using alternative analysis approaches, in particular deductive proof systems, it is possible to reason about DynAlloy specifications with unbounded traces [23] (via a sound and complete deductive proof system); this approach has also been adapted to deal with Alloy itself [48]. However, these systems obviously sacrifice automation, i.e., deductive verification for these languages can be assisted but are not fully automated. Thus, DynAlloy Analyzer, which concentrates in automated analysis, only supports bounded analysis via SAT solving [52].

In the case of Electrum, the approach is based on extending Alloy with temporal logic operators. This approach leads to a strictly more expressive language, compared to DynAlloy, for specifying properties of executions. This is so due to the fact that DynAlloy does not employ dynamic logic formulas for specifying properties of executions, only partial correctness assertions (which correspond to a specific pattern of use of the box modality of dynamic logic [31]). In particular, Electrum is capable of capturing *liveness* properties [4], whereas DynAlloy is restricted to safety properties (through partial correctness assertions). Moreover, Electrum has been designed with the motivation of making specifications analyzable by alternative technologies. Its analysis tool supports not only SAT-based bounded model checking through Alloy, but also the use of model checkers, such as nuXmv [12], for *unbounded* analysis [9]. This makes the expressiveness enhancement, in relation to fully automated analysis, an actual improvement. It should be nevertheless noted that there is currently only partial support for the use of nuXmv in the analysis Electrum models, since the current implementation cannot yet handle Electrum's full syntax.

In conclusion, in terms of expressiveness, Electrum is a strictly more expressive language than DynAlloy, in relation to the analysis mechanisms associated with their corresponding automated analysis tools. A concrete example was given previously in this section, with the "eventually ideal" property of the Chord protocol, which can be specified in Electrum's notation but not as a DynAlloy partial correctness assertion.

E. Tool Usability

The tools associated with both extensions have been developed, in their latest versions, as extensions to Alloy Analyzer (in fact, Electrum is now being adopted as part of the official Alloy Analyzer release). Tools are therefore straightforward to be used by Alloy users, both from the point of view of the tool

itself, or by accessing their provided APIs (in both cases, these extend Alloy Analyzer's). There is some distinction in how witnessing instances (counterexamples of properties, runs of systems) can be explored, when these correspond to properties of executions. Electrum integrates the standard visualizer in Alloy Analyzer and a navigation bar, to "explore" a trace through time instants. Electrum also offers some more sophisticated "next instance" queries, that allow the user to ask for further instances associated to the one being currently explored, e.g., by asking for one with a different initial state. DynAlloy, on the other hand, offers a debugger-like visualizer, that allows users to navigate traces, highlighting the corresponding parts of the program in the model, set "watched" expressions as in IDEs, and observe intermediate values of the involved expressions. These visualization mechanisms fit well the specification styles of each of the notations, as trace visualizations are more appropriate for transition-system like specifications, and watch/debug view is appropriate for abstract sequential programs. Overall, the usability experience is very good with both extensions.

F. Summary

The two extensions share some common characteristics: they attempt to be faithful to Alloy, and provide kind of conservative extensions of the original language. That is, valid Alloy specifications are valid Electrum and DynAlloy specifications too, and the semantics is preserved for the Alloy fragments of the extensions. Both notations also try to be *declarative* in nature. Electrum chooses a higher level language for transition system specification, defined via LTL constraints, while DynAlloy uses an abstract sequential programming language approach. These two alternative styles are also present in other analysis contexts, model checking in particular. LTL is the specification language for various model checkers at design level (e.g., [33], [45]), while partial correctness assertions are used in software model checkers (e.g., [13], [14], [61]). This is an important distinction of the two languages, that may serve potential users in deciding which extension to use, depending on the properties of interest, and the level of abstraction at which the transition system is defined. More precisely, for more abstract system models, with relatively simple control flow, Electrum is in principle a better choice; for system models that have more complex control flow, the programming language constructs that DynAlloy uses, makes it a better choice. Finally, if the properties of interest are liveness properties, then Electrum is the only alternative that offers direct support for their analysis. For instance, for the formal models that we will consider for assessing efficiency in the following section, for the most algorithmic (InsertionSort, Dijkstra, Firewire), DynAlloy is the most appropriate; on the other hand, for SpanTree, RingElection and Chord, Electrum is better suited (coincidently, these three models have liveness properties, enforcing the fact that Electrum is the language to use). Simpler dynamic models, such as RiverCross, FileSystem, Hotel and CacheMemory, are equally easy to be captured in both notations.

Regarding how state is captured, the two languages employ different approaches. Electrum uses the `var` annotations, which make the identification of the state dependent parts more direct, compared to DynAlloy, which uses program parameters for this task. Electrum leads, in this respect, to more compact specifications, as the explicit mutable state as parameters in DynAlloy makes that notation more verbatim. If the model requires a complex control flow, on the other hand, having to manually define a sort of program counter, to enforce the control flow, would make Electrum more cumbersome. DynAlloy also has frame constraints built-in in the semantics of programs, whereas these constraints need to be manually specified in the case of Electrum. It is worth remarking that, especially with the Chord model, the need for concurrency constructs becomes apparent; none of the notations directly support it, although both have the expressive power to allow for it, as it was shown in the models in this section. Finally, in relation to tool support, both notations are very good, and have features that match the characteristics of the notation. We found DynAlloy Analyzer's instance exploration more convenient for debugging models, solely because it is easier to match which event (when multiple events are enabled) are the ones involved in specific transitions. This of course can be achieved in Electrum too, at the cost of adding some history information as part of the "mode" each node is in, implemented as a sort of program counter in our running case study.

TABLE I
CHARACTERISTICS OF THE EXTENSIONS

	Electrum	DynAlloy
Expressiveness	Equivalent to Alloy's	Equivalent to Alloy's
Trace analysis	Bounded/Unbounded	Bounded
Syntax	Declarative (LTL)	Imperative (abstract programs)
Instance visualization	Trace visual. style	Program debugging style

The main characteristics of the extensions, without considering efficiency, are summarized in Table I.

V. PERFORMANCE EVALUATION

This section concentrates in empirically evaluating DynAlloy and Electrum, as well as Alloy, from the point of view of the efficiency and effectiveness of their corresponding analysis tools. This empirical evaluation is driven by a set of well-known Alloy specifications demanding dynamic behavior analysis, taken from the literature. Each case study is accompanied by a brief description of the model and the intended properties to be verified (except when explicitly stated, models are correct and thus the properties should not lead to counterexamples). The experimental analyses are detailed in Tables II and III, which compare the performance of the tools. More precisely, the tables describe:

- The property being analyzed.
- The scope used for analysis (each property is checked for increasingly larger scopes). This is separated between the trace length (which is captured differently by the different tools, loop unrolls (`lurs`) in DynAlloy, `steps` in Electrum, scope for sequences in Alloy[2]), and the scope for the

[2]In Electrum and Alloy, the `steps` and Time's scope refer to the maximum number of different trace states to be considered in the analysis. In DynAlloy, on the other hand, `lurs` refer to the maximum number of times that *iterations* (`*`) will be unrolled.

TABLE II
EXPERIMENTAL RESULTS PERFORMANCE TABLE

Model scopes	Traces lurs/steps	Verification Time(ms) Alloy	DynAlloy	Electrum	Speed up *Exts. (Alloy)*
River Cross					
property: *No Quantum Objects*, model scopes = fixed (number of objects)					
4	7	**4**	12	8	1x (-2x)
4	30	**43**	96	53	2x (-1x)
4	50	158	160	**121**	1x (1x)
4	100	492	**480**	484	1x (1x)
property: *Farmer Always There*, model scopes = fixed (number of objects)					
4	7	0	**0**	1	3x (2x)
4	30	3	**1**	7	6x (3x)
4	50	13	**1**	19	14x (10x)
4	100	40	**3**	74	22x (12x)
property: *No Resurrection*, model scopes = fixed (number of objects)					
4	7	4	**0.8**	5.1	6x (6x)
4	30	47	**2**	18	7x (18x)
4	50	83	**4**	42	10x (20x)
4	100	343	**10**	173	17x (34x)
Chord protocol					
property: *Safety*, model scopes = #Nodes					
6	10	-	**901**	2608	3x
6	30	-	**8802**	25491	3x
6	50	-	**28783**	77592	3x
SpanTree					
property: *Good Safety*, model scopes = #Processes					
5	10	40	**2**	670	261x (15x)
5	50	323	**13**	36648	2804x (25x)
8	5	69	**2**	534	271x (35x)
8	10	8459	**3**	219763	70891x(2729x)
8	50	110226	**14**	TO	- (7655x)
10	5	147	**4**	2251	567x (37x)
10	10	803128	**6**	TO	- (139190x)
10	20	2179296	**9**	TO	- (253407x)
10	30	TO	**13**	TO	- (-)
10	50	TO	**18**	TO	- (-)
Hotel					
property: *Good Safety*, model scopes = #Guests/#Rooms					
2/2	5	**3**	20	7	3x (-2x)
2/2	30	**227**	360	295	1x (-1x)
3/3	5	**13**	166	58	3x (-5x)
3/3	30	**4920**	19682	7599	3x (-2x)
4/4	5	**21**	469	330	1x (-16x)
4/4	10	**1552**	12677	5974	2x (-4x)
4/4	30	**128206**	2621684	368751	7x (-3x)
property: *Bad Safety*, model scopes = #Guests/#Rooms					
3/3	5	**5**	20	21	1x (-4x)
3/3	8	**34**	54	49	1x (-1x)
5/5	5	29	**16**	123	8x (2x)
5/5	8	**35**	53	160	3x (-2x)
10/10	5	87	**40**	931	23x (2x)
10/10	8	**22**	93	1467	16x (-4x)
Ring Election					
property: *Safety*, model scopes = #Process					
3	10	**2.8**	51.4	21	2x (-8x)
3	50	**7**	1233	232	5x (-31x)
3	100	**14**	5513	1322	4x (-92x)
5	10	**84**	220	203	1x (-2x)
5	50	**1050**	22938	3402	7x (-3x)
5	100	**3559**	152359	20302	8x (-6x)
Dijkstra					
property: *Prevents Deadlocks*, model scopes = #Processes/#Mutexes					
3/2	5	**5**	42	8	5x (-2x)
3/2	25	**954**	1007	1206	1x (-1x)
3/2	50	**3086**	3215	3479	1x (-1x)
4/3	5	**12**	109	37	3x (-3x)
4/3	25	12284	**5439**	12391	2x (2x)
4/3	50	32972	**21603**	35433	2x (2x)
5/4	5	**56**	272	74	4x (-1x)
5/4	25	206935	**56753**	474028	8x (4x)
5/4	50	522356	**277046**	484716	2x (2x)

TABLE III
EXPERIMENTAL RESULTS PERFORMANCE TABLE (CONTINUED)

Model scopes	Traces lurs/steps	Verification Time(ms) Alloy	DynAlloy	Electrum	Speed up *Exts. (Alloy)*
Firewire					
property: *At Most One Elected*, model scopes = #Nodes					
4	5	31	**2**	38	14x (12x)
4	25	31488	**5**	885	184x (6560x)
5	5	75	**2**	441	201x (34x)
6	5	429	**3**	426	141x (141x)
6	15	1740026	**7**	3921	521x (231079x)
6	25	TO	**10**	12021	1248x (-)
property: *One Eventually Elected*, model scopes = #Nodes					
4	5	18	**9**	29	3x (2x)
4	25	141	**17**	651	39x (8x)
5	5	**34**	63	138	2x (-2x)
5	25	275	**29**	5376	184x (9x)
6	5	**19**	104	162	2x (-5x)
6	25	424	**37**	6912	187x (11x)
property: *No Overflow*, model scopes = #Nodes					
4	10	441	**3**	222	61x (121x)
4	25	3614	**6**	1532	242x (571x)
5	10	551	**4**	1188	270x (125x)
5	25	7291	**10**	7364	708x (701x)
6	10	2432	**8**	1824	229x (305x)
6	25	27686	**26**	28160	1093x (1074x)
Cache memory					
property: *DirtyInv*, model scopes = memory and data size					
4	10	**27**	74	1133	15x (-3x)
4	25	**253**	583	7681	13x (-2x)
5	10	**32**	136	80471	592x (-4x)
5	25	**372**	1167	1439226	1232x (-3x)
6	10	**80**	248	TO	- (-3x)
6	25	**873**	2993	TO	- (-3x)
property: *FreshDir*, model scopes = memory and data size					
4	5	21	**13**	15	1x (2x)
4	15	40.	54	**30**	2x (1x)
5	5	123	**6**	11	2x (2x)
5	15	611	**35**	46	1x (17x)
6	5	226	**9**	16	2x (25x)
6	15	396	**59**	87	1x (7x)
property: *CacheInMain*, model scopes = memory and data size					
4	5	202	**39**	339	9x (5x)
4	20	3157	**141**	3706	26x (22x)
5	5	2261	**144**	8493	59x (16x)
5	20	342739	**463**	289202	625x (741x)
6	5	6934	**371**	102408	276x (19x)
6	10	57972	**455**	1862589	4096x (127x)
6	20	TO	**798**	TO	- (-)
File System					
property: *Read Matches Prior Write*, model scopes = #inodes and data					
4	10	1	1	**1**	1x (1x)
4	50	3	**2**	18	8x (2x)
6	10	**1**	2	2	1x (-1x)
6	50	8	**3**	22	7x (2x)
8	10	**3**	3	4	1x (-1x)
8	30	8	**3**	13	4x (2x)
8	50	16	**4**	32	8x (4x)
Insertion Sort					
property: *Sort Works*, model scopes = sequence size					
4	5	4	5	**2**	2x (2x)
4	25	23	**19**	22	1x (1x)
4	100	115	**81**	1165	14x (1x)
6	5	5	5	**2**	2x (2x)
6	25	28	**26**	27	1x (1x)
6	50	**64**	79	160	2x (-1x)
6	100	137	**136**	1182	9x (1x)
property: *Find Min Works*, model scopes = sequence size					
4	5	**4**	30	169	6x (-8x)
4	50	**68**	142	48058	338x (-2x)
4	100	**218**	374	623541	1666x (-2x)
6	5	**3**	26	253	10x (-8x)
6	25	**25**	50	1244	248x (-2x)
6	50	**75**	126	92077	733x (-2x)
6	100	**237**	426	458268	1077x (-2x)

rest of the model (referenced as "Model scopes" in tables). When, besides the trace length, other aspects of the scope are increased in the experiments, we indicate in the corresponding table what part of the scope is being increased and how. Since Alloy models use "ordering" on Time, and this forces the analysis to be strict on trace length, we report the results of DynAlloy and Electrum analysis commands using "exact" scopes for the trace length too.

More detailed analysis results (including performance for "non-strict" analyses on trace length), can be found in the accompanying site [1].

- The time taken by each tool, in milliseconds (we highlight in boldface the fastest of the three tools). Timeout (TO) is set at 60 minutes.
- The speed-up of the fastest extension with respect to the other extension, and with respect to Alloy. When Alloy is the fastest tool, the speed-up with respect to Alloy shows instead the slow down rate of the fastest extension, with respect to Alloy. Also, when one of the tools times out, we do not report the corresponding speed-up rates.

It is worth remarking that these tables only refer to SAT-based bounded analysis, and therefore this part of the analysis is restricted to bounded properties only. Moreover, the Alloy dynamic models consider non-looping traces (as opposed to Electrum), since for safety properties, looping traces do not provide stronger analysis results: a safety property has a k-length lasso-trace counterexample iff it has a k-length non-looping trace counterexample. All tools were configured to use MiniSAT as the underlying solver, since this was the solver that in general achieved better performance. All the experiments were run on a Intel Core i7 with 8 threads and 16GB RAM. Further details and experiments, including results regarding different versions of the tools, different underlying solvers for the three tools, and model checkers for Electrum, can be found in the experiments site [1].

River cross: The classic river crossing puzzle. In this model, we check three properties, namely:

- *No Quantum Objects*, that specifies that no object can be in two places at the same time;
- *Farmer Always There*, which states that the river crossing puzzle transitions cannot make the farmer disappear, and
- *No Resurrection*, which states that, once an object is lost, it cannot be recovered.

For this case study, the scope is fixed (4 objects, farmer, grain, fox and chicken), except for the trace length. As it can be noticed from Table II, in the first property Alloy outperforms both of the extensions, and DynAlloy outperforms Electrum. In the other two properties, DynAlloy is the most efficient tool.

Notice that we do not consider the satisfiable predicate *solvePuzzle*, because it finds a solution for a relatively small scope (8). Details about *solvePuzzle* can be found in the experiments site.

Chord: The peer-to-peer lookup protocol discussed at large in the previous section. In this model, we check two properties:

- *Safety*: states that the validity property is an invariant of the Chord system.
- *Progress*: states that if at some point no new nodes join, and no existing nodes fail, then eventually the system reaches a continuously ideal situation.

Since the second property cannot be checked using Alloy or DynAlloy, the tables will only mention *Safety*. Liveness is assessed later on in this section. The evaluation is for increasingly longer traces. As it can be seen from the Table II, DynAlloy is faster than Electrum for this property.

Span Tree: A specification of a distributed spanning tree algorithm over arbitrary network topologies. The properties of interest accompanying this model are the following:

- *Bad Liveness/Good Liveness*, that show how important it is to consider *fairness* in order to ensure that the algorithm progresses, and
- *Good Safety*, that specifies that all nodes are covered by the algorithm.

As for the Chord model, only *Good Safety* is assessed for comparison (liveness delayed to later on in this section). Notice that, for this case study, we are able to increase the "static" scope of the model, as well as the trace length. The evaluation is for increasing number of processes, and increasingly longer traces. As it can be seen from the Table II, DynAlloy outperforms, by an important margin, both Alloy and Electrum.

Hotel: This model specifies the assignment of electronic keys to guests in a hotel. Essentially, the assignment works as follows: the hotel issues a new key to the next occupant of a room, which upon first use will recode the lock, so that previous keys for that room will no longer work. In this model, the properties of interest are the following:

- *Bad Safety*, expected to fail, if it is not a requirement that every guest must enter the room immediately after check-in, and
- *Good Safety* that holds if the mentioned requirement is met.

Also, in this case study, the "static" scope of the model and trace length are increased. The evaluation is for increasing number of guests and rooms, and increasingly longer traces. As it can be seen from the Table II, the three tools reach a similar performance in the first predicate, and Alloy outperforms the other tools for most scopes in the second property.

Ring Election: The model of a well-known distributed algorithm for leader election, for processes connected in a ring topology. This model proposes the following three properties for analysis:

- *Liveness1*: it states that, if there exist processes in the system, then eventually one will become elected. This property is expected to fail, since progress is not assumed in the assertion,
- *Liveness2*: same as the previous property, but assuming progress,
- *Safety*: it specifies that once a process is elected leader, it will always remain the leader.

As for Chord and Span Tree, only the efficiency for *Safety* property was compared. In this case study, we increase the number of processes and the trace length. Experimental results are shown in Table II; Alloy shows the best performance in this model, with DynAlloy being the fastest between the two extensions.

Dijkstra: A model of Dijkstra's algorithm for mutual exclusion. The property of interest associated with this model is:

- *Dijkstra Prevents Deadlocks*: it states that if the mutex ordering criterion is satisfied, then deadlocks are prevented.

TABLE IV
EXPERIMENTAL RESULTS ON LIVENESS WITH BOUNDED TRACES

#Model scope	#steps	Verification Time (ms)		
		Ring Election	Span Tree	Chord
4	5	104	16	1209
5	6	593	245	11836
6	7	3488	3229	108350
7	8	19283	33412	1151701
8	9	173813	342392	TO
9	10	2542928	TO	TO
10	11	TO	TO	TO

TABLE V
EXPERIMENTAL RESULTS ON LIVENESS WITH UNBOUNDED TRACES

#Model Scope	Verification Time (ms)		
	Ring Election	Span Tree	Chord
3	2243	2327	-
4	28531	3492	1159
5	TO	31531	11349
6	TO	523563	30564
7	TO	TO	81966
8	TO	TO	193477
9	TO	TO	TO

The experimental results are shown in Table II. For this case study, we increase the number of processes and mutexes, as well as trace length. As it can be seen from the Table, DynAlloy outperforms the other tools for most scopes in the analysis of this property.

Firewire: A model adapted from [17], describing a leader election protocol used in Firewire, an IEEE standard for connecting consumer electronic devices. Three properties are proposed with this model:

- *One Eventually Elected*, stating that the algorithm should guarantee that a leader is eventually elected (it is expected to produce counterexamples),
- *No Overflow*, which asserts that link queues do not overflow, and
- *At Most One Elected*, asserting that at most one leader is eventually elected.

The first property is actually a liveness property. But the original Alloy model considers it, checking for reachability (it finds counterexamples though). We have then considered this property, as a reachability property, both in DynAlloy and Electrum too. The evaluation is for increasing scopes (messages, requests, nodes, links) and trace length. DynAlloy is in this case the fastest tool, and for a very large margin.

Cache Memory: A model of cache memories, with operations for writing (through cache) and flushing dirty addresses to main memory. The original "static" model is described in [38], and the dynamic version of the model is the one driving the presentation in [21]. The properties of interest in this model are the following:

- *DirtyInv*, which states that, for non-dirty addresses in the cache, the contents of the cache and main memory coincide
- *CacheInMain*, which states that, if there are no dirty addresses, then the cache is contained in main memory.
- *FreshDir*, which asserts that there is always an address that has not been written to (it is expected to produce counterexamples).

The third property is an invalid property, so counterexamples are found for it. The experimental results for this case study are shown in Table III. The evaluation is for increasingly larger memories (more addresses and data values), and increasingly longer traces. As the experiments show, in the first two properties DynAlloy and Alloy reach a similar performance; DynAlloy is faster than the other tools in the third predicate.

FileSystem: A model of a file system where an inode is either a directory node or a file node; a directory node maps names of files and directories to other inodes, and a file node contains some mutable data. The model, taken from [49], also defines operations on a file system, including navigation, writing and reading to/from a file system, considering specific locations. The property of interest associated with this model is:

- *Read Matches Prior Write*: which states that once a write operation is applied to a file system, reading operations will not change the information until the next write.

For this case study, we increase the number of objects, as well as trace length. As it can be seen from Table III, DynAlloy outperforms the other tools for most scopes in the analysis of this property.

InsertionSort: A model of the well-known Insertion Sort algorithm, taken from [49]. The properties of interest associated with this model are:

- *Sort Works*: which checks that the algorithm correctly sorts a given sequence.
- *Find Min Works*: which checks that the algorithm uses the index of the minimum element.

The experimental results are shown in Table III. For this case study, we increase the number of sequences, as well as trace length. In the first property DynAlloy outperforms the other tools and in the second property Alloy achieves a better performance.

A. Liveness Properties

As we mentioned in the previous section, Electrum can handle liveness properties in a way that the other tools cannot, by resorting to model checkers as opposed to Alloy for unbounded analysis. Moreover, the trace characterization behind Electrum also allows the tool to check liveness properties in a bounded fashion. We have mentioned a few liveness properties present in some of the models, that we did not consider for performance comparison, since Electrum is the only tool that can directly handle them with bounded traces as well as unbounded traces. In this section we first report analysis times for liveness properties over bounded lasso traces. The results are shown in Table IV, for liveness properties in the *Ring Election*, *Span Tree* and *Chord* models. We also report analysis times for liveness properties over unbounded traces, using nuXmv as the model checker underlying Electrum. The results are shown in Table V. As for the previous experiments, times in these tables are shown in milliseconds.

The Electrum language for capturing system state is significantly richer than that of other (lower level) model checkers: since it extends Alloy, it directly supports relations, and the use of relational operators including closure operators. It is then

TABLE VI
EXPERIMENTAL RESULTS ON CHORD WITH ELECTRUM AND SPIN

	Verification Time (ms)	
#Model scope	Electrum	SPIN
4	728	0
5	6567	0
6	17314	80
7	41847	200
8	92935	310
9	TO	670
10	TO	1850
14	TO	129200
15	TO	TO

to be expected that model checking Electrum models would be more expensive than model checking lower level models. To put the analysis numbers in context, without risking being unfair due to differences in modeling decisions, we compared Electrum with Spin, on Zave's Chord models (with the Alloy model being adapted to Electrum, as shown earlier in this paper). Zave's Spin model starts from a specific initial ring configuration [65], so we mimicked this behavior in Electrum too. The comparison of Spin and Electrum is shown in Table VI. In terms of efficiency, Spin is significantly more efficient; in terms of scalability, Spin scales to almost twice the scope that Electrum can handle when analyzing this property. These numbers provide a reference of the analysis efficiency that is currently sacrificed, in order to use a higher-level, richer specification language. There may be significant room for improvement too: Spin is a very mature model checker that implements many optimizations, whereas Electrum is a relatively recent project.

B. Assessment

Let us discuss the experimental results on the above case studies. Firstly, notice that the main motivation for the extensions for dynamic behavior over Alloy specifications is specification convenience, i.e., declarativeness in modeling, not necessarily efficiency improvement with respect to Alloy. Regarding this issue, in [44], it is explicitly stated that a comparison with Alloy, in terms of efficiency, is less interesting than comparing with alternative tools, since Electrum is based on Alloy (in fact, Electrum implements the "time" idiom, so increased efficiency with respect to Alloy is in principle not expected). In the case of DynAlloy, on the other hand, the results shown in early works such as [21] report that DynAlloy analyses, based on weakest precondition computation, lead to improved efficiency with respect to Alloy. This is a first interesting result of our evaluation: as we can see in Table VII, while DynAlloy is still the most efficient tool in the majority of cases, Alloy does perform very well in a good number of cases (in other words, the speedup gain by DynAlloy is relatively small for many cases). This reduces the difference with earlier evaluations [21], and can be explained by the significant advances both in the SAT solving technologies underlying Alloy, and in the language's specific optimizations (it is worth remarking that [21] is previous to the development of KodKod, the relational solver behind the current version of Alloy).

Regarding the case studies in which Alloy outperforms DynAlloy, there are various different characteristics, that prevent us from having a consistent conclusion in this regard. In some cases, River Cross for instance, it is due to the simplicity of the model and the analysis. In others, like Ring Election, most of the state space is "touched" by the state-modifying operations, thus making the weakest precondition approach less effective. DynAlloy's weakest precondition computation profits from cases where not all state-modifying operations affect the state in the same way, which allows the tool to generate less intermediate variables for property verification. Overall, DynAlloy has evolved specifically as an intermediate language for source code verification, making it, in some cases, less efficient for dynamic behavior analysis in the context of more abstract Alloy modeling.

TABLE VII
SUMMARY OF EXPERIMENTAL RESULTS FOR SAFETY PROPERTIES

Total (18 properties)				
Extensions		Alloy vs Extensions		
Electrum	DynAlloy	Alloy	Electrum	DynAlloy
3	**15**	6	0	**12**

TABLE VIII
SUMMARY OF EXPERIMENTAL RESULTS FOR LIVENESS PROPERTIES

Total (2 properties)			
Electrum (Bounded)	Electrum (Unbounded)	Alloy	DynAlloy
3	3	-	-

Among the two extensions, and as far as safety properties are concerned, DynAlloy clearly outperforms, and by a large margin, the Electrum tool. The difference in performance between Alloy and Electrum is significant too (recall that Alloy models have non-looping traces, while Electrum models inherently consider looping traces). This is, in our opinion, an indication that there is room for improvement in bounded analysis of safety properties as implemented in the Electrum tool. It is also worth pointing out that some of the mechanisms that are implemented in DynAlloy's weakest precondition computation, are not specifically tied to this language, and can also be exploited in the context of Electrum.

It is important to remark that Electrum can also verify safety properties on unbounded traces (resorting to model checkers), as opposed to the other two tools that only support analysis on bounded scenarios. We do not report here the results of Electrum's unbounded verification of safety properties because most models are either not supported by Electrum's translation into nuXmv/NuSMV, or reach a time out with the smallest scopes (in fact, only some properties of 4 out of the 9 models can be handled by Electrum with unbounded traces). The results of unbounded verification of safety properties with Electrum can however be found in the experiments site [1].

Regarding the verification of liveness properties of the considered case studies, the Electrum tool shows a feature that the other tools do not support. We summarize the liveness verification results in Table VIII. This is an important distinction of Electrum in comparison with the other tools. As far as unbounded analysis is concerned, the tool is still below the scalability of other mature model checkers. Moreover, the rich definition of the state structure offered by Alloy, and thus

supported by Electrum, makes it very difficult for the tool to analyze large state spaces in the same way that model checkers do. But performance and scalability is not the only dimension to take into account: Electrum supports a very expressive declarative language to describe system state (Alloy), that can favor model understandability and avoid introducing unnecessary operational detail into specifications. This calls for a different kind of comparative study, between Electrum and other model checkers such as Spin and NuSMV, which is beyond the scope of our paper.

VI. Threats to Validity

We have made our best effort to take into account potential threats to the validity of our evaluation. Firstly, the comparison may be biased by the case studies that we considered; we focused on well-known Alloy specifications demanding dynamic behavior analysis, from a variety of sources, and trying to cover different kinds of models, ranging from abstract behavioural models to more concrete algorithmic models. Another potential bias is in how each model is captured, as different modeling alternatives may lead, especially, to disparate performances. For each language, we took models exactly as published in the literature, when available. The Chord models were largely discussed in the paper, and their associated modeling decisions come from Zave's work [65]. Most other Alloy versions of the models come as part of the Alloy Analyzer distribution (except InsertionSort and FileSystem, which were taken from [49]); Electrum Analyzer comes with versions of all models except CacheMemory, FileSystem and InsertionSort; and DynAlloy had models for all case studies except Span Tree, Dijkstra, Firewire, FileSystem and InsertionSort. For the missing case studies, and in some cases missing properties (e.g., Farmer Always There and No Resurrection), we constructed the models ourselves, trying our best to be faithful to the corresponding notation. The constructed models were independently checked by different authors of this paper. The Chord model plays a central role in our qualitative analysis. It has a simple control flow in its dynamic aspects, favoring Electrum for its characterization. While its introduction may be considered a bias in our comparison, we believe it is a highly relevant "dynamic" model in the context of Alloy, worth studying in this paper. It is also a model that, in our opinion, is difficult to replace by one of similar importance, and fairer to both studied extensions.

Our qualitative evaluation has not exhaustively covered the syntax of the studied notations. While we have not been exhaustive, we made our best effort to include the most relevant parts of the two notations, especially in our introduction of the notations and the qualitative analysis. In particular, we did not use some temporal logic operators available with Electrum, but tried to use the most suitable for the specification of the required properties (we used `eventually`, `always`, `after` and primed expressions for "next"). Similarly, we have not covered some of DynAlloy's program composition operators, e.g., `repeat` and `while`, in the introduction of the notation and the qualitative analysis.

Another potential threat to validity is in the scopes of the evaluations, including trace lengths, in the considered SAT solvers, and the intrinsic non-determinism in solving. Although we only report a representative sample of scopes and configurations (including the selected SAT solvers) in this paper due to space reasons, further options are reported in the paper's evaluation site [1]. The reader can check that the results reported explicitly in the paper are indeed representative of the general case. To account for the non-determinism in SAT solvers, each configuration was run 30 times; the reported times are averages, as the differences between the different runs of the same experiment were negligible (see standard deviation and other statistics in the paper's evaluation site [1]). Finally, we tried to consider all the variables in the experiments, that may affect performance. In particular, all tools were run using *strict* (fixed scope) as well as *non-strict* (all scopes up to a given maximum) mode, as some of the tools, notably Alloy (which uses ordering and thus forces a strict analysis), can have a better performance in one of the two modes. All experimental results and the corresponding comparisons can also be found in [1].

VII. Related Work

Comparative studies of formal notations are useful elements, that allow one to better understand the different characteristics of alternative notations, their advantages and drawbacks, and helps users in choosing the right tool for their analysis purposes. Comparison of notations, in particular formal notations, have been proposed in the literature, concentrating in expressiveness in some earlier comparisons (e.g., [3]), and in analysis in more recent efforts (e.g., [20], [27], [30], [51]). Comparisons involving specifically Alloy, as in this paper, are also present in [36], [60], where the focus is in similarities with related languages, especially in expressiveness and style. The work reported in [5] studies the issues that arise in capturing UML class diagrams with OCL constraints into Alloy. These works however concentrate in Alloy as a language for *static* aspects of system modeling. The work in [58] is related to our current evaluation, since the authors compare alternative ways of capturing state change (mutability) in Alloy. The analysis is however more "low level", concentrating on alternative mechanisms to describe mutability within Alloy and their relative performances. The works introducing the extensions to deal with dynamic behavior in Alloy typically only compare against the original notation [21], [44], and not among alternative extensions. Moreover, some of these works involve comparisons that are now outdated, since they do not take into account modern optimizations that significantly improve analysis, compared to early versions of the tools (see previous section for more details on this issue).

Besides DynAlloy and Electrum, other extensions dealing with the problem of specifying dynamic behavior more appropriately have been proposed, notably [49]. The language presented therein shares similarities especially with DynAlloy, and also features some elements now present in Electrum. We left it out of the comparison due to a lack of an updated tool for analysis. Tools such as Squander [47] and TACO [26] combine Alloy with "imperative" notations, but do so for higher-level

programming languages, rather than software models. Thus, we also left them out of the comparison. Another proposal based on Alloy for modeling dynamic behavior is Dash [55], [56]. We left it out of the comparison because of two reasons: *(i)* the modeling approach in Dash is based on directly capturing a transition system, different from Electrum and DynAlloy, where the transition system is *implied* by a model at a higher level of abstraction; and *(ii)* at the time this study was initiated, the language for property specification [19], [59] was not supported as part of Dash's tool[3] (tool support has recently evolved to support temporal property specification). Dash uses Alloy as an analysis backend, but its modeling style is actually closer to other tools, as reflected in the comparison reported in [2]. Previous work attempted to capture the Chord model in Electrum [10]. The model therein is however a simplification of Zave's model, as fewer event interleavings are allowed for. Moreover, the model uses non-standard Electrum features (e.g., action definitions), that are not part of Electrum's public releases.

Earlier in this paper we also mentioned tools and notations that require capturing dynamism in Alloy [16], [26], [43], [53], [62]. These tools and notations can benefit from understanding the available Alloy alternatives to support dynamic behavior specification and analysis.

VIII. Conclusion

Formal specification languages, and in particular those that, like Alloy, put an emphasis in automated or semi-automated analysis, are useful tools for the analysis of system specifications. With the continuous development of new languages, with different characteristics and support for different analysis, and even different specification styles, it becomes important for software engineers to understand each language's peculiarities, to help engineers decide what is the most appropriate formalism and tool, for a specific specification or analysis problem. Efforts such as those around the steam boiler specification [3] or the electronic wallet [20], [27], [30], [51] show the need to better understand the modeling styles and associated analyses of different formal methods. In this paper, we tackled this problem, to study two different extensions of Alloy, that aim at better incorporating dynamic behavior specification into Alloy. Our comparison is based both in qualitative and quantitative aspects, and we discussed specification styles, tool usability, degree of abstraction offered by each style, expressiveness and performance. We conclude that both languages are faithful extensions to the Alloy style of specification, that ease the specification of system execution properties, with Electrum being at the same time the most abstract language (specifications in this language are at a higher level of abstraction) and the most expressive in relation to analysis (liveness properties are expressible and analyzable in Electrum). DynAlloy, on the other hand, is better suited for system specifications that are more directly captured through abstract programs (e.g., with nondeterminism), or have a more complex control flow.

[3]cf. http://dash.uwaterloo.ca:8080/dash/editor.html.

Since Alloy is designed with automated analysis in mind, this is a characteristic that both studied extensions maintain. In this respect, Electrum offers more flexibility, allowing users to select different analysis engines, including model checkers for the analysis of unbounded execution traces. DynAlloy offers the greatest scalability among the extensions for analyzing safety properties in a bounded fashion, with consistent significant speed-ups in relation to Electrum. We believe this performance of DynAlloy is related to how the translation of analysis problems into SAT is implemented by the tool. It does not seem that the optimizations in DynAlloy's translation are specific to DynAlloy: they could be exploited by Electrum developers too (at least as far as bounded safety verification is concerned). Alloy also showed good efficiency, paying a price, of course, in specification convenience compared to the evaluated extensions.

References

[1] "Detailed results and replication package for DynAlloy and electrum evaluation." [Online]. Available: https://sites.google.com/view/dynalloyelectrumevaluation

[2] A. Abbassi, A. Bandali, N. A. Day, and J. Serna, "A comparison of the declarative modelling languages B, Dash, and TLA+," in *Proc. 8th IEEE Int. Model-Driven Requirements Eng. Workshop (MoDRE)*, A. Moreira, G. Mussbacher, J. Araújo, and P. Sánchez, Eds., Banff, AB, Canada. Los Alamitos, CA, USA: IEEE Comput. Soc., Aug. 20, 2018, pp. 11–20.

[3] J.-R. Abrial, E. Börger, and H. Langmaack, Eds., *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (The Book Grow Out of a Dagstuhl Seminar)* (Lecture Notes in Computer Science), vol. 1165. Berlin, Heidelberg: Springer, Jun. 1995.

[4] B. Alpern and F. B. Schneider, "Defining liveness," *Inf. Process. Lett.*, vol. 21, no. 4, pp. 181–185, 1985.

[5] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray, "On challenges of model transformation from UML to Alloy," *Softw. Syst. Model.*, vol. 9, no. 1, pp. 69–86, 2010.

[6] L. Baresi and P. Spoletini, "On the use of alloy to analyze graph transformation systems," in *Proc. 3rd Int. Conf. Graph Transformations (ICGT)*, in Lecture Notes in Computer Science, A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, Eds., vol. 4178, Natal, Rio Grande do Norte, Sep. 17–23, 2006, pp. 306–320.

[7] G. Booch, J. E. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide—The Ultimate Tutorial to the UML From the Original Designers* (Object Technology Series). Reading, MA, USA: Addison-Wesley, 1999.

[8] J. P. Bowen and M. G. Hinchey, "Formal methods," in *Computing Handbook*, T. F. Gonzalez, J. Diaz-Herrera, and A. Tucker, Eds., 3rd ed.: Computer Science and Software Engineering. Boca Raton, FL, USA: CRC Press, 2014, pp. 71–25.

[9] J. Brunel, D. Chemouil, A. Cunha, and N. Macedo, "The electrum analyzer: Model checking relational first-order temporal specifications," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng. (ASE)*, M. Huchard, C. Kästner, and G. Fraser, Eds., Montpellier, France. New York, NY, USA: ACM, Sep. 3–7, 2018, pp. 884–887.

[10] J. Brunel, D. Chemouil, and J. Tawa, "Analyzing the fundamental liveness property of the Chord protocol," in *Proc. Formal Methods Comput. Aided Des. (FMCAD)*, N. S. Bjørner and A. Gurfinkel, Eds., Austin, TX, USA. Piscataway, NJ, USA: IEEE, Oct. 30–Nov. 2, 2018, pp. 1–9.

[11] A. Bucchiarone and J. P. Galeotti, "Dynamic software architectures verification using dynalloy," *Electron. Commun. EASST*, vol. 10, pp. 1–14, 2008.

[12] R. Cavada et al., "The NUXMV symbolic model checker," in *Proc. 26th Int. Conf. Comput. Aided Verification (CAV), Held as Part of Vienna Summer Log. (VSL)*, in Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559, Vienna, Austria. Cham, Switzerland: Springer, Jul. 18–22, 2014, pp. 334–342.

[13] E. M. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Proc. 10th Int. Conf. Tools Algorithms Constr. Anal. Syst.*

(TACAS), Held as Part of Joint Eur. Conf. Theory Pract. Softw. (ETAPS), Barcelona, Spain, Mar. 29–Apr. 2, 2004, pp. 168–176.

[14] L. C. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtík, "JBMC: A bounded model checking tool for verifying Java bytecode," in *Proc. 30th Int. Conf. Comput. Aided Verification (CAV), Held as Part of Federated Log. Conf. (FloC),* Oxford, U.K., Jul. 14–17, 2018, pp. 183–190.

[15] A. Cunha, "Bounded model checking of temporal formulas with Alloy," in *Proc. 4th Int. Conf. ABZ*, in Lecture Notes in Computer Science, vol. 8477. Toulouse, France. Berlin, Heidelberg: Springer, Jun. 2–6, 2014, pp. 303–308.

[16] G. Dennis, F. S.-H. Chang, and D. Jackson, "Modular verification of code with SAT," in *Proc. Int. Symp. Softw. Testing Anal. (ISSTA)*. New York, NY, USA: ACM, 2006, pp. 109–120.

[17] W. G. Marco Devillers, J. Romijn, and F. W. Vaandrager, "Verification of a leader election protocol: Formal methods applied to IEEE 1394," *Formal Methods Syst. Des.*, vol. 16, no. 3, pp. 307–320, 2000.

[18] E. W. Dijkstra and C. S. Scholten, *Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science*. New York, NY, USA: Springer, 1990.

[19] S. Farheen, N. A. Day, A. Vakili, and A. Abbassi, "Transitive-closure-based model checking (TCMC) in Alloy," *Softw. Syst. Model.,* vol. 19, no. 3, pp. 721–740, 2020.

[20] L. Freitas and J. Woodcock, "Mechanising Mondex with Z/Eves," *Formal Asp. Comput.*, vol. 20, no. 1, pp. 117–139, 2008.

[21] M. F. Frias, J. P. Galeotti, C. G. Pombo, and N. M. Aguirre, "DynAlloy: Upgrading alloy with actions," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*. New York, NY, USA: ACM, 2005, pp. 442–451.

[22] M. F. Frias, C. L. Pombo, G. Baum, N. Aguirre, and T. Maibaum, "Taking *Alloy* to the movies," in *Proc. Int. Symp. Formal Methods Eur. (FME)*, in Lecture Notes in Computer Science, K. Araki, S. Gnesi, and D. Mandrioli, Eds., vol. 2805, Pisa, Italy. Berlin, Heidelberg: Springer, Sep. 8–14, 2003, pp. 678–697.

[23] M. F. Frias, C. L. Pombo, G. A. Baum, N. Aguirre, and T. Maibaum, "Reasoning about static and dynamic properties in alloy: A purely relational approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 4, pp. 478–526, 2005.

[24] M. F. Frias, C. L. Pombo, J. P. Galeotti, and N. Aguirre, "Efficient analysis of DynAlloy specifications," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 1, pp. 1–34, 2007.

[25] D. M. Gabbay, A. Pnueli, S. Shelah, and J. Stavi, "On the temporal basis of fairness," in *Proc. Conf. Rec. 7th Annu. ACM Symp. Princ. Program. Lang.*, P. W. Abrahams, R. J. Lipton, and S. R. Bourne, Eds., Las Vegas, NV, USA. New York, NY, USA: ACM, Jan. 1980, pp. 163–173.

[26] J. P. Galeotti, N. Rosner, C. G. L. Pombo, and M. F. Frias, "TACO: Efficient SAT-based bounded verification using symmetry breaking and tight bounds," *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1283–1307, Sep. 2013.

[27] C. George and A. E. Haxthausen, "Specification, proof, and model checking of the mondex electronic purse using raise," *Formal Asp. Comput.*, vol. 20, no. 1, pp. 101–116, 2008.

[28] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2002.

[29] M. Hammad, H. Bagheri, and S. Malek, "DelDroid: An automated approach for determination and enforcement of least-privilege architecture in android," *J. Syst. Softw.*, vol. 149, no. 83, pp. 83–100, 2019.

[30] D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif, "Verification of Mondex electronic purses with KIV: From transactions to a security protocol," *Formal Asp. Comput.*, vol. 20, no. 1, pp. 41–59, 2008.

[31] D. Harel, J. Tiuryn, and D. Kozen, *Dynamic Logic*. Cambridge, MA, USA: MIT Press, 2000.

[32] M. Hinchey, M. Jackson, P. Cousot, B. Cook, J. P. Bowen, and T. Margaria, "Software engineering and formal methods," *Commun. ACM*, vol. 51, no. 9, pp. 54–59, 2008.

[33] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[34] G. J. Holzmann, *The SPIN Model Checker—Primer and Reference Manual*. Reading, MA, USA: Addison-Wesley, 2004.

[35] M. Huth and M. D. Ryan, *Logic in Computer Science—Modelling and Reasoning About Systems*, 2nd ed. Cambridge, U.K.: Cambridge Univ. Press, 2004.

[36] D. Jackson, "A comparison of object modelling notations: Alloy, UML and Z," MIT Lab for Computer Science, unpublished, 1999.

[37] D. Jackson, "Alloy: A lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[38] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. Cambridge, MA, USA: MIT Press, 2006.

[39] D. Jackson, I. Shlyakhter, and M. Sridharan, "A micromodularity mechanism," in *Proc. 8th Eur. Softw. Eng. Conf. Held Jointly With 9th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, Vienna, Austria, Sep. 10–14, 2001, pp. 62–73.

[40] S. A. Khalek, G. Yang, L. Zhang, D. Marinov, and S. Khurshid, "TestEra: A tool for testing Java programs using alloy specifications," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng. (ASE)*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds., Lawrence, KS, USA. Los Alamitos, CA, USA: IEEE Comput. Soc., Nov. 6–10, 2011, pp. 608–611.

[41] D. Kroening and O. Strichman, *Decision Procedures—An Algorithmic Point of View* (Texts in Theoretical Computer Science. An EATCS Series), 2nd ed. Berlin, Heidelberg: Springer, 2016.

[42] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Reading, MA, USA: Addison-Wesley, 2002.

[43] Y. Laurent, R. Bendraou, S. Baarir, and M.-P. Gervais, "Formalization of fUML: An application to process verification," in *Proc. 26th Int. Conf. Adv. Inf. Syst. Eng. (CAiSE)*, in Lecture Notes in Computer Science, M. Jarke et al., Eds., vol. 8484, Thessaloniki, Greece. Cham, Switzerland: Springer, Jun. 16–20, 2014, pp. 347–363.

[44] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng. (FSE)*, T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds., Seattle, WA, USA. New York, NY, USA: ACM, Nov. 13–18, 2016, pp. 373–383.

[45] J. Magee and J. Kramer, *Concurrency—State Models and Java Programs*. Hoboken, NJ, USA: Wiley, 1999.

[46] Z. Manna and A. Pnueli, "Adequate proof principles for invariance and liveness properties of concurrent programs," *Sci. Comput. Program.*, vol. 4, no. 3, pp. 257–289, 1984.

[47] A. Milicevic, D. Rayside, K. Yessenov, and D. Jackson, "Unifying execution of imperative and declarative code," in *Proc. 33rd Int. Conf. Softw. Eng. (ICSE)*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., Waikiki, Honolulu, HI, USA. New York, NY, USA: ACM, May 21–28, 2011, pp. 511–520.

[48] M. M. Moscato, C. L. Pombo, and M. F. Frias, "Dynamite: A tool for the verification of alloy models based on PVS," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, pp. 1–37, 2014.

[49] J. P. Near and D. Jackson, "An imperative extension to alloy," in *Proc. 2nd Int. Conf. Abstr. State Mach. Alloy B Z (ABZ)*, in Lecture Notes in Computer Science, M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, Eds., vol. 5977, Orford, QC, Canada. Springer, Feb. 22–25, 2010, pp. 118–131.

[50] D. A. Peled, *Software Reliability Methods. Texts in Computer Science*. New York, NY, USA: Springer, 2001.

[51] T. Ramananandro, "*Mondex*, an electronic purse: Specification and refinement checks with the *Alloy* model-finding method," *Formal Asp. Comput.*, vol. 20, no. 1, pp. 21–39, 2008.

[52] G. Regis et al., "DynAlloy analyzer: A tool for the specification and analysis of alloy models with dynamic behaviour," in *Proc. 11th Joint Meet. Found. Softw. Eng., (ESEC/FSE)*, E. Bodden, W. Schäfer, A. Deursen, and A. Zisman, Eds., Paderborn, Germany. New York, NY, USA: ACM, Sep. 4–8, 2017, pp. 969–973.

[53] J. A. Ross, A. Murashkin, J. H. Liang, M. Antkiewicz, and K. Czarnecki, "Synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded systems," *Softw. Syst. Model.*, vol. 18, no. 1, pp. 739–767, 2019.

[54] B. L. Schwartz, "An analytic method for the 'difficult crossing' puzzles," *Math. Mag.*, vol. 34, no. 4, pp. 187–193, 1961.

[55] J. Serna, N. A. Day, and S. Esmaeilsabzali, "Dash: Declarative behavioural modelling in alloy with control state hierarchy," *Softw. Syst. Model.*, vol. 22, no. 2, pp. 733–749, 2023.

[56] J. Serna, N. A. Day, and S. Farheen, "Dash: A new language for declarative behavioural requirements with control state hierarchy," in *Proc. IEEE 25th Int. Requirements Eng. (RE) Conf. Workshops*, Lisbon, Portugal. Los Alamitos, CA, USA: IEEE Comput. Soc., Sep. 4–8, 2017, pp. 64–68.

[57] I. Stoica, R. T. Morris, D. R. Karger, M. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protoc. Comput. Commun.*, R. L. Cruz and G. Varghese, Eds., San Diego, CA, USA, Aug. 27–31, 2001, pp. 149–160.

[58] A. Sullivan, K. Wang, S. Khurshid, and D. Marinov, "Evaluating state modeling techniques in alloy," in *Proc. 6th Workshop Softw. Qual. Anal. Monit. Improvement* Appl., Z. Budimac, Ed., vol. 1938, Belgrade, Serbia. Sep. 11–13, 2017. [Online]. Available: https://ceur-ws.org/
[59] A. Vakili and N. A. Day, "Temporal logic model checking in Alloy," in *Proc. 3rd Int. Conf. Abstr. State Mach. Alloy B VDM Z (ABZ)*, in Lecture Notes in Computer Science, J. Derrick et al., Eds., vol. 7316, Pisa, Italy. Berlin, Heidelberg: Springer, Jun. 18–21, 2012, pp. 150–163.
[60] M. Vaziri and D. Jackson, "Some shortcomings of OCL, the object constraint language of UML," in *Proc. 34th Int. Conf. Technol. Object-Oriented Lang. Syst. (TOOLS)*, Q. Li, D. Firesmith, R. Riehle, and B. Meyer, Eds., Santa Barbara, CA, USA, Jul. 30–Aug. 3, 2000, pp. 555–562.
[61] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs," *Autom. Softw. Eng.*, vol. 10, no. 2, pp. 203–232, 2003.
[62] C. Wallace, "Using alloy in process modelling," *Inf. Softw. Technol.*, vol. 45, no. 15, pp. 1031–1043, 2003.
[63] J. M. Wing, "A specifier's introduction to formal methods," *IEEE Comput.*, vol. 23, no. 9, pp. 8–22, Sep. 1990.
[64] P. Zave, "Using lightweight modeling to understand chord," *Comput. Commun. Rev.*, vol. 42, no. 2, 2012.
[65] P. Zave, "A practical comparison of alloy and spin," *Formal Asp. Comput.*, vol. 27, no. 2, 2015.
[66] P. Zave, "Reasoning about identifier spaces: How to make Chord correct," *IEEE Trans. Softw. Eng.*, vol. 43, no. 12, pp. 1144–1156, Dec. 2017.

César Cornejo is pursuing a Ph.D. degree in Computer Science with FAMAF, University of Córdoba, Argentina, under a CONICET scholarship. He belongs to the Formal Methods and Software Engineering Group, Department of Computer Science, University of Río Cuarto, Argentina, where he is also a Teaching Assistant. His research interests are software verification, program analysis, and specification languages.

Germán E. Regis received the Ph.D. degree from the University of Buenos Aires, Argentina. He is currently an Adjunct Professor with the Department of Computer Science, University of Río Cuarto, Argentina, and a member of the Formal Methods and Software Engineering Group, with the same Department. His research interests are the application of formal methods to the specification, design, analysis, and verification of software.

Nazareno Aguirre received the Ph.D. degree from King's College, University of London, U.K. He is currently an Associate Professor with the Computer Science Department, University of Rio Cuarto, Argentina, and a Researcher with Argentina's National Council for Scientific and Technical Research (CONICET). His research interests focus on software engineering, in particular formal specification, software testing, and program verification. He is currently serving as an Associate Editor of IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.

Marcelo F. Frias is a Professor with the Department of Computer Science at the University of Texas at El Paso. He has been a Program Committee Member of many conferences, including ASE, ISSTA, ICST, FME, and ICSE. He has published in ACM TOSEM and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. His research has received funding from AWS, The Qatar Foundation, and The University of Texas at El Paso.