ALDB: Debugging Alloy Models of Behavioural Requirements

Aman Dureja, Aditya Keerthi, Andrew Liang, Paul Zhang, and Nancy A. Day
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, Canada N2L 3G1
Email: {adureja, a2keerth, a29liang, jl5zhang, nday} @uwaterloo.ca

***Abstract*—Declarative modelling languages, such as Alloy, are becoming popular for describing behavioural requirements very early in system development because automated analysis of these models provides valuable feedback. Typically, these languages are supported by constraint solvers (SAT, SMT) for providing instances or model checking properties. However, a user can quickly find simple bugs and gain confidence in their model by concretely simulating steps of the transition system. We present ALDB: a debugger for models of transition systems written in the Alloy language. It provides a familiar debugging interface to walk around in the behaviour of the model, enabling users to quickly explore scenarios, find errors via concrete simulation, and incrementally build up to bounded model checking.**

I. Introduction

Model-driven engineering (MDE) [1] seeks to conquer complexity through the use of more abstract descriptions of a system than code. While MDE approaches generally centre around the Unified Modelling Languages (UML) [2], there is a growing area of research surrounding declarative formal modelling languages. These languages, such as Alloy [3], TLA+ [4], and Event-B [5], are well-suited for describing behavioural requirements because they allow users to write formal models very early in the development process to explore designs and catch errors before going to the effort of creating a UML model or coding. These models are precise, but not necessarily detailed, through the use of uninterpreted functions and constraints rather than operational language constructs. For example, an Alloy model consists of constraints over sets and relations where the constraints are written in terms of set operators plus the transitive closure operator. There are several examples of the effective use of declarative modelling languages, such as using Alloy to find bugs in the CHORD protocol [6] and using TLA+ to find bugs in concurrency designs at Amazon [7]. In this paper, we focus on declarative behavioural models (transition systems).

Generally, declarative modelling languages are supported by automated formal analysis tools that search for instances or counterexamples over a finite scope of values for each set. Limiting the search to a finite scope makes it tractable, but such analysis may take time, and it can be harder to debug a model from static feedback such as a model instance or an entire counterexample trace. In particular, for the popular Alloy Analyzer, an instance of a behavioural model is a static representation of either a path or the entire transition system. Techniques for temporal logic model checking of Alloy models are being developed (e.g., [8], [9]), but users might like to explore the behaviour of an abstract model of a transition system through debugging/simulation first to find simple bugs and to understand the behaviour prior to waiting for a solver to complete.

Driven by the goal of providing quicker, more interactive feedback to Alloy users, we have created a debugger for Alloy models of transition systems. The idea is to provide a modeller with the functions needed to explore an Alloy model of a transition system by stepping through it in the fashion of a typical debugger. Currently, iterating through a relation must be done by including facts in the Alloy model that either create a path by explicitly iterating the relation a fixed number of times (as in bounded model checking [10]) or writing formulas in the Alloy Analyzer's interactive evaluator component. What is missing is a tool that focuses particularly on transition systems and provides the commonly-understood interactive debugging interface.

Providing debugging functionality for declarative modelling languages has different challenges than creating a debugger for code. Declarative models are abstract (e.g., sets/relations/functions) and the transition relation can be non-deterministic. The next step cannot be "computed" directly, but rather interaction with a solver is needed to determine the possible next states. Because of the non-determinism, the choice of a particular next state may eliminate some future paths of interest for exploration. Debugging needs both functions for step-by-step and backtracking interaction, and goal-based functions to find paths that reach or follow states based on user-chosen constraints. Debugging a declarative model is an incremental process that goes from local exploration all the way to bounded model checking.

Existing tools for declarative modelling provide some of these debugging/simulation features, although we are not aware of any such tool for Alloy. TLC [11], the model checker for TLA+ can generate a randomly chosen path through the model. ProB [12] provides users with the ability to take individual steps of the model. NuSMV [13] and NuXMV [14] provide interactive simulation functions that allow both stepping through a model and finding paths that satisfy constraints but their modelling languages are less abstract than Alloy.

In this paper, we describe a set of functions for our new tool ALDB, which is a debugger for Alloy models of transition systems. ALDB provides a command-line interface but is integrated with the code base of the Alloy Analyzer through the use of debugging function templates. ALDB is available for download at: HTTPS://GITHUB.COM/WATFORM/ALDB. We also describe how ALDB can be used in a case study.

Declarative modelling languages provide users with the ability to write even more abstract models than those of UML and thus provide a stepping stone in conquering complexity between requirements and design. A familiar debugging-like interface for Alloy models will support modellers in transitioning to this new modelling paradigm by immediately providing feedback on their models. We believe ALDB provides valuable functionality to both novice and advanced Alloy users. For novice users, it helps them learn the effects of Alloy language structures. For advanced users, it helps them zero in quickly on modelling mistakes or problematic cases in the model.

II. Background

Alloy is a declarative language based on relational logic and models system components as sets. A user can create a set as a signature, declare relations between sets, and write facts, which are invariants about the relations and sets. Predicates (similar to macros) are used to define formulas, and may be parametrized. Alloy's graphical tool - the Alloy Analyzer - produces satisfying instances of a model and exposes invariant violations by generating counterexamples. Finite scopes (sizes) for each set are required, which restrict the universe that the Analyzer explores.

The focus of our work is on Alloy models of transition systems. A transition system consist of a set of states and a transition relation that operates on them. In Alloy, there are two idioms for defining a transition system, which we will call implicit and explicit state. Implicit state transition system models encode the changing behaviour in relations from an object to a state or time [3]. Explicit state systems utilize a dedicated state signature and the transition relation is a binary relation between states [15]. Additionally, there are relations from the states to the dynamic elements of the model (similar to treating the state as a record/structure). Sullivan et al. compare these styles of modelling in Alloy [16]. ALDB assumes explicit state models since it localizes the definition of the state and it has only one transition relation (rather than multiple relations that together form the transition relation).

In detail, ALDB assumes that state models consist of a `State` signature, an `init` predicate, and a `next` predicate, as follows:

```
sig State { ... }
pred init [s: State] { ... }
pred next [s, s': State] { ... }
```

The names of these elements are configurable. The `State` signature contains the elements of the state that change value with transitions. There can be other non-dynamic elements in the model that do not need to be part of the `State` signature. The `init` predicate contains constraints on the initial state of the system. The `next` predicate is the transition relation, and relates s and s' where s is the current state and s' is the next state. As it is common, we assume that the user has defined a unique (although possibly non-deterministic) transition relation.

As an illustrative example of an Alloy transition system, we use the classic River Crossing Problem (RCP). In the RCP, there exists a farmer, a fox, a chicken, a bag of grain, and a small boat. There is a river dividing two sides of land: the near side and the far side. Every entity starts on the near side. The goal is to use the boat to transport all entities to the far side of the river. There are some restrictions:

- The boat can only hold two entities at any time. One of the entities must be the farmer, as only they can operate the boat.
- If left together without the farmer, the fox will eat the chicken.
- If left together without the farmer, the chicken will eat the bag of grain.

Listing 1 is an explicit state Alloy model representing the RCP[1] [17].

Currently, in the Alloy Analyzer, if a user wishes to investigate what happens in two steps of their model, in addition to the above model, they must perform the following process:

1) Load the model into the Analyzer.
2) Optionally (but commonly), import the util/ordering module to create a path via a total ordering on states.
3) Define a fact that constrains the first state in the total order to satisfy the `init` predicate.
4) Define a fact that constrains the linear order to respect the transition relation, meaning a state can follow another in the order only if it is related by the `next` predicate.
5) Add a line to the end of the model that searches for an instance of the linear order of a certain length.
6) Observe the entire path in the Analyzer visualizer.

The facts mentioned above could be written in the evaluator and the visualizer can be configured to isolate parts of the model or show the states in different views.

But this process either requires non-trivial manual model instrumentation which bloats the model with debugging specifics, or requires the modeller to type in long formulas in the evaluator. If the user desires to see a path that reaches a state where a certain condition is true, as in bounded model checking, then they must further instrument the model.

Activities such as stepping and running a system until specific constraints are satisfied (breakpoints) are commonly performed when designing programs (which are transition systems), in order to debug incorrect behaviour or to observe system execution. Manual model instrumentation as previously mentioned is fragile and opens the model up to more faults. The process also requires the user to always load the graphical Analyzer tool, which can be undesirable for those who work primarily in command-line environments, disrupting

[1]The model presented here very closely matches the Alloy of RCP found in https://github.com/AlloyTools/org.alloytools.alloy/blob/master/org.alloytools.alloy.extra/extra/models/examples/tutorial/farmer.als .

```
/* Farmer and his possessions are objects. */
abstract sig Object {
  /* eats is a binary relation on Objects */
  eats: set Object
}
/* Particular objects of the model */
one sig Farmer, Fox, Chicken, Grain
        extends Object {}

/* Define what eats what
   when the farmer is not around. */
fact { eats = Fox->Chicken + Chicken->Grain }

/* States of the transition relation consist of:
  1) a set of Objects on the near side
  2) a set of Objects on the far side
*/
sig State { near, far: set Object }

/* In the initial state, all objects
   are on the near side. */
pred init [s: State] {
  (s.near = Object) && (no s.far)
}

/* At most one item to move from
   'from' to 'to'; eating occurs if possible. */
pred crossRiver
  [from, from', to, to': set Object] {
  one x: from | {
    from' = from - x - Farmer - from'.eats
    to' = to + x + Farmer
  }
}

/* Transition Relation */
pred next [s, s': State] {
  Farmer in s.near =>
    /* cross from near to far */
    crossRiver [s.near, s'.near, s.far, s'.far]
  else
    /* cross from far to near */
    crossRiver [s.far, s'.far, s.near, s'.near]
}
```

Listing 1. Alloy model of River Crossing Problem (RCP)

their workflow. Hence, there is a need for a command-line tool that performs desired debugging functions on transition systems.

III. Illustrative Example

In order to demonstrate its functionality, in this section we show the use of ALDB to explore and find a solution to the RCP that was introduced earlier.

We begin by loading the model into ALDB:

```
(aldb) load river_crossing.als
Reading model from river_crossing.als...done.
(aldb) current

S1
----
far: {  }
near: { Chicken, Farmer, Fox, Grain }
```

The changing state in the RCP Alloy model is the set of objects that are on the near and far sides. There is only one initial state that satisfies the `init` constraint: everything is on the near side and nothing is on the far side. The river can only be crossed starting from where the farmer currently is, and ending on the opposite side. When crossing the river, the farmer has the option to take any one of the other objects with them, with the consequence that some object may be eaten if left unattended.

Using ALDB, a modeller can incrementally explore the model's state space by stepping (continuing the example run from above):

```
(aldb) step

S2
----
far: { Farmer, Grain }
near: { Fox }
```

The reached state in this step does not contain all the objects. The chicken is missing because in this specific execution path, the farmer took the grain and left the fox to eat the chicken. This behaviour is undesired. The `reverse-step` function can be used to go back to the initial state, or using the `alt` command we can explore other states that could have been reached (continuing the example run above):

```
(aldb) alt

S3
----
far: { Farmer, Fox }
near: { Chicken }

(aldb) alt

S4
----
far: { Chicken, Farmer }
near: { Fox, Grain }
```

In the first alternate state, the farmer takes the fox, leaving the chicken to eat the grain. The second alternate state from the initial state may lead to a valid solution to the puzzle as all objects are still present. If the user wishes to see only steps that lead to states where all entities exist, then they can leverage constrained stepping with a formula alias. We begin by using the `init` command to return to the model's initial state, as follows:

```
(aldb) init

S1
----
far: {  }
near: { Chicken, Farmer, Fox, Grain }

(aldb) alias myFormula "near + far = Object"
(aldb) step [myFormula]

S2
----
far: { Chicken, Farmer }
near: { Fox, Grain }
```

Here, the user specifies a formula where the union of the near and far sets is equal to the Object set (i.e., all entities). The formula is aliased as `myFormula` for convenience. When the step is performed, ALDB takes one step in the transition system that has a destination state that satisfies `myFormula`.

The question that we want to answer in this puzzle is: what sequence of events – if any – results in every entity safely reaching the far side of the river? Since the desired end state of the puzzle is known, using ALDB's `until` command with a breakpoint is the quickest method to get a solution to the puzzle. The breakpoint is set to a constraint where the far side contains all the objects, and there is nothing on the near side, as follows:

```
(aldb) init

S1
----
far: { }
near: { Chicken, Farmer, Fox, Grain }

(aldb) break "far = Object && no near"
(aldb) until

S8
----
far: { Chicken, Farmer, Fox, Grain }
near: {  }
```

The desired state has been reached. By default the `until` function takes up to 10 steps, but a user can choose an alternative number of steps. We can use the `history` function to show the sequence of transitions that resulted in this end state as shown in Listing 2. From this history, we know that the desired end state can be reached in seven transitions.

The process of manually exploring the state space exposes execution states that a user might not see in an instance produced by the Alloy Visualizer. Looking at concrete states allows the user to quickly find errors in the model.

```
(aldb) history 10

S1 (-7)
---------
far: {  }
near: { Farmer, Fox, Chicken, Grain }

S2 (-6)
---------
far: { Chicken, Farmer }
near: { Fox, Grain }

S3 (-5)
---------
far: { Chicken }
near: { Farmer, Fox, Grain }

S4 (-4)
---------
far: { Chicken, Farmer, Fox }
near: { Grain }

S5 (-3)
---------
far: { Fox }
near: { Chicken, Farmer, Grain }

S6 (-2)
---------
far: { Farmer, Fox, Grain }
near: { Chicken }

S7 (-1)
---------
far: { Fox, Grain }
near: { Chicken, Farmer }
```

Listing 2. Output of History for RCP Debugging Interaction

IV. Debugging Functions

Inspired by common code debugging tools, we created the functions described in this section for Alloy debugging. We describe how these functions are customized for a declarative model. For any of our functions that require constraint solving, a template is populated to pass to the Alloy Analyzer's solver. ALDB then interprets the solver's result to display the concrete states of the model as needed for the output of the function. The template instantiation uses information stored internally to ALDB about the explored states. This method is easily extensible to add more functionality to ALDB.

In order to understand the function descriptions, we first introduce the internal data structures that ALDB uses to keep track of states and execution traces. StateGraph is a directed graph that represents the parts of the transition system that the user has explored so far in the debugging session. Each node represents a concrete state (with unique values for its state elements), and each edge represents a transition. We build StateGraph incrementally as the user steps through a transition system and explores its state space. Two states are equivalent if they have same values for their state elements[2]. Thus, the StateGraph may contain loops. While this graph may grow large, because it is based on interactive use, we do not expect state space explosion problems. StatePath is a subgraph of StateGraph, and represents the path of the transition system currently being followed in this debugging session.

Next, we describe the functions of ALDB, how they match to constraint problems in Alloy, and how they change the StateGraph and StatePath.

A. *Configuration*

ALDB operates on transition systems modelled using the explicit state modelling idiom. The names of the expected signatures and predicates can be set in a custom configuration.

The configuration is defined in YAML. It can be specified within a comment block in the model file, or set via a separate YAML file using the `set conf [path/to/config/file]` function. Listing 3 shows an example of a configuration block within an Alloy model. It has the following parts:

- `stateSigName` is the name of the signature that represents the state set.

[2]Alloy allows different atoms of the state set to relate to the same values for state elements, which gives the appearance of two different state names for the same state. In ALDB, we collapse these differences for the user.

```
/*  BEGIN_ALDB_CONF
 *
 *  stateSigName: State
 *  transitionRelationName: next
 *  initPredicateName: init
 *  additionalSigScopes:
 *    Team: 4
 *    Runner: 15

 *  END_ALDB_CONF
 */

sig State { ... }
pred init[s: State] { ... }
pred next[s,s':State] { ... }
```

Listing 3. ALDB Configuration File

```
1  open util/ordering[State]
2
3  // Information about current state
4  // Example:
5  // field1 -> {}
6  // field2 -> {a, b}
7  // ...
8
9  < all of original model (except init predicate) >
10
11 // Generated init predicate
12 pred init[s: State] {
13   // Example:
14     s.field1 = none
15     s.field2 = a + b
16     ...
17 }
18
19 fact { init[first] }
20
21 fact {
22     all s: State, s': s.next { next[s, s']  }
23 }
24
25 run {} for exactly <1 + inputNumStates> State
```

Listing 4. Template for Step Function

- `transitionRelationName` is the name of the transition relation predicate.
- `initPredicateName` is the name of the predicate that constrains the initial states of the transition system.
- `additionalSigScopes` is an optional map of (String, Integer) pairs that define specific scopes for signatures (other than the `State` signature) within the model. These override any choice of scopes included in the model. The scope of a set cannot be changed in the middle of a debugging session without reloading the model and restarting execution.

When a model is loaded, the session begins with ALDB choosing an initial state by asking the Alloy solver to return a state that satisfies the `init` predicate. This initial state forms the first node of the internal StateGraph and StatePath data structures.

B. `Step` and `Reverse-Step` Functions

Input to the `step` function is the number of steps to take. The template for the step function is shown in Listing 4. A temporary Alloy file is created from this template. In the template, ALDB creates a new `init` predicate based on the current state, which is maintained in an internal data structure. The `util/ordering` module is imported to create a path of states via imposing a total ordering on the `State` set. The ordering module reserves the names `first` and `next` (as in `s.next`) to be the first element in the order and the function to get the next element in the order respectively. The first state must satisfy the new `init` predicate (line 19). Then, we must tell Alloy that all consecutive states are related by the next predicate of the transition system (lines 21–23). Finally, Alloy must actually run the system for the given number of steps. We set the scope of the `State` set to one more than the desired number of steps to account for states at both ends of the path (line 25).

The Alloy solver returns an instance of the appropriate length path. ALDB displays the last of the states in the path to the user, but also updates the internal StateGraph structure to keep track of visited states. If Alloy returns two states with identical values for their state elements, internally, this is represented as a loop in StateGraph[3]. The `step` function output shows only the fields that have changed in the step.

The `reverse-step` function returns the previous step on the path and does not require interaction with the solver because of the internal StatePath data structure.

C. `Alt` Function

The `alt` function allows users to explore alternate states that are reachable from the previous state as was shown in Section III. Using the `alt` function the user can incrementally explore the complete state space of a transition system. It can also be used at the initial state to view the set of all possible initial states.

Using the StateGraph, ALDB first looks to see if there are any previously discovered alternative states that have not yet been displayed to the user (at this time) to choose an alternative state. If not, rather than using a template, it is more efficient to leverage the previous solver run by asking the Alloy solver for its "next" solution.

D. `Until` and `Break` Functions

The `until` function is used together with breakpoints. The user can add any number of formulas that are breakpoints to halt continued exploration of a path in the model. The formula of a breakpoint is entered via the `break` function (possibly using a formula alias for convenience).

The input to the `until` function is the maximum number of steps to explore. ALDB generates a path from the current

[3]The ordering module will not present a path with a loop in it (because then the states are not in a total order), thus it is appropriate to allow Alloy to have multiple states related to the same state values. A modeller should avoid including a fact that two states cannot contain the same values for their state elements.

state that is either as long as this maximum number of steps or reaches a state where one of the breakpoints is true.

To instantiate the template for the `until` function, the formulas entered as breakpoints are disjuncted together to create a predicate in Alloy. There are two different stopping points for the `until` function: when the breakpoint is reached or when the maximum path length is checked. We sequentially run the Alloy solver for every path length between one and the specified maximum number of steps. For each of these runs we use a template similar to the one for the `step`, except that the `break` predicate must be true in the last state of the run, as in:

```
// User-specified breakpoint formulas:
// Example
// breakpoint1 = "field1 = a"
// breakpoint2 = "field1 + field2 = a + b"

// Generated break predicate:
pred break[s: State] {
    (s.field1 = a) or
    (s.field1 + s.field2 = a + b)
}

fact {
    break[last]
}

run { } for exactly <numSteps + 1 > State
```

If the solver finds a solution that satisfies the break predicate at a certain path length, then we do not check longer path lengths. If we execute the solver with the maximum number of steps and it still returns no satisfiable solution, then we inform the user that the breakpoint could not be satisfied within the specified maximum path length. This iteration over path length ensures that less solver time will be taken when shorter paths (than the maximum) satisfy the breakpoint.

Our function `until` is bounded model checking but phrased in terms of common debugging functions for the user. For example, model checking that an invariant is true for all paths of lengths up to k can be done using a breakpoint on the negation of the invariant and running ALDB's `until` function for k steps.

E. Step Function with Path Constraints

As an alternative to providing the number of steps as input to the `step` function, users can provide a comma-separated list of formulas to be satisfied at each state in a path. This function can be used to simulate inputs for the transition system by setting a constraint that includes a formula containing a value for an input. Also, using formula aliases, it is easy to constain a path where every step satisfies an invariant. For a list of length n, the function will execute n steps, and each state i in the path will satisfy the formula at position i in the list.

In order to perform the path-constrained step, we instantiate an Alloy model template with predicates to represent the constraint at each step in the path and apply these constraints to the appropriate state in the ordering of the `State` set (lines 22– 26) as shown in Listing 5. If no such execution path exists, then the user is informed that the constrained step could

```
1  // User specified formula aliases:
2  // Example:
3  // f1 = "field1 = a"
4  // f2 = "field1 + field2 = a + b"
5
6  // User enters: step [f1, f2, f1]
7
8  // Generated Alloy code:
9
10 pred path_s1[s: State] {
11     s.field1 = a
12 }
13
14 pred path_s2[s: State] {
15     s.field1 + s.field2 = a + b
16 }
17
18 pred path_s3[s: State] {
19     s.field1 = a
20 }
21
22 pred path[s: State] {
23     path_s1[s.next] and
24     path_s2[s.next.next] and
25     path_s3[s.next.next.next]
26 }
27
28 fact {
29     init[first]
30     path[first]
31 }
32
33 run { } for
34     exactly <length of constraint list> State
```

Listing 5. Template for Step with Path Constraints

not be completed. Otherwise, it is guaranteed to find a path if one exists. Note that this is unlike other simulation methods that do not 'look ahead' to ensure that the state chosen in the next step has a future path that satisfies all the constraints.

F. Trace Mode

The Alloy Analyzer is able to generate counterexample paths when inconsistencies are found during bounded model checking. A counterexample can be exported from the Analyzer as an XML file that encodes a specific erroneous execution trace. ALDB is able to load a counterexample file, parse the XML, and convert it to an internal representation for user exploration. Loading a counterexample does not require the user to load the original Alloy model from which the counterexample was derived. As such, a limitation is that ALDB cannot find alternate states, nor step beyond the final state of the counterexample. Trace mode is useful however to highlight what state elements change in each step.

G. Session Log

Every execution of ALDB is considered to be a unique session. When ALDB is started, a session log is created in a temporary file. This file records every full, completed function executed in the current session. If a session is terminated, it can be recovered up to the point of the last completed command by starting ALDB with the `--restore` flag and the file path of the desired session log to restore from. A new session log with the contents of the previous session's log - and any further commands - will be created for the new session. This file can also be used for scripting a debugging session.

H. Additional Functions

ALDB implements a number of additional functions such as:

- `alias`: label a formula. As shown in the example in Section III, to avoid having to type large formulas multiple times, a label can be given to a formula and reused in any function that has formulas as arguments.
- `init`: return to an initial state. This function returns to the first initial state selected by ALDB in this session. Multiple initial states can be explored by using the `alt` function when an initial state is the current state.
- `scope`: show the elements in the scope of a set. This function is useful to learn what values are possible for a state element.
- `dot`: output a graph of the state space explored so far in the DOT graph description language [18] where it can be visualized using packages such as graphviz [19].

V. Implementation

In order to implement ALDB functions, we leverage the public Alloy Analyzer code[4]. Interfacing directly with the Alloy API maximizes compatibility between Alloy and ALDB. Whenever Alloy introduces new features or better solvers, the changes will be immediately reflected in ALDB when it is updated to use the latest Alloy JAR.

We use the Alloy Analyzer's existing code to parse models and use the Alloy API to call its solvers to find satisfying instances of Alloy models created using our template.

A key Alloy data structure we rely on is the one containing the instances returned from the solver. This object contains information about the system state at every step. Furthermore, the instance data structure exposes a "next" method which returns another instance with an alternate path. This method allows us to show the user alternate states at each step, and build a complete StateGraph for the model.

We are currently considering how ALDB might be integrated directly into the Alloy evaluator, which is part of its graphical user interface, but there is value in continuing to have ALDB's functionality at the command-line.

VI. Evaluation

Beyond testing for correctness, to evaluate the effectiveness of ALDB as a debugger for Alloy models of transition systems, we tackled creating a new model using ALDB to help with the debugging process. Our goal in this section is to provide readers with a sense of how ALDB can be used in their workflow.

We received an informal written specification of a program called FastFeet (FF) [20], which had been previously used as formal specification exercise in a second year course at the University of Waterloo. From this informal description, we wrote a formal Alloy model[5] FF manages a running race. There are multiple teams of runners, and a runner can belong only to a single team. A team consists of a maximum of five runners, and a minimum of one. When a race is over, FF returns the result: a list of runners in the order that they finished in the race. Initially, nobody is entered in a race and no result exists. The following are the system operations:

- Register Team: Add a team and its runners to the race.
- Substitute Runner: Replace a runner on a particular team with another runner.
- Run Race: Obtain an ordered list of finishing runners. Some runners may not have finished.
- Disqualify Team: Remove all runners from a particular team from the results.
- Compute Score: For each team that has a full set of runners that finished, obtain the sum of finishing positions for the team.

When writing the Alloy model for FF, we began by encoding information about the changing entities into a State signature. We wrote a predicate that encapsulates the behaviour of each system operation (sys_op). Then we defined an init predicate to specify initial conditions, and a next predicate as the transition relation. In each step, one of the FF operations is taken.

We utilized ALDB to incrementally test and build a correct model. Within the ALDB config, we used `additionalSigScopes` to set the scope of the Team and Runner signatures. For simplicity, we used four teams and fifteen runners.

Our primary debugging strategy was to leverage ALDB's constrained step functionality. The various system operations were tested using the command: `step ["operation=SYS_OP_NAME"]`. The debugging method allowed us to examine individually the effects of each operation. We then verified that the output corresponded to what we expected. If the output was surprising, then it signalled that the sys_op predicate was erroneous, particularly the statements that referenced the state elements with unexpected results.

Next, we wanted to inspect how a race would progress once it began, so we used ALDB's `until` function to get to a state where the race has started, and then manually stepped to observe transitions. Listing 6 shows an unexpected state transition. The operation in S4 is "SUBSTITUTE_RUNNER". According to the informal specification, it does not make sense for a team to substitute a runner during a race. As such, ALDB exposed a flaw in the model where we neglected to ensure that the "SUBSITUTE_RUNNER" operation should only be possible if the current mode is "REGISTRATION", meaning that it occurs before a race begins.

ALDB's `until` functionality allowed us to test reachability of the end of the race. We set a breakpoint for the existence

[4]https://github.com/AlloyTools/org.alloytools.alloy

[5]Our complete model in Alloy is available at: https://github.com/WatForm/aldb/blob/master/models/river_crossing.als .

```
(aldb) load fast_feet.als
Reading model from fast_feet.als...done.
(aldb) break "mode=RACE"
(aldb) until

S3
----
mode: { RACE }
operation: { RUN_RACE }
results: {  }
roster: { Team_3->Runner_1, Team_3->Runner_10 }
runners: { Runner_1, Runner_10 }
scores: {  }
teams: { Team_3 }

(aldb) step

S4
----
operation: { SUBSTITUTE_RUNNER }
roster: { Team_3->Runner_1, Team_3->Runner_14 }
runners: { Runner_1, Runner_14 }
```

Listing 6. Unexpected State Transition Discovered using Step Function

of results and scores and used the `until` function to check if such a state was reachable. This use of ALDB is essentially bounded model checking. The history function showed us how the system reached the desired state in Listing 7. This output shows a simple path of state transitions that can be manually examined for errors.

Once we had a correct model, one member of our team seeded three bugs in it and tasked another member to discover and fix the bugs. We will now discuss how ALDB was used to discover the bugs. Listing 8 shows the output from ALDB that helped us find the first seeded bug. After stepping through team registration operations, we see that no state can be reached where all runners have completed the race – a situation that obviously should be possible. The issue here was a scope problem. The `seq` (sequence) relation used for the list of results had been set to five in the model configuration, meaning that the results set could not contain more than five elements, hence the `#(results)=#(runners)` constraint could not be satisfied. The appropriate fix is to set the scope of `seq` to match the scope of the set of runners.

Listing 9 shows a series of states during team registration, from which we discovered the second seeded bug. After each step during the team registration, teams and rosters are updated but the set of runners is not updated correctly. The set of runners only contains the new added runners and not the previously registered runners. The cause of this bug was an omission in the operation for registering teams to ensure that the next state keeps all the runners of the existing state.

Listing 10 shows the ALDB output that helped us discover the third seeded bug. The teams that did not participate in the race were unexpectedly assigned scores. The fix was to ensure that scores are only assigned to registered teams in the operation that computes the scores.

It was sufficient to use ALDB to discover the bugs discussed above rather than the full Alloy Analzyer. ALDB has a familiar interface for programmers and gives quick results making it possible to fix these simple bugs right away while developing the model.

```
(aldb) load fast_feet.als
Reading model from fast_feet.als...done.
(aldb) break "#(results) > 0 and #(scores) > 0
    and mode=RESULTS"
(aldb) until

S4
----
mode: { RESULTS }
operation: { COMPUTE_SCORES }
results: { 0->Runner_0, 1->Runner_14 }
roster: { Team_3->Runner_0, Team_3->Runner_14 }
runners: { Runner_0, Runner_14 }
scores: { Team_3->1 }
teams: { Team_3 }

(aldb) history

S1 (-3)
---------
mode: { REGISTRATION }
operation: {  }
results: {  }
roster: {  }
runners: {  }
scores: {  }
teams: {  }

S2 (-2)
---------
mode: { REGISTRATION }
operation: { REGISTER_TEAM }
results: {  }
roster: { Team_3->Runner_0, Team_3->Runner_14 }
runners: { Runner_0, Runner_14 }
scores: {  }
teams: { Team_3 }

S3 (-1)
---------
mode: { RACE }
operation: { RUN_RACE }
results: { 0->Runner_0, 1->Runner_14 }
roster: { Team_3->Runner_0, Team_3->Runner_14 }
runners: { Runner_0, Runner_14 }
scores: {  }
teams: { Team_3 }
```

Listing 7. History Function shows Path to Reach End of Race

VII. Related Work

Various forms of simulation of transition systems are available in the tools supporting the formal specification languages TLA+ and B.

TLC is a model checker for a subclass of TLA+ specifications and it has a graphical user interface [11]. TLC's main usage modality is model checking: it generates a finite state space and checks for invariant violations within it. TLC can generate a random path of finite-length through the model [21], which is conceptually similar to systematically stepping through a model. TLC does not allow for incremental debugging and stepping to gradually build the state space. If

```
(aldb) load fast_feet.als
Reading model from fast_feet.als...done.
(aldb) step ["operation=REGISTER_TEAM"]

S2
----
mode: { REGISTRATION }
operation: { REGISTER_TEAM }
results: {  }
roster: { Team_0->Runner_0, Team_0->Runner_14 }
runners: { Runner_0, Runner_14 }
scores: {  }
teams: { Team_0 }

(aldb) step ["operation=REGISTER_TEAM"]

S3
----
roster: { Team_0->Runner_0, Team_0->Runner_14,
    Team_3->Runner_1, Team_3->Runner_2 }
runners: { Runner_0, Runner_1, Runner_14,
    Runner_2 }
teams: { Team_0, Team_3 }

(aldb) step ["operation=REGISTER_TEAM"]

S4
----
roster: { Team_0->Runner_0, Team_0->Runner_14,
    Team_2->Runner_3, Team_2->Runner_4, Team_3->
    Runner_1, Team_3->Runner_2 }
runners: { Runner_0, Runner_1, Runner_14,
    Runner_2, Runner_3, Runner_4 }
teams: { Team_0, Team_2, Team_3 }

(aldb) step ["operation=RACE and #(results)=#(
    runners)"]
Cannot perform step. Transition constraint is
    unsatisfiable.
```

Listing 8. Discovery of First Seeded Bug

```

S5
----
mode: { REGISTRATION }
operation: { REGISTER_TEAM }
results: {  }
roster: { Team_0->Runner_0, Team_0->Runner_14 }
runners: { Runner_0, Runner_14 }
scores: {  }
teams: { Team_0 }

(aldb) step

S6
----
roster: { Team_0->Runner_0, Team_0->Runner_14,
    Team_3->Runner_1, Team_3->Runner_2 }
runners: { Runner_1, Runner_2 }
teams: { Team_0, Team_3 }

(aldb) step

S7
----
roster: { Team_0->Runner_0, Team_0->Runner_14,
    Team_2->Runner_3, Team_2->Runner_4, Team_3->
    Runner_1, Team_3->Runner_2 }
runners: { Runner_3, Runner_4 }
teams: { Team_0, Team_2, Team_3 }
```

Listing 9. Step Function shows Runners are Missing

```
(aldb) load fast_feet.als
Reading model from fast_feet.als...done.
(aldb) break "mode=RESULTS"
(aldb) until

S4
----
mode: { RESULTS }
operation: { COMPUTE_SCORES }
results: {  }
roster: { Team_0->Runner_11, Team_0->Runner_13,
    Team_0->Runner_14 }
runners: { Runner_11, Runner_13, Runner_14 }
scores: { Team_1->0, Team_2->0, Team_3->0 }
teams: { Team_0 }
```

Listing 10. Last State in Path has Unexpected Scores

TLC finds an erroneous trace that violates model invariants, then users can step through that trace using the error-trace explorer, which is part of the TLA+ Toolbox [22].

The ProB Animator simulates the execution of specifications written in the B modelling language. It is available as both graphical and command-line tools. The animator allows users to step through a model's state space but we could not find a command similar to our `until` or constrained stepping commands. ProB has been extended to load Alloy models and translate them into B models for analysis with ProB [23], however this work is still experimental. Our tool allows users to write constraints directly in the Alloy language while debugging.

NuSMV [13] and NuXMV [14] provide lower-level languages for specifications, but they do provide some simulation capabilities. For example, in both tool sets, there is a command-line interactive tool called simulate. Within it the user can choose an initial state based on constraints, and generate a finite-length trace from the current state that satisfies constraints by using the next state temporal operator for different constraints over each step.

The Alloy Analyzer contains various method of customizing the layout of an instance. For example, Rayside et al. [24] describes using inferred properties of a model to customize its visualization, including views of projections of the instance to show individual states and their relations to illustrate steps in a dynamic execution. Sterling is a new visualizer for Alloy [25] under development. However, none of these methods are customized for transition systems or support the common code debugging steps such as breakpoints.

VIII. Conclusion

We have presented ALDB, a debugger for Alloy models of transition systems. Our debugger supports users in locally exploring concrete steps of their model and incrementally building up all the way to bounded model checking. It provides an alternative interface to the Alloy Analyzer that is customized for interactive analysis of transition systems. ALDB uses the existing Alloy solver via templates, which is an example for how additional functionality can be provided for creating and exploring Alloy models.

We envision a variety of uses for ALDB. First, it can be used in a *basic exploratory* model just after the user has initially composed a model. Many errors are likely to be found just walking around in the graph. Second, it can be used in a *what-if exploratory* mode, where the user follows particular paths via the constrained stepping function. For example, asking questions such as "what if this input occurs?" Third, the user might want to investigate *coverage* by looking at all the alternative paths. Fourth, ALDB can be used in a *diagnostic* mode where the user is either using the `until` command to do bounded model checking, or the user is reviewing a counterexample XML file previously found by the Alloy Analyzer. In these cases, the user is trying to diagnose the error in the model. However, a user case study is needed to determine the utility of ALDB in an Alloy modellers' workflow.

Acknowledgments

We thank Amin Bandali for his help with learning Alloy. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] D. C. Schmidt, "Guest editor's introduction: Model-driven engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] "OMG unified modeling language," http://www.omg.org/spec/UML/2.5/PDF/, 2015, [Online; accessed 16 May 2020].

[3] D. Jackson, *Software abstractions: logic, language, and analysis*, rev. ed ed. MIT Press, 2012.

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

[5] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, 2010.

[6] P. Zave, "Using lightweight modeling to understand Chord," in *ACM SIGCOMM Computer Communication Review*, vol. 2, no. 42, 4 2012, pp. 50–57.

[7] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How Amazon web services uses formal methods," *Communications of the ACM*, vol. 58, no. 4, 2015.

[8] N. Macedo, J. Brunel, D. Chemouil, A. Cunha, and D. Kuperberg, "Lightweight specification and analysis of dynamic systems with rich configurations," in *Foundations of Software Engineering (FSE)*. ACM, 2016, pp. 373–383.

[9] S. Farheen, N. A. Day, A. Vakili, and A. Abbassi, "Transitive-closure-based model checking in Alloy," *Journal of Software and Systems Modelling*, vol. 19, p. 721–740, 2020.

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," ser. Advances in Computers. Elsevier, 2003, vol. 58, no. Supplement C, pp. 117 – 148.

[11] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Correct Hardware Design and Verification Methods (CHARME)*, ser. Lecture Notes In Computer Science, no. 1703. Springer, 1999, pp. 54–66.

[12] M. Leuschel and M. Butler, "ProB: A Model Checker for B," in *FME 2003: Formal Methods*, ser. Lecture Notes In Computer Science. Springer, 2003, vol. 2805, pp. 855–874.

[13] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren *et al.*, *NuSMV 2.6 User Manual*, 2010 (accessed May 11, 2020). [Online]. Available: http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf

[14] M. Bozzano, R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio *et al.*, *nuXmv 2.0.0 User Manual*, 2019 (accessed May 11, 2020). [Online]. Available: https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf

[15] D. Jackson and A. Fekete, "Lightweight analysis of object interactions," in *Theoretical Aspects of Computer Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 492–513.

[16] A. Sullivan, K. Wang, and S. Khurshid, "Evaluating State Modeling Techniques in Alloy," *SQAMIA 2017 - Proc. 6th Work. Softw. Qual. Anal. Monit. Improv. Appl.*, pp. 11–13, 2017.

[17] "River crossing problem in Alloy," Accessed 11 May 2020. [Online]. Available: https://github.com/AlloyTools/org.alloytools.alloy/blob/master/org.alloytools.alloy.extra/extra/models/examples/tutorial/farmer.als

[18] "The dot language," Accessed 11 May 2020. [Online]. Available: https://www.graphviz.org/doc/info/lang.html

[19] "Graphviz - graph visualization software," Accessed 11 May 2020. [Online]. Available: http://graphviz.org

[20] N. A. Day, "University of Waterloo SE212 Assignment 7," 2018.

[21] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu, "Specifying and verifying systems with TLA+," *Proc. 10th Work. ACM SIGOPS European Workshop – EW10*, p. 45, 2002.

[22] *TLA+ Toolbox User's Guide*, Accessed May 16 2020. [Online]. Available: https://tla.msr-inria.inria.fr/tlatoolbox/doc/contents.html

[23] *The ProB Animator and Model Checker*, 2020 (Accessed May 11 2020). [Online]. Available: https://www3.hhu.de/stups/prob/index.php/Main_Page

[24] D. Rayside, F. S. Chang, G. Dennis, R. Seater, and D. Jackson, "Automatic visualization of relational logic models," *Electronic Communications of the EASST*, vol. 7, 2007.

[25] T. Dyer, "Sterling," Accessed May 22 2020. [Online]. Available: https://sterling-js.github.io