Live Programming for Finite Model Finders

Allison Sullivan
University of Texas at Arlington
Arlington, TX USA
allison.sullivan@uta.edu

***Abstract*—Finite model finders give users the ability to specify properties of a system in mathematical logic and then automatically find concrete examples, called solutions, that satisfy the properties. These solutions are often viewed as a key benefit of model finders, as they create an exploratory environment for developers to engage with their model. In practice, users find less benefit from these solutions than expected. For years, researchers believed that the problem was that too many solutions are produced. However, a recent user study found that users actually prefer enumerating a broad set of solutions. Inspired by a recent user study on Alloy, a modeling language backed by a finite model finder, we believe that the issue is that solutions are too removed from the logical constraints that generate them to help users build an understanding of the constraints themselves. In this paper, we outline a proof-of-concept for live programming of Alloy models in which writing the model and exploring solutions are intertwined. We highlight how this development environment enables more productive feedback loops between the developer, the model and the solutions.**

***Index Terms*—Relational Model Finders, Live Programming**

I. Introduction

The expressibility of software modeling languages has greatly expanded and the automated analysis over these models has reached a point of efficiency where we can reason about real world systems [34], [9], [22], [68], [13], [43], [12], [64]. However, software models are not widely adopted because there is still the bottleneck of writing the specification. While a formal methods expert can write the specifications, there is no guarantee that the expert will understand the domain area well enough to accurately capture the intricacies of the system. Ideally, the software architect would write the specifications.

Unfortunately, modeling languages are notoriously difficult to learn, which is compounded by software modeling toolsets that lag behind the state-of-the-art for integrated development environments (IDE) [2], [27], [52]. Most notably, modeling toolsets lack quality feedback. For instance, when a model is executed, the main result presented to the user is often a simple boolean result: either the analysis over the model is satisfiable or unsatisfiable. This does not give users enough context to answer the question "did I write my model correctly?" Additionally, a number of features that aide in composition, such as code completion, are almost universally absent.

To address the feedback issue, there has been a rise in the creation of finite model finders, where users write a software model of their system's design and the finite model finder then produces concrete examples of behavior allowed by the model [25], [37], [30], [6], [16], [18], [59], [50]. In this paper, we will refer to the output of finite model finders as *solutions* and the logical constraints that are executed as *software models*. Clearly, in place of a boolean result, these solutions provide more context to the user about the behavior of their model. Furthermore, besides helping validate software designs [35], [42], [68], [65], [11], these solutions have been used to test and debug code [17], [36], to repair program states [51], [67] and to provide security analysis of systems [60], [1], [3].

While these solutions are often mentioned as one of the core benefits of finite model finders, users have found them to be less helpful in practice [69], [33], [14]. A long held belief is that there are too many solutions, which can overwhelm the user. As a result, there is a whole body of work [53], [55], [47], [46], [54], [41] dedicated to trying to reduce the number of solutions, often by creating additional criteria a solution must adhere too, such as minimality [41]. However, a recent user study found that users often abandon these tailored enumerations in favor of the default enumeration [14], implying that users want more solutions to explore to better understand their model and thus their system. Furthermore, another user study with a mix of novice and expert users found that all users struggled with inspecting solutions and in turn, refining the model based on the solutions. Therefore, as a community we have been solving the *wrong* problem.

Our theory is that the solutions, while visually approachable, are too divorced from the logical constraints that form the model to actually help the user follow up on the question "did I write my model correctly?" Finite model finders do not convey *why* solutions are generated, only that they *are* generated. As a result, finite model finders place the burden on the user to determine how the abstract constraints they are writing ultimately influence the concrete solutions. *Our vision* is that a live programming environment, which interweaves writing the model and evaluating the model, is the answer to turning the solutions enumerated by finite model finders into constructive feedback, which in turn, can make software modeling approachable to the average software architect.

II. Motivating Example

In this paper, we illustrate a live programming environment for the modeling language Alloy [25]. Alloy enables the specification of both structural and behavioral properties in the form of relational, first order and linear temporal logic constraints. Alloy is packaged in an automated analysis engine, the Analyzer, which utilizes Pardinus [31], a temporal relational model finder, to enumerate solutions.

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00016

(a)

```
1. one sig Queue { var head : lone Node }
2. var sig Node { var link : lone Node }
3. fact WellFormed {
4.    always all n : Node | n !in n.^link
5.    always all n : Node | n in Queue.head.*link
6. }
7. pred dequeue {
8.    head' = head.link
9.    all n : Queue.head.^link | n.link = n.link'
10. }
11. run dequeue for 3
```

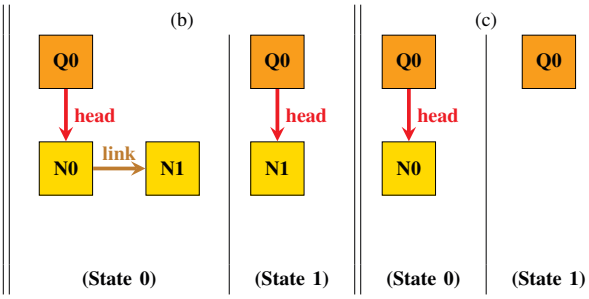

Fig. 1. Alloy Model of a Queue Data Structure

To highlight the limitations of the current model development process, consider the following model of a queue data structure seen in Figure 1 (a). To start, the user would write the model in the text editor portion of the Analyzer. Signature blocks introduce atoms and their relationships (lines 1 - 2). The keyword `var` indicates which portions of the model can change between states. For example, line 1 introduces a named set `Queue` and defines a mutable relation `head` that expresses that each `Queue` atom has zero or one (`lone`) node at the start of the queue. Users can write named formulas in fact (`fact`) blocks, which have to be true for any solution found, or predicate (`pred`) blocks, which have to be true for a solution if the predicate is invoked. To outline well-formed queues, the user specifies no node is reachable from itself (line 4) and all nodes are in the queue (line 5). To outline `dequeue`, the user specifies the new head of the queue is the second node in the queue (line 8) and there are no other changes to the order of the nodes in the queue (line 9).

When the user is ready to check their constraints, the user executes the run command on line 11. The Analyzer will then use Pardinus to search for all satisfying solutions such that `dequeue` and `WellFormed` are true using up to 3 `Queue` atoms and 3 `Node` atoms. These solutions are rendered graphically in a separate pop up window, where the user can iterate over the solutions one by one. To illustrate, Figure 1 (b) and (c) display the first two solutions the Analyzer finds. If the user encounters a malformed solution, the user now knows that their model allows for behavior she intended to prevent. The user then returns to the text editor and tries to update their current constraints to prevent the malformed solution. However, as the user makes edits, the only way to know the impact on the solutions is to re-execute the command and re-start the enumeration and inspection process.

In this workflow, the burden is on the user to mentally visualize the impact abstract constraints will have on the concrete shape of the solutions, as editing the model and enumerating solutions are distinct tasks. However, if we could integrate writing constraints with the construction of the solutions, then users could witness the impact of their constraints in real time.

III. Live Programming for Finite Model Finders

In this section, we outline our proof-of-concept of a live programming development environment for Alloy. First, we

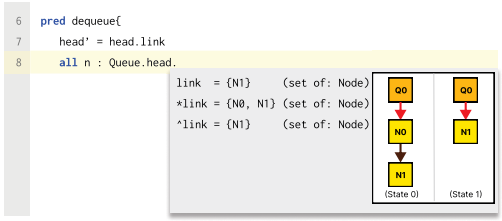

Fig. 2. Prototype of Suggestion Box for Formula Completion

introduce suggestion boxes that provide formula completion suggestions with concrete illustrations of how each of the suggested formulas would behave. Second, we highlight two different development views that a user can toggle between: (1) a dynamic Enumeration View, which presents the broad impact across all possible solutions and (2) a Focus View, which presents the narrow impact on a single solution

A. Formula Completion Suggestions

Code completion, such as presenting valid API calls to make on an object, is a common feature in many integrated development environments (IDE) that is a lightweight intervention to help users efficiently compose their programs. The equivalent concept for a modeling language would be formula completion where we distill for the user valid extensions of the formula they are actively writing. We have begun developing a series of rules for formula completion based on using grammar and type rules to produce valid extensions. For instance, a relational join "`a.b`" will default to the empty set if there is a type mismatch between `a` and `b`, e.g. `link.head` which joins types `(Node×Node).(List×Node)` will always be empty while `head.link`, which joins types `(List×Node).(Node×Node)` will not. Therefore, if the user starts typing "`link.`" we do not want to suggest `head` as a continuation, but we would want to suggest `link` if the user is typing "`head.`". In addition, within Alloy we have a lightweight mechanic for constraint checking through the Evaluator, a toolset carried over from KodKod [58]. This allows us to provide more context by annotating our suggestions with their actual value over a solution of the user's choosing. We combine completion suggestions with projected values into the concept of a suggestion box.

To illustrate, consider the user trying to capture "all nodes in the queue except for the first node" – which is the domain

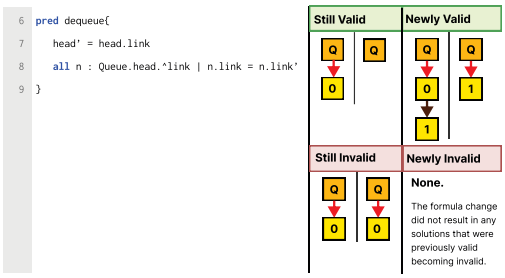

Fig. 3. Prototype of Enumeration View

of the quantified formula from `dequeue`. On their own, the user may struggle to determine the order to place the relational joins and whether to use reflexive transitive closure ('$*$') or transitive closure ('$\hat{}$'). Figure 2 highlights how our suggestion box can aid the user in composing this domain. As the user types "`Queue.head.`," the only built in set from the base model that does not produce an empty set is `link`. Therefore, the suggestion box populates with `link` and common extensions of `link`. With the added context of the value of these extensions over a solution, the user can determine that "`^link`" likely matches their intention.

In our final live programming IDE, we plan to enable the user to swap the solution, which will update the projected values. For instance, the user could update the solution in Figure 2 to a queue with 3 nodes, and the difference between selecting "`link`" and "`^link`" would be revealed. In addition, we plan to explore how to make more complex suggestions by generating our own set of common formula templates. RexGen, a relational algebra expression generator [62], can produce formulas to further populate our suggestion boxes. However, in practice, RexGen produces too many formulas to use as is. Therefore, we plan to use a recently published corpus of all Alloy models on Github [21] to distill common templates for formula structures, which we will combine with RexGen to produce a small collection of complex suggestions. These common formula templates can also be utilized for other research, including the new but active field of automated repair for Alloy [61], [7], [10], [23], [71].

B. Enumeration View

The main goal of live programming is for users to get instant feedback while changing their program. For an Alloy model, this feedback is the collection of *all* possible solutions enabled by their model. Therefore, as the user adds constraints to the model, the user is constantly shifting the boundary of valid and invalid solutions. To bring this changing boundary to the forefront, our Enumeration View actively display two sets of solutions side by side with the text editor that update every time the user's edits create a model that compiles: a set of valid and a set of invalid solutions. This is inspired by a recent user study that found that novice users better understood pre-written Alloy constraints when presented with a combination of valid and invalid solutions [19]. The user study, while a promising result, used hand selected solutions. Our live programming environment will instead generate these solutions automatically. Since we are working with logical constraints, we can go beyond simply asking "what solutions are valid?" and "what solutions are invalid?" Instead, we can present four categories of solutions to the user: two valid categories – (a) solutions that remained valid and (b) solutions that are now valid, and two invalid categories – (c) solutions that stayed invalid and (d) solutions that became invalid. This is possible since we can use Alloy itself to compare and contrast two formulas. For instance, for two iterations of a formula (`A` and `A'`), we can execute the following command to get category (a) "`run {A and A'}`" and "`run {!A and A'}`" to get category (b).

To illustrate, Figure 3 shows the Enumeration View that would be produced if the user went from a faulty `dequeue` predicate in which the user used reflexive transitive closure to specify the domain of the quantified formula ("`Quene.head.*link`" to the correct `dequeue` predicate in Figure 1 (a). Since each category would be maintained by a command, the user can enumerate solutions within each category. However, to avoid too high of a computational overhead, we will first use constraint checking to see if the currently displayed solution for a given category is still representative. If not, then we will generate a new solution.

C. Focus View

While the output of a model is the collection of all solutions, as a user builds a model, it is not uncommon for the user to have a handful of key solutions in mind that the user is expecting to confirm that their model should generate or prevent. In fact, the adhoc practice of outlining a solution within a predicate to reason over individually was distilled into a unit testing framework [55]. Leaning into this practice, the Focus View allows the user to see the impact on a single solution as they write their constraints. Within the Focus View, the user selects a solution and labels whether the solution is expected to be valid or invalid for their model. As the user writes, we constantly display the solution and it's current behavior (valid or invalid). As a result, the user is actively aware of any changes that result in discrepancies.

Furthermore, we use Alloy to provide debugging information to aid the user in understanding the current boundary between the solution the user expects to be valid (or invalid) and the behavior the current model prevents (or enables). The debugging information comes in two forms. First, for a solution s whose behavior violates its expectation, we present the closest valid (or invalid) solution to s. We can generate the closest behavioral solution using a Partial-Max-SAT solver. For instance, for a solution that is expected to be valid but is not, we can make the hard constraints those enforced by the model and the soft constraints those outlining s. To provide even more context, we can also automatically decompose the difference between s and the closest behavioral solution by presenting a breakdown of all the formulas in the model in which the two solutions produce different truth values. This evaluation breakdown can be done efficiently by turning the Evaluator from a black-box toolset into a white-box toolset

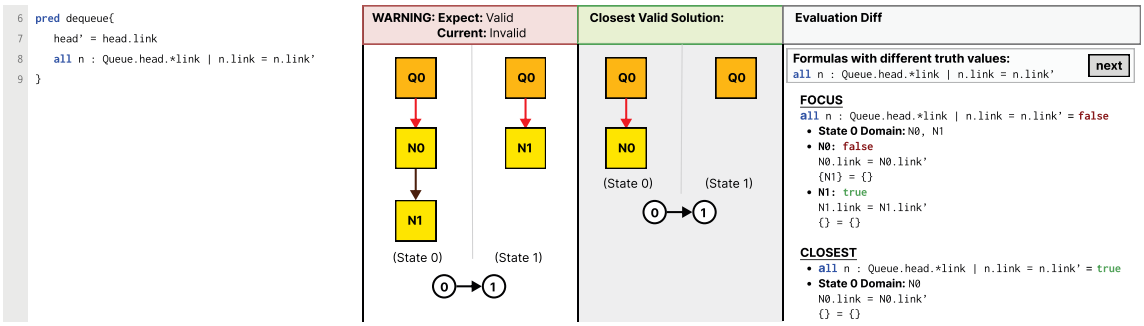

Fig. 4. Prototype of Focus View

that presents the intermediate evaluations discovered along the way to producing the final truth result.

To illustrate, Figure 4 shows the Focus View in action. The first pane is the text editor where the user just added the faulty quantified formula for `dequeue` with the incorrect domain mentioned in the Enumeration View example. In the second pane, the user has the solution from Figure 1 (b) in view. This solution was valid when the user only had the formula on line 7 written; however, after adding the formula on line 8, the solution is now prevented. Therefore, the third pane displays the current closest valid solution, which reveals to the user that their formula results in valid behavior when the queue starts with one node but not when the queue starts with two nodes. The final pane shows a breakdown of the faulty quantified formula across the two solutions. This breakdown reveals to the user that their current model incorrectly includes the first node in the quantified domain.

IV. Future Plans

To bring our live programming environment from a proof-of-concept to a reality, there are two high level problems to address throughout. First, while we have initial prototypes, we will need to refine the design of our interfaces to ensure a smooth transfer of knowledge without overwhelming the user with too much information. In addition to how we present information, there is also a question of what information is best to present. For instance, for the starting solutions in our Enumeration View – Should the solutions be as close to one another as possible across categories? Should we present maximal solutions to convey more context or is this too cluttered? Given the usability focus of this work, it is important that these decisions be vetted through active engagement with novice and expert end users. To that end, we plan to conduct multiple user studies as we build out our live programming framework. We anticipate using students as novice users and active members of the Alloy discourse group as expert users.

Second, we need to produce responsive implementations of these interfaces. Across our different live programming designs, we often take advantage of Alloy itself to add rich information, such the value of suggestions or generating boundary solutions. However, Alloy's runtime is not nominal. Therefore, we need to utilize existing optimizations and develop our own to ensure that the tandem presentation of solutions does not make the development environment sluggish. For instance, our final backend implementation of the Enumeration View will need to be carefully designed. There are existing bodies of work for incremental analysis of Alloy models, which can help ease the runtime overhead when we need to explicitly search for new solutions [4], [70], [26], [63]. In addition, we can look for opportunities to use constraint checking to determine the impact of changes in order to delay, or even avoid, invoking a SAT solver for all interfaces. Along the way, we also expect to port the Analyzer codebase from Java to facilitate better implementations of these visual interfaces.

V. Related Work

Live programming is an active research field [56], [38], [28], [39], [32], [5], [15], [20], [66], [45], [8], [24], [44] that has been applied to variety of imperative, declarative and functional languages. To the best of our knowledge, live programming for modeling languages has only been explored for finite state machines [57], [49]. Our suggestion boxes are in the same spirit as projection boxes for Python, which display the live value of variables [29]. The research most closely related to our work for Alloy is (1) Amalgam [40], which uses provenance chains to inform the user why a specific tuple does or does not appear in a solution and (2) abstract instances [48], which decomposes a solution into the parts present to satisfy the facts versus the parts present to satisfy predicates. Both of these aim to help the user understand why a solution was generated, and are complementary work that could be incorporated into a live programming environment to provide on-the-fly explanations of solutions.

VI. Conclusion

Currently, the disconnect between the logical constraints that form a model and the graphical solutions that get produced by the model prevent users from utilizing solutions to better understand the logical constraints themselves. This paper presents the concept of a live programming environment for Alloy that interweaves writing and evaluating a model. Our proof-of-concept highlights how the interconnection of these two artifacts enables users to fully address and follow up on the question "did I write my model correctly?"

References

[1] Akhawe, D., Barth, A., Lam, P.E., Mitchell, J., Song, D.: Towards a formal foundation of web security. In: 2010 23rd IEEE Computer Security Foundations Symposium. pp. 290–304 (2010)

[2] Almstrum, V.L., Dean, C.N., Goelman, D., Hilburn, T.B., Smith, J.: Support for teaching formal methods. In: Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education. p. 71–88. ITiCSE-WGR '00, Association for Computing Machinery, New York, NY, USA (2001)

[3] Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the Android permission system. Formal Asp. Comput. (2018)

[4] Bagheri, H., Malek, S.: Titanium: efficient analysis of evolving alloy specifications. In: Proceedings of the 2016 24th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE). pp. 27–38 (2016)

[5] Black, A.P., Nierstrasz, O., Ducasse, S., Pollet, D.: Pharo by example. Lulu. com (2010)

[6] Blanchette, J., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. vol. 6172, pp. 131–146 (07 2010)

[7] Brida, S.G., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.F.: Bounded exhaustive search of alloy specification repairs. In: ICSE (2021)

[8] Burnett, M.M., Atwood, J.W., Welch, Z.T.: Implementing level 4 liveness in declarative visual programming languages. In: Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No. 98TB100254). pp. 126–133. IEEE (1998)

[9] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Havelund, K., Holzmann, G., Joshi, R. (eds.) NASA Formal Methods. pp. 3–11. Springer International Publishing (2015)

[10] Cerqueira, J., Cunha, A., Macedo, N.: Timely specification repair for alloy 6. In: Software Engineering and Formal Methods. pp. 288–303 (2022)

[11] Chong, N., Sorensen, T., Wickerson, J.: The semantics of transactions and weak memory in x86, Power, ARM, and C++. SIGPLAN Not. **53**(4), 211–225 (2018)

[12] Cook, B.: Formal reasoning about the security of amazon web services. In: Computer Aided Verification. pp. 38–47. Springer International Publishing (2018)

[13] Cunha, A., Macedo, N.: Validating the hybrid ertms/etcs level 3 concept with electrum **22**(3), 281–296 (jun 2020)

[14] Danas, N., Nelson, T., Harrison, L., Krishnamurthi, S., Dougherty, D.J.: User studies of principled model finder output. In: SEFM (2017)

[15] DeLine, R., Fisher, D., Chandramouli, B., Goldstein, J., Barnett, M., Terwilliger, J.F., Wernsing, J.: Tempe: Live scripting for live data. In: VL/HCC. vol. 15, pp. 137–141 (2015)

[16] Dennis, G., Chang, F.S.H., Jackson, D.: Modular verification of code with sat. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis. p. 109–120. ISSTA '06 (2006)

[17] Dini, N., Yelen, C., Alrmaih, Z., Kulkarni, A., Khurshid, S.: In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing. pp. 1934–1943 (2018)

[18] Dolby, J., Vaziri, M., Tip, F.: Finding bugs efficiently with a sat solver. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. p. 195–204. ESEC-FSE '07, Association for Computing Machinery, New York, NY, USA (2007)

[19] Dyer, T., Nelson, T., Fisler, K., Krishnamurthi, S.: Applying cognitive principles to model-finding output: the positive value of negative information. Proceedings of the ACM on Programming Languages **6**(OOPSLA1), 1–29 (2022)

[20] Edwards, J.: Subtext: uncovering the simplicity of programming. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 505–518 (2005)

[21] Eid, E., Day, N.A.: Static profiling alloy models. IEEE Transactions on Software Engineering pp. 1–1 (2022)

[22] Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., Millstein, T.: A general approach to network configuration analysis. In: Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation. p. 469–483. NSDI'15, USA (2015)

[23] Gutiérrez Brida, S., Regis, G., Zheng, G., Bagheri, H., Nguyen, T., Aguirre, N., Frias, M.: ICEBAR: Feedback-Driven Iterative Repair of Alloy Specifications (2023)

[24] Ingalls, D., Palacz, K., Uhler, S., Taivalsaari, A., Mikkonen, T.: The lively kernel a self-supporting system on a web page. In: Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers. pp. 31–50. Springer (2008)

[25] Jackson, D.: Alloy: A lightweight object modelling notation. IEEE Transactions on Software Engineering (TSE) **11**, 256–290 (2002)

[26] Jovanovic, A., Sullivan, A.: Reach: Refining alloy scenarios by size. In: ISSRE (2022)

[27] Krishnamurthi, S., Nelson, T.: The human in formal methods. In: Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11800, pp. 3–10. Springer (2019)

[28] Kubelka, J., Robbes, R., Bergel, A.: The road to live programming: insights from the practice. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1090–1101 (2018)

[29] Lerner, S.: Projection boxes: On-the-fly reconfigurable visualization for live programming. In: Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems. pp. 1–7 (2020)

[30] Leuschel, M., Butler, M.: Prob: A model checker for b. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods. pp. 855–874. Berlin, Heidelberg (2003)

[31] Macedo, N., Brunel, J., Chemouil, D., Cunha, A.: Pardinus: A temporal relational model finder. J. Autom. Reason. **66**(4), 861–904 (nov 2022)

[32] Maloney, J.H., Smith, R.B.: Directness and liveness in the morphic user interface construction environment. In: Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology. p. 21–28. UIST '95, New York, NY, USA (1995)

[33] Mansoor, N., Bagheri, H., Kang, E., Sharif., B.: An empirical study assessing software modeling in alloy. In: International Conference on Formal Methods in Software Engineering. p. To Appear (2023)

[34] Mansoor, N., Saddler, J.A., Silva, B., Bagheri, H., Cohen, M.B., Farritor, S.: Modeling and testing a family of surgical robots: An experience report. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 785–790. ESEC/FSE 2018, Association for Computing Machinery, New York, NY, USA (2018)

[35] Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: MODELS (2011)

[36] Marinov, D., Khurshid, S.: Testera: a novel framework for automated testing of java programs. In: Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001). pp. 22–31 (2001)

[37] McCune, W.: Mace 2.0 reference manual and guide. (6 2001)

[38] McDirmid, S.: Living it up with a live programming language. ACM SIGPLAN Notices **42**(10), 623–638 (2007)

[39] McDirmid, S.: Usable live programming. In: Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software. pp. 53–62 (2013)

[40] Nelson, T., Danas, N., Dougherty, D.J., Krishnamurthi, S.: The power of "why" and "why not": Enriching scenario exploration with provenance. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE). pp. 106–116 (2017)

[41] Nelson, T., Saghafi, S., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Aluminum: Principled scenario exploration through minimality. In: ICSE (2013)

[42] Nelson, T., Barratt, C., Dougherty, D.J., Fisler, K., Krishnamurthi, S.: The Margrave tool for firewall analysis. In: Proceedings of the 24th International Conference on Large Installation System Administration (LISA). pp. 1–8 (2010)

[43] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How amazon web services uses formal methods. Commun. ACM **58**, 66–73 (Mar 2015)

[44] Omar, C., Moon, D., Blinn, A., Voysey, I., Collins, N., Chugh, R.: Filling typed holes with live guis. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 511–525 (2021)

[45] Omar, C., Voysey, I., Chugh, R., Hammer, M.A.: Live functional programming with typed holes. Proceedings of the ACM on Programming Languages **3**(POPL), 1–32 (2019)

[46] Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE). pp. 908–919 (2016)

[47] Porncharoenwase, S., Nelson, T., Krishnamurthi, S.: CompoSAT: Specification-guided coverage for model finding. In: FM (2018)

[48] Ringert, J.O., Sullivan, A.: Abstract alloy instances. In: Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings. pp. 364–382 (2023)

[49] van Rozen, R., van der Storm, T.: Toward live domain-specific languages: From text differencing to adapting models at run time. Software & Systems Modeling **18**, 195–212 (2019)

[50] Samimi, H., Aung, E.D., Millstein, T.: Falling back on executable specifications. In: Proceedings of the 24th European Conference on Object-Oriented Programming. p. 552–576. ECOOP'10 (2010)

[51] Samimi, H., Aung, E.D., Millstein, T.D.: Falling back on executable specifications. In: ECOOP. pp. 552–576 (2010)

[52] Siegel, A., Santomauro, M., Dyer, T., Nelson, T., Krishnamurthi, S.: Prototyping formal methods tools: a protocol analysis case study. In: Protocols, Strands, and Logic: Essays Dedicated to Joshua Guttman on the Occasion of his 66.66 th Birthday. pp. 394–413. Springer (2021)

[53] Sullivan, A.: Hawkeye: User-guided enumeration of scenarios. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE). pp. 569–578. IEEE (2021)

[54] Sullivan, A., Marinov, D., Khurshid, S.: Solution enumeration abstraction - A modeling idiom to enhance a lightweight formal method. In: The International Conference on Formal Engineering Methods (ICFEM). pp. 336–352 (2019)

[55] Sullivan, A., Wang, K., Zaeem, R.N., Khurshid, S.: Automated test generation and mutation testing for Alloy. In: 2017 IEEE Conference on Software Testing, Validation and Verification (ICST). pp. 264–275 (2017)

[56] Tanimoto, S.L.: A perspective on the evolution of live programming. In: 2013 1st International Workshop on Live Programming (LIVE). pp. 31–34. IEEE (2013)

[57] Tikhonova, U., Stoel, J., Van Der Storm, T., Degueule, T.: Constraint-based run-time state migration for live modeling. In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering. pp. 108–120 (2018)

[58] Torlak, E., Jackson, D.: Kodkod: A relational model finder. In: TACAS (2007)

[59] Torlak, E., Vaziri, M., Dolby, J.: Memsat: Checking axiomatic specifications of memory models. In: Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 341–350. PLDI '10 (2010)

[60] Trippel, C., Lustig, D., Martonosi, M.: Security verification via automatic hardware-aware exploit synthesis: The CheckMate approach. IEEE Micro (2019)

[61] Wang, K., Sullivan, A., Khurshid, S.: Automated model repair for Alloy. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE). pp. 577–588 (2018)

[62] Wang, K., Sullivan, A., Koukoutos, M., Marinov, D., Khurshid, S.: Systematic generation of non-equivalent expressions for relational algebra. In: 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ). pp. 105–120 (2018)

[63] Wang, W., Wang, K., Gligoric, M., Khurshid, S.: Incremental analysis of evolving alloy models. In: Vojnar, T., Zhang, L. (eds.) TACAS (2019)

[64] Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. SIGPLAN Not. **52**(1), 190–204 (jan 2017)

[65] Wickerson, J., Batty, M., Sorensen, T., Constantinides, G.A.: Automatically comparing memory consistency models. In: POPL (2017)

[66] Wilcox, E.M., Atwood, J.W., Burnett, M.M., Cadiz, J.J., Cook, C.R.: Does continuous visual feedback aid debugging in direct-manipulation programming systems? In: Proceedings of the ACM SIGCHI Conference on Human factors in computing systems. pp. 258–265 (1997)

[67] Zaeem, R.N., Khurshid, S.: Contract-based data structure repair using Alloy. In: ECOOP. pp. 577–598 (2010)

[68] Zave, P.: Using lightweight modeling to understand chord. SIGCOMM Comput. Commun. Rev. **42**, 49–57 (2012)

[69] Zave, P.: A practical comparison of alloy and spin. Formal Aspects of Computing **27**, 239–253 (2015)

[70] Zheng, G., Bagheri, H., Rothermel, G., Wang, J.: Platinum: Reusing constraint solutions in bounded analysis of relational logic. In: FASE (2020)

[71] Zheng, G., Nguyen, T., Brida, S.G., Regis, G., Aguirre, N., Frias, M.F., Bagheri, H.: Atr: Template-based repair for alloy specifications. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis. p. 666–677. ISSTA 2022 (2022)