Mining Temporal Properties of Data Invariants

Caroline Lemieux
Computer Science
University of British Columbia
Vancouver, BC, Canada

***Abstract*—System specifications are important in maintaining program correctness, detecting bugs, understanding systems and guiding test case generation. Often, these specifications are not explicitly written by developers. If we want to use them for analysis, we need to obtain them through other methods; for example, by mining them out of program behavior. Several tools exist to mine data invariants and temporal properties from program traces, but few examine the temporal relationships *between* data invariants. An example of this kind of relationship would be "the return value of the method `isFull`? is false until the field `size` reaches the value `capacity`". We propose a *data-temporal* property miner, Quarry, which mines Linear Temporal Logic (LTL) relations of arbitrary length and complexity between Daikon-style data invariants. We infer data invariants from systems using Daikon, recompose these data invariants into sequences, and mine temporal properties over these sequences. Our preliminary results suggest that this method may recover important system properties.**

I. Introduction

Data invariants play an important role in maintaining program correctness and aiding system comprehension. However, they are rarely specified. An alternative to manual specification is inference with a tool such as Daikon [6], which instruments a program and dynamically infers data invariants by observing multiple program runs. For example, a `Queue` class may include fields `size` and `capacity` to represent the current and maximum size of the queue, respectively. By observing the concrete values of these fields from multiple program runs (often from test cases), Daikon would infer an invariant `size` <= `capacity`.

However, data invariants fail to capture program function over an important dimension: time. Understanding the temporal relationships between events such as method calls is often important; for example, we likely want the method call `open_file`() to always be followed by the method call `close_file`().

We introduce a tool called Quarry to help to bridge the gap in understanding the interaction between data and time. In particular, Quarry discovers temporal relationships between data invariants. Returning to our `Queue` example; in addition to `size` and `capacity`, our `Queue` may also have an `isFull` flag. Daikon can infer a data invariant like $(\texttt{isFull} == \texttt{true}) \iff (\texttt{size} == \texttt{capacity})$ at some program points, but deeper comprehension of the relationship requires us to talk about the relationship between these invariants through time. Mining the temporal property $x \text{ U } y$, "x holds until y becomes true" should reveal

$$(\texttt{isFull} == \texttt{false}) \text{ U } (\texttt{size} == \texttt{capacity}),$$

read as "`isFull` is `false` until `size` is equal to `capacity`". In fact, this "data-temporal" invariant reflects an important correctness condition for the queue: it is not flagged as full until it reaches capacity. We believe that data-temporal properties will be useful to developers in developing test cases and understanding systems, and could be incorporated into bug detection or system modeling tools.

II. Related Work

Since finding a satisfying assignment of events to a temporal relation is NP-complete in general [9], most temporal property mining tools focus on extracting small patterns and composing them. Perracotta [21] mines response patterns ("x is always followed by y"); Javert [8] mines alternating ($(xy)^*$) and resource allocation ($(xz^+y)^*$) patterns. Both combine these small patterns into larger properties. Reger et al. use a similar pattern-composition technique in [12] and extend it to accomodate imperfect traces in [11]. Many other tools base their inference on basic properties similar to these ([4], [9], [10], [18], [20]). A general LTL formula checker is developed in [19]. Texada [17] mines general LTL formulae from program execution traces. The data invariant detection ideas developed in Daikon [6], [7] have been adopted to create several tools for program comprehension and test generation [3], [13]–[15].

Lorenzoli et al. propose merging temporal event relations and data invariants into Extended Finite State Machine models [16]. Our work differs from this approach in that we explicitly discover temporal relationships *between* data invariants instead of classifying different temporal relationships between method calls depending on which data invariants hold. Such relationships are discovered by the tools presented in [1], [2], which focus on application to embedded software and hardware systems. These tools can only mine a small subset of LTL patterns, while Quarry supports any LTL pattern. All these tools are important precedents for the utility of data-temporal invariants in broader contexts, such as in aiding test case generation.

III. Approach

To build Quarry, we combine two existing tools. To detect data invariants, we use Daikon [6], a dynamic tool which infers likely data invariants by instrumenting the target program and tracking variable state. Daikon mines invariants from templates including inequality, implication and containment between variables, which can be return values, global variables, or fields [7]. To infer temporal relationships between data invariants, we use Texada [17], a general Linear Temporal Logic (LTL) property miner we previously developed.

LTL supports the usual boolean logic operators as well as temporal operators, such as: $\text{X}\, p$, "p is true at the next point in time"; $\text{G}\, p$, "p holds for this and all future points in time"; $\text{F}\, p$, "p holds at some point in the present or future"; and $p \text{ U } q$, "p is true up to the first time q becomes true." These operators allow a rich expression of temporal properties [5]. Since many users may not be familiar with LTL, we provide

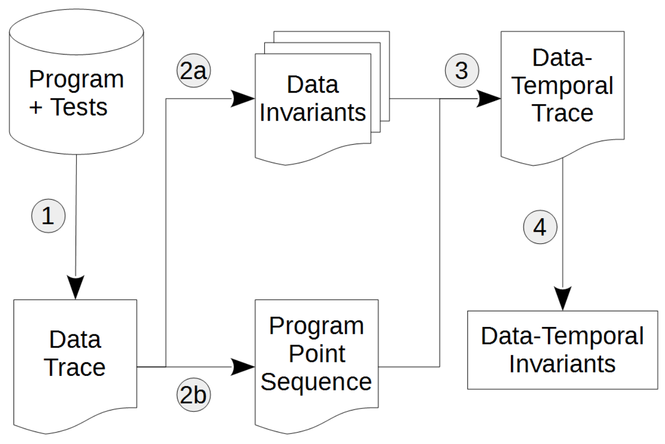

Fig. 1. The Quarry process. Data trace creation in step 1 is performed by a Daikon front-end, and data invariant inference in step 2a is performed by Daikon. Data-temporal invariant inference in step 4 is performed by Texada.

commonly-used temporal invariant templates with Texada. We now outline the Quarry process in Figure 1.

We first generate data traces by applying a Daikon front-end (such as Chicory or Celeriac), to a program and its test suite. These tools automatically instrument programs to produce data trace files (step 1 in Figure 1). We then run Daikon on the data trace to obtain data invariants on program points (step 2a). Since Daikon removes the temporal sequence context of program points, we retrieve the sequence of program points from the data trace (step 2b). By replacing each program point with an event consisting of the invariants holding at that point, we produce the multi-propositional data-temporal event trace (step 3), which is passed as input to Texada to infer data-temporal invariants (step 4).

We extended Texada to mine temporal invariants in multi-propositional traces (traces where several events occur at each time step). This extension is vital because unlike method calls or other temporal events, data invariants may persist through time, even as other data invariants change. So, we need to state all of the invariants which hold at each program point, requiring support for multi-propositional mining. We have also implemented confidence[1] thresholds in Texada, which allow invariants to be mined in spite of exceptional traces or events.

IV. Proof of Concept

We tested Quarry on data-temporal traces found by running Daikon on one of its Java examples, QueueAr, an array-based queue. These data-temporal traces were created from the Daikon output as described in the previous section.

By mining the pattern $\mathrm{G}(p)$ with a confidence threshold of 0.9 we found $\mathrm{G}(\text{"}\texttt{this.back} <= \texttt{size}(\texttt{this.theArray}[\,]) - 1\text{"})$, meaning the queue's back pointer was nearly always within range of the array representing the queue. We were initially surprised that this property did not hold with confidence 1. But, further inspection revealed that this confidence reflects the fact that the array is empty at some time: at one violating program point the Daikon invariant $\texttt{this.back} >= 0$ was inferred but not $\texttt{this.back} <= \texttt{size}(\texttt{this.theArray}[\,]) - 1$. In fact, if this invariant held at confidence 1, it might be a signal that the test suite needs to be expanded to include situations when the queue is empty.

[1]Confidence of a property type measures how often that property type is falsified. A confidence of 1 indicates the property type is never falsified on the trace; a lower confidence means that the property type is sometimes falsified.

As another example, mining the property type $\neg x \mathrm{~U~} y$ with x set constant to "$\texttt{this.currentSize} <= \texttt{size}(\texttt{this.theArray}[\,]) - 1$" (current size of the queue is smaller than the maximum size of the array it is represented by) yielded two interesting values for y;

$$\boxed{\begin{array}{l}\neg(\texttt{this.currentSize} <= \texttt{size}(\texttt{this.theArray}[\,]) - 1) \mathrm{~U} \\ \qquad (\texttt{this.back} \leq \texttt{size}(\texttt{this.theArray}[\,]) - 1)\end{array}}$$

and

$$\boxed{\begin{array}{l}\neg(\texttt{this.currentSize} <= \texttt{size}(\texttt{this.theArray}[\,]) - 1) \mathrm{~U} \\ \qquad (\texttt{this.front} \leq \texttt{size}(\texttt{this.theArray}[\,]) - 1).\end{array}}$$

These suggest that the invariants relating the front position, back position, and size of the queue all appear at similar program points in the execution, likely when the queue is first created. We see that these invariants confirm correctness conditions for the queue.

V. Future Work and Concluding Remarks

Texada in its current form treats atomic propositions as strings, ignoring the logical interpretation of the invariants (data invariants are not simple atomic propositions, but logical clauses). This is problematic for several reasons; notably, Texada still considers two atomic propositions to be distinct based on their equality as strings. Consider a queue with initial capacity 10. We would expect to discover the invariant "$(\texttt{size} < 10) \mathrm{~U~} (\texttt{isFull} == \texttt{true})$". However, suppose at some point before $\texttt{isFull} == \texttt{true}$, the mined invariant is $\texttt{size} == 5$ instead of $\texttt{size} < 10$. This point in time would cause rejection of "$(\texttt{size} < 10) \mathrm{~U~} (\texttt{isFull} == \texttt{true})$" with full confidence. We are working on expanding this equality determination to more accurately reflect the logical meaning of the data invariants.

Run on its own, Daikon outputs a wide variety of invariants, some of which have little meaning in a temporal property mining context. For example, invariants which involve the local return value of a method describe this value simply as `return` regardless of which method returns it; out of context, we cannot know which method a return value corresponds to. This problem could be solved by re-introducing the context of these return values when we create the data-temporal trace by adding program point names.

Quarry presents a major contribution to program analysis tools, as it is the first tool to our knowledge concerned with finding temporal relationships of arbitrary length and complexity between data invariants. Our preliminary results suggest data-temporal properties yield more information about systems than data invariants and temporal properties alone.

Acknowledgements

Thanks to Dennis Park for implementing several important Quarry features, including multi-propositional trace support and data-temporal trace parsing. This research is funded by an NSERC discovery award and an NSERC USRA award.

References

[1] M. Bertasi, G. Di Guglielmo, and G. Pravadelli. Automatic generation of compact formal properties for effective error detection. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Sept 2013.

[2] M. Bonato, G. Di Guglielmo, M. Fujita, F. Fummi, and G. Pravadelli. Dynamic property mining for embedded software. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '12, pages 187–196, New York, NY, USA, 2012. ACM.

[3] M. Boshernitsan, R. Doong, and A. Savoia. From daikon to agitator: Lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, pages 169–180, New York, NY, USA, 2006. ACM.

[4] S.-C. K. David Lo and C. Liu. Mining temporal rules for software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):227–247, 2008.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 1999.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering (TSE)*, 27(2):99–123, 2001.

[7] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.

[8] M. Gabel and Z. Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, Atlanta, GA, USA, 2008.

[9] M. Gabel and Z. Su. Symbolic Mining of Temporal Specifications. In *Proceedings of the 2008 International Conference on Software Engineering*, 2008.

[10] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[11] H. B. Giles Reger and D. Rydeheard. Automata-based pattern mining from imperfect traces. In *Proceedings of the Second International Workshop on Software Mining*. ASE 2013, November 11-15.

[12] H. B. Giles Reger and D. Rydeheard. A pattern-based approach to parametric specification mining. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2013, November 11-15.

[13] S. Hangal, N. Chandra, S. Narayanan, and S. Chakravorty. Iodine: A tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42Nd Annual Design Automation Conference*, DAC '05, pages 775–778, New York, NY, USA, 2005. ACM.

[14] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 291–301, Orlando, FL, USA, 2002.

[15] J. Henkel and A. Diwan. A tool for writing and debugging algebraic specifications. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 449–458, Washington, DC, USA, 2004. IEEE Computer Society.

[16] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2008.

[17] Texada. https://bitbucket.org/bestchai/texada.

[18] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 496–506, Washington, DC, USA, 2009. IEEE Computer Society.

[19] W. van der Aalst, H. de Beer, and B. van Dongen. Process mining and verification of properties: An approach based on temporal logic. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE*, volume 3760 of *Lecture Notes in Computer Science*, pages 130–147. Springer Berlin Heidelberg, 2005.

[20] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[21] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 282–291, Shanghai, China, 2006.