Automated Support for Diagnosis and Repair

By Dalal Alrajeh, Jeff Kramer, Alessandra Russo, and Sebastian Uchitel

Model checking and logic-based learning together deliver automated support, especially in adaptive and autonomous systems.

key insights

* The marriage of model checking for finding faults and machine learning for suggesting repairs promises to be a worthwhile, synergistic relationship.
* Though separate software tools for model checking and machine learning are available, their integration has the potential for automated support of the common verify-diagnose-repair cycle.
* Machine learning ensures the suggested repairs fix the fault without introducing any new faults.

Edward Feigenbaum and Raj Reddy won the ACM A.C. Turing Award in 1994 for their pioneering

work demonstrating the practical importance and potential impact of artificial intelligence technology. Feigenbaum was influential in suggesting the use of rules and induction as a means for computers to learn theories from examples. In 2007, Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis won the Turing Award for developing model checking into a highly effective verification technology for discovering faults. Used in concert, verification and AI techniques can provide a powerful discovery and learning combination. In particular, the combination of model checking[10] and logic-based learning[15] has enormous synergistic potential for supporting the verify-diagnose-repair cycle software engineers commonly use in complex systems development. In this article, we show how to realize this synergistic potential.

Model checking exhaustively searches for property violations in formal descriptions (such as code, requirements, and design specifications, as well as network and infrastructure configurations), producing counterexamples when these properties do not hold. However, though model checkers are effective at uncovering faults in formal descriptions, they provide only limited support for understanding the causes of uncovered problems, let alone how to fix them. When uncovering a violation, model checkers usually provide one or more examples of how such a fault occurs in the description or model being analyzed. From this feedback, producing an explanation for the failure and generating a fix are complex tasks that tend to be human-intensive and error-prone. On the other hand, logic-based learning algorithms use correct examples and violation counterexamples to extend and modify a formal description such that the description conforms to the examples while avoiding the counterexamples. Although these counterexamples are usually provided manually, examples and counterexamples can be provided through verification technology (such as model checking).

Consider the problem of ensuring that a contract specification of an API satisfies some invariant.

Automated verification can be performed through a model checker that, should the invariant be violated, will return an example sequence of operations that breaks the invariant. Such a trace constitutes a counterexample that can then be used by a learning tool to correct the contract specification so the violation can no longer occur. The correction typically results in a strengthened post-condition for some operation so as to ensure the sequence does not break the invariant or perhaps a strengthened operation pre-condition so as to ensure the offending sequence of operations is no longer possible. For example, in Alrajeh[4] the contract specification of the engineered safety-feature-actuation subsystem for the safety-injection system of a nuclear power plant was built from scratch through the combined use of model checking and learning.

Another software engineering application for the combined technologies is obstacle analysis and resolution in requirements goal models. In it, the problem for software engineers is to identify scenarios in which high-level system goals may not be satisfied due to unexpected obstacles preventing lower-level requirements from being satisfied; for instance, in the London Ambulance System[21] an incident is expected to be resolved some time after an ambulance intervenes. For an incident to be so resolved, an injured patient must be admitted to the nearest hospital and the hospital must have all the resources to treat that patient. This goal is flawless performance, as it does not consider the case in which the nearest hospital lacks sufficient resources (such as a bed), a problem not identified in the original analysis. Model checking and learning helped identify and resolve this problem automatically. Model checking the original formal description of the domain against the stated goal automatically generates a scenario exemplifying this case; logic-based learning automatically revises the goal description according to this scenario by substituting the original with one saying patients should be admitted to a nearby hospital with available resources. A similar approach has also been used to identify and repair missing use cases in a television-set configuration protocol.[3]

The marriage of model checking and logic-based learning thus provides automated support for specification verification, diagnosis, and repair, reducing human effort and potentially producing a more robust product. The rest of the article explores a general framework for integrating model checking and logic-based learning (see Figure 1).

Basic Framework

The objective of the framework is to produce—from a given formal description and a property—a modified description guaranteed to satisfy the property. The software engineer's intuition behind combining model checking and learning is to view them as complementary approaches; model checking automatically detects errors in the formal description, and learning carries out the diagnosis and repair tasks for the identified errors, resulting in a correctly revised description.

To illustrate the framework—four steps executed iteratively—we consider the problem of developing a contract-based specification for a simplified train-controller system.[20] Suppose the specification includes the names of operations the train controller may perform and some of the pre- and post-conditions for each operation; for instance, the specification says there is an operation called "close doors" that causes a train's open doors to be closed. Other operations are for opening the train doors and starting and stopping the train. Two properties the system must guarantee are safe transportation (*P1*, or "train doors are closed when the train is moving") and rapid transportation (*P2*, or "train shall accelerate when the block signal is set to go") (see Figure 2).

Step 1. Model checking. The aim of this step is to check the formal description for violations of the property. The result is either a notification saying no errors exist, in which case the cycle terminates, or that an error exists, in which case a counterexample illustrating the error is produced

and the next step initiated. In the train-controller example, the model checker checks whether the specification satisfies the properties *P1* and *P2*. The checker finds a counterexample demonstrating a sequence of permissible operation executions leading to a state in which the train is moving and the doors are open, thereby violating the safe-transportation property *P1*. Since a violation exists, the verify-diagnose-repair process continues to the next step.

Step 2. Elicitation. The counterexample produced by the model-checking step is not an exhaustive expression of all ways property *P1* may be violated; other situations could also lead to a violation of *P1* and also of *P2*. This step gives software engineers an opportunity to provide additional and, perhaps, related counterexamples. Moreover, it may be that the description and properties are inconsistent; that is, all executions permitted by the description violate some property. Software engineers may therefore provide traces (called "witnesses") that exemplify how the property should have been satisfied. Such examples may be manually elicited by the software engineer(s) or automatically generated through further automated analysis. In the simplified train-controller system example, a software engineer can confirm the specification and properties are consistent by automatically eliciting a witness trace that shows how *P1* can be satisfied keeping the doors closed while the train is moving and opening them when the train has stopped.

Step 3. Logic-based learning. Having identified counterexamples and witness traces, the logic-based learning software carries out the repair process automatically. The learning step's objective is to compute suitable amendments to the formal description such that the detected counterexample is removed while ensuring the witnesses are accepted under the amended description. For the train controller, the specification corresponds to the available background theory; the negative example is the doors opening when the train is moving, and the positive example is the doors opening when it has stopped. The purpose of the repair task is to strengthen the pre- and post-conditions of the train-controller operations to prevent the train doors from opening when undesirable to do so. The learning algorithm finds the current pre-condition of the open-door operation is not restrictive

enough and consequently computes a strengthened pre-condition requiring the train to have stopped and the doors to be closed for such an operation to be executed.

Step 4. Selection. In the case where the logic-based learning algorithm finds alternative amendments for the same repair task, a selection mechanism is needed for deciding among them. Selection is domain-dependent and requires input from a human domain expert. When a selection is made, the formal description is updated automatically. In the simplified train-controller-system example, an alternative strengthened pre-condition—the doors are closed, the train is not accelerating—is suggested by the learning software, in which case the domain experts could choose to replace the original definition of the open-doors operation.

The framework for combining model checking and logic-based learning is intended to iteratively repair the formal description toward one that satisfies its intended properties. The correctness of the formal description is most often not realized in a single application of the four steps outlined earlier, as other violations of the same property or other properties may still exist. To ensure all violations are removed, the steps must be repeated automatically until no counterexamples are found. When achieved, the correctness of the framework's formal description is guaranteed.

Concrete Instantiation

We now consider model checking more formally, focusing on Zohar Manna's and Amir Pnueli's Linear Temporal Logic (LTL) and Inductive Logic Programming (ILP), a specific logic-based learning approach. We offer a simplified example on contract-based specifications and discuss our experience supporting several software-engineering tasks. For a more detailed account of model checking and ILP and their integration, see Alrajeh et al.,[5] Clarke,[10] and Corapi et al.[11]

Model checking. Model checkers require a formal description (M), also referred to as a "model," as

input. The input is specified using well-formed expressions of some formal language ($\mathcal{L}_M$) and a semantic mapping (s: $\mathcal{L}_M \rightarrow D$) from terms in $\mathcal{L}_M$ to a semantic domain (D) over which analysis is performed. They also require that the property (P) be expressed in a formal language ($\mathcal{L}_P$) for which there is a satisfiability relation ($\models \subseteq D \times \mathcal{L}_P$) capturing when an element of D satisfies the property. Given a formal description M and a property P, the model checker decides if the semantics of M satisfies the property $s(M) \models P$.

Model checking goes beyond checking for syntactic errors a description M may have by performing an exhaustive exploration of its semantics. An analogy can be made with modern compilers that include sophisticated features beyond checking if the code adheres to the program language syntax and consider semantic issues (such as to de-reference a pointer with a null value). One powerful feature of model checking for system fault detection is its ability to automatically generate counterexamples that are meant to help engineers identify and repair the cause of a property violation (such as an incompleteness of the description with respect to the property being checked, or $s(M) \not\models P$ and $s(M) \not\models \neg P$), an incorrectness of the description with respect to the property, or $s(M) \models \neg P$), and the property itself being invalid. However, these tasks are complex, and only limited automated support exists for resolving them consistently. Even in relatively small simplified descriptions, such resolution is not trivial since counterexamples are expressed in terms of the semantics rather than the language used to specify the description or the property, counterexamples show symptoms of the cause but not the cause of the violation itself, and any manual modification to the formal description could fail to resolve the problem and introduce violations to other desirable properties.

Consider the example outlined in Figure 3. The formal description M is a program describing a train-controller class using $\mathcal{L}_M$, a JML-like specification language. Each method in the class is coupled with a definition of its pre-conditions (preceded with the keyword `requires`) and post-conditions (preceded by the keyword `ensures`). The semantics of the program is defined over a labeled transition system (LTS) in which nodes represent the different states of the program and edges represent the method calls that cause the program to transit from one state to another. Property P is an assertion indicating what must hold at every state in every execution of the LTS $s(M)$. The language $\mathcal{L}_P$ used for expressing these properties is LTL. The first states it should always be the case (where $\Box$ means always) that if a train tr is moving, then its doors are closed. The second states the train tr shall accelerate within three seconds of the block signal b at which it is located being set to go. To verify $s(M) \models P$, an explicit model checker first synthesizes an LTS that represents all possible executions permitted by the given program M. It then checks whether P is satisfied in all executions of the LTS.

In the train-controller example, there is an execution of $s(M)$ that violates $P1 \wedge P2$; hence a counterexample is produced (see Figure 3). Despite the simplicity and size of this counterexample, the exact cause of the violation is not obvious to the software engineer. Is it caused by an incorrect method invocation, a missing one, or both? If an incorrect method invocation, which method should not have been called? Should this invocation be corrected by strengthening its precondition or changing the post-condition of previously called operations? If caused by a missing invocation, which method should have been invoked? And under what conditions?

To prepare the learning step for a proper diagnosis of the encountered violations, witness traces to the properties are elicited. They may be provided either by the software engineer through

specification, simulation, and animation techniques or through model checking. Figure 3 includes a witness trace elicited from $s(M)$ by model checking against $(\neg P1 \vee \neg P2)$. In this witness, the train door remains closed when the train is moving, satisfying $P1$ and satisfying $P2$ vacuously.

Inductive Logic Programming

Once a counterexample and witness traces have been produced by the model checker, the next step involves generation of repairs to the formal description. If represented declaratively, automatic repairs can be computed by means of ILP. ILP is a learning technique that lies at the intersection of machine learning and logic programming. It uses logic programming as a computational mechanism for representing and learning plausible hypothesis from an incomplete or incorrect background knowledge and a set of examples. A logic programs is defined as a set of rules of the form $\mathtt{h} \leftarrow \mathtt{b}_1$, …, $\mathtt{b}_j$, $\mathtt{not\ b}_{j+1}$, …, $\mathtt{not\ b}_n$, which can be read as whenever $\mathtt{b}_1$ and … and $\mathtt{b}_j$ hold, and $\mathtt{b}_{j+1}$ and … and $\mathtt{b}_n$ do not hold, then $\mathtt{h}$ holds. In a given clause, $\mathtt{h}$ is called the "head" of the rule, and the conjunction $\{\mathtt{b}_1$, …, $\mathtt{b}_j$, $\mathtt{not\ b}_{j+1}$, …, $\mathtt{not\ b}_n\}$ is the "body" of the rule. A rule with an empty body is called an "atom," and a rule with an empty head is called an "integrity constraint." Integrity constraints given in the initial description are assumed to be correct (therefore not revisable) and must be satisfied by any learned hypothesis.

In general, ILP requires as input background knowledge (B) and set of positive (E^+) and negative (E^-) examples that, due to incomplete information, may not be inferable but that are consistent with the current background knowledge. The task for the learning algorithm is to compute a hypothesis (H) that extends B so as to entail the set of positive examples ($B \wedge H \models E^+$) without covering the negative ones ($B \wedge H \not\models E^-$). Different notions of entailment exist, some weaker than others;[16] for instance, cautious (respectively brave) entailment requires what appears on the right-hand side of

the entailment operator to be true (in the case of $\models$) or false (in the case of $\not\models$) in every (respectively at least one) interpretation of the logic program on the right. ILP, like all forms of machine learning, is fundamentally a search process. Since the goal is to find a hypothesis for given examples, and many alternative hypotheses exist, these alternatives are considered automatically during the computation process by traversing a lattice-type hypothesis space based on a generality-ordering relation for which the more examples a hypothesis explains, the more general is the hypothesis.

"Non-monotonic" ILP is a particular type of ILP that is, by definition, capable of learning hypothesis *H* that alters the consequences of a program such that what was true in *B* alone is not necessarily true once *B* is extended with *H*. Non-monotonic ILP is therefore well-suited for computing revisions of formal descriptions, expressed as logic programs. The ability to compute revisions is particularly useful when an initial, roughly correct specification is available and a software engineer wants to improve it automatically, or semi-automatically, according to examples acquired incrementally over time; for instance, when evidence of errors in a current specification is detected, a revision is needed to modify the specification such that the detected error is no longer possible. However, updating the description with factual information related to the evidence would simply amount to recording facts. So repairs must generalize from the detected evidence and construct minimal but general revisions of the given initial specification that would ensure its semantics no longer entails the detected errors. The power of non-monotonic ILP to make changes to the semantics of a given description makes it ideal for the computation of repairs. Several non-monotonic ILP tools (such as XHAIL and ASPAL) are presented in the machine learning literature where the soundness and completeness of their respective algorithms have been shown. These tools typically aim to find minimal solutions according to a predefined score function that considers the size of the constructed hypotheses, number of positive examples covered, and number of negative examples not covered as parameters.

Integration. Integration of model checking with ILP involves three main steps: translation of the formal description; counterexamples and witness traces generated by the model checker into logic programs appropriate for ILP learning; computation of hypotheses; and translation of the hypotheses into the language $\mathcal{L}_M$ of the specification (see Figure 4).

Consider the background theory in Figure 4 for our train example. This is a logic program representation of the description *M* together with its semantic properties. Expressions "`train(tr1)`" and "`method(openDoors(Tr))` ← `train(Tr)`" say `tr1` is a train and `openDoors(Tr)` is a method, whereas the expression "← `execute(M, T), requires(M, C), not holds(C, T)`" is an integrity constraint that captures the semantic relationship between method execution and their pre-conditions; the system does not allow for a method *M* to be executed at a time *T* when its pre-condition *C* does not hold at that time. Expressions like `execute(openDoors(Tr), T)` denote the narrative of method execution in a given execution run.

The repair scenario in Figure 4 assumes a notion of "brave entailment" for positive examples E^+, or E^+ must be consistent with $B \wedge H$, and a notion of "cautious" entailment for E^-, or E^- must be inconsistent with $B \wedge H$. Although Figure 4 gives only an excerpt of the full logic program representation, it is possible to intuitively see that, according to this definition of entailment, the conjunction of atoms in E^+ is consistent with *B*, but the conjunction of the negative examples in E^- is also consistent with *B*, since all defined pre-conditions of the methods executed in the example runs are satisfied. The current description expressed by the logic program *B* is thus erroneous. The learning phase, in this case, must find hypotheses *H*, regarding method pre- and post-conditions, that together with *B* would ensure the execution runs represented by E^- would no longer be

consistent with B. In the train-controller scenario, two alternative hypotheses are found using ASPAL.

Once the learned hypotheses are translated back into the language $\mathcal{L}_M$, the software engineer can select from among the computed repairs, and the original description can be updated accordingly.

Applications. As mentioned earlier, we have successfully applied the combination of model checking and ILP to address a variety of requirements-engineering problems (see the table here). Each application consisted of a validation against several benchmark studies, including London Ambulance System (LAS), Flight Control System (FCS), Air Traffic Control System (ATCS), Engineered Safety Feature Actuation System (ESFAS) and Philips Television Set Configuration System (PTSCS).

The size of our case studies was not problematic. In general, our approach was dependent on the scalability of the underlying model checking and ILP tools influenced by the size of the formal description and properties being verified, expressiveness of the specification language, number and size of examples, notion of entailment adopted, and characteristics of the hypotheses to be constructed. As a reference, in the goal operationalization of the ESFAS, the system model consists of 29 LTL propositional atoms and five goal expressions. We used LTL model checking, which is coNP-hard and PSPACE-complete, and XHAIL, the implementation of which is based on a search mechanism that is NP-complete. We had to perform 11 iterations to reach a fully repaired model, with an average cycle computation time of approximately 88 seconds; see Alrajeh et al.[4] for full details on these iterations.

Related Work

Much research effort targets fault detection, diagnosis, and repair, some looking to combine verification and machine learning in different ways; for example, Seshia[17] showed tight integration of induction and deduction helps complete incomplete models through synthesis, and Seshia[17] also made use of an inductive inference engine from labeled examples. Our approach is more general in both scope and technology, allowing not only for completing specifications but also for changing them altogether.

Testing and debugging are arguably the most widespread verify-diagnose-repair tasks, along with areas of runtime verification, (code) fault localization, and program repair. Runtime verification aims to address the boundary between formal verification and testing and could provide a valid alternative to model checking and logic-based learning, as we have described here. Fault localization has come a long way since Mark Weiser's[22] breakthrough work on static slicing, building on dynamic slicing,[1] delta debugging,[23] and others. Other approaches to localization based on comparing invariants, path conditions, and other formulae from faulty and non-faulty program versions also show good results.[12] Within the fault localization domain, diagnosis is often based on statistical inference.[14]

Model checking and logic-based reasoning are used for program repair; for example, Buccafurri et al.[8] used abductive reasoning to locate errors in concurrent programs and suggest repairs for very specific types of errors (such as variable assignment and flipped consecutive statements). This limitation was due to the lack of a reasoning framework that generalizes from ground facts. Logic-based learning allows a software engineer to compute a broader range of repairs.

A different, but relevant, approach for program synthesis emerged in early 2010[18] where the emphasis was on exploiting advances in verification (such as inference of invariants) to encode a

program-synthesis problem as a verification problem. Heuristic-based techniques (such as genetic algorithm-based techniques[13]) aimed to automatically change a program to pass a particular test. Specification-based techniques aim to exploit intrinsic code redundancy.[9] Contrary to our work and that of Buccafurri et al.,[8] none of these techniques guarantees a provably correct repair.

Theorem provers are able to facilitate diagnosing errors and repairing descriptions at the language level. Nonetheless, counterexamples play a key role in human understanding, and state-of-the-art provers (such as Nitpick[6]) have been extended to generate them. Beyond counterexample generation, repair is also being studied; for instance, in Sutcliffe and Puzis[19] semantic and syntactic heuristics for selecting axioms were changed. Logic-based learning offers a means for automatically generating repairs over time rather than requiring the software engineer to predefine them. Machine-learning approaches that are not logic-based have been used in conjunction with theorem proving to find useful premises that help prove a new conjecture based on previously solved mathematical problems.[2]

Conclusion

To address the need for automated support for verification, diagnosis, and repair in software engineering, we recommend the combined use of model checking and logic-based learning. In this article, we have described a general framework combining model checking and logic-based learning. The ability to diagnose faults and propose correct resolutions to faulty descriptions, in the same language engineers used to develop them, is key to support for many laborious and error-prone software-engineering tasks and development of more-robust software.

Our experience demonstrates the significant benefits this integration brings and indicates its potential for wider applications, some of which were explored by Borjes et al.[7] Nevertheless,

important technical challenges remain, including support for quantitative reasoning like stochastic behavior, time, cost, and priorities. Moreover, diagnosis and repair are essential not only during software development but during runtime as well. With the increasing relevance of adaptive and autonomous systems, there is a crucial need for software-development infrastructure that can reason about observed and predicted runtime failures, diagnose their causes, and implement plans that help them avoid or recover from them.

References

1. Agrawal, H. and Horgan, J.R. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, New York, June 20–22). ACM Press, New York, 1990, 246–256.

2. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., and Urban, J. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning 52,* 2 (Feb. 2014), 191–213.

3. Alrajeh, D., Kramer, J., Russo, A., and Uchitel, S. Learning from vacuously satisfiable scenario-based specifications. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering* (Tallinn, Estonia, Mar. 24–Apr. 1). Springer, Berlin, 2012, 377–393.

4. Alrajeh, D., Kramer, J., Russo, A., and Uchitel, S. Elaborating requirements using model checking and inductive learning. *IEEE Transaction Software Engineering 39,* 3 (Mar. 2013), 361–383.

5. Alrajeh, D., Russo, A., Uchitel, S., and Kramer, J. Integrating model checking and inductive logic programming. In *Proceedings of the 21st International Conference on Inductive Logic Programming* (Windsor Great Park, U.K., July 31–Aug. 3). Springer, Berlin, 2012, 45–60.

6. Blanchette, J.C. and Nipkow, T. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *Proceedings of the first International Conference on Interactive Theorem Proving* (Edinburgh, U.K., July 11–14). Springer, Berlin, 2010, 131–146.

7. Borges, R.V., d'Avila Garcez, A.S., and Lamb, L.C. Learning and representing temporal knowledge in recurrent networks. *IEEE Transactions on Neural Networks 22,* 12 (Dec. 2011), 2409–2421.

8. Buccafurri, F., Eiter, T., Gottlob, G., and Leone, N. Enhancing model checking in verification by AI techniques. *Artificial Intelligence 112,* 1–2 (Aug. 1999), 57–104.

9. Carzaniga, A., Gorla, A., Mattavelli, A., Perino, N., and Pezzé, M. Automatic recovery from runtime failures. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, May 18–26). IEEE Press, Piscataway, NJ, 2013, 782–791.

10. Clarke, E.M. The birth of model checking. In *25 Years of Model Checking,* O. Grumberg and H. Veith, Eds. Springer, Berlin, 2008, 1–26.

11. Corapi, D., Russo, A., and Lupu, E. Inductive logic programming as abductive search. In *Technical Communications of the 26th International Conference on Logic Programming,* M. Hermenegildo and T. Schaub, Eds. (Edinburgh, Scotland, July 16–19). Schloss Dagstuhl, Dagstuhl,

Germany, 2010, 54–63.

12. Eichinger, F. and Bohm, K. Software-bug localization with graph mining. In *Managing and Mining Graph Data,* C.C. Aggarwal and H. Wang, Eds. Springer, New York, 2010, 515–546.

13. Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation* (Montreal, Canada, July 8–12). ACM Press, New York, 2009, 947–954.

14. Liblit, B., Naik, M., Zheng, A.X., Aiken, A., and Jordan, M.I. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, June 12–15). ACM Press, New York, 2005, 15–26.

15. Muggleton, S. and Marginean, F. Logic-based artificial intelligence. In *Logic-based Machine Learning,* J. Minker, Ed. Kluwer Academic Publishers, Dordrecht, the Netherlands, 2000, 315–330.

16. Sakama, C. and Inoue, K. Brave induction: A logical framework for learning from incomplete information. *Machine Learning 76,* 1 (July 2009), 3–35.

17. Seshia, S.A. Sciduction: Combining induction, deduction, and structure for verification and synthesis. In *Proceedings of the 49th ACM/EDAC/IEEE Design Automation Conference* (San Francisco, CA, June 3–7). ACM, New York, 2012, 356–365.

18. Srivastava, S., Gulwani, S., and Foster, J.S. From program verification to program synthesis. *SIGPLAN Notices 45,* 1 (Jan. 2010), 313–326.

19. Sutcliffe, G., and Puzis, Y. Srass: A semantic relevance axiom selection system. In *Proceedings of the 21st International Conference on Automated Deduction* (Bremen, Germany, July 17–20). Springer, Berlin, 2007, 295–310.

20. van Lamsweerde, A. *Requirements Engineering: From System Goals to UML Models to Software Specifications.* John Wiley & Sons, Inc., New York, 2009.

21. van Lamsweerde, A. and Letier, E. Handling obstacles in goal-oriented requirements engineering. *IEEE Transaction on Software Engineering 26,* 10 (Oct. 2000), 978–1,005.

22. Weiser, M. Program slicing. In *Proceedings of the Fifth International Conference on Software Engineering* (San Diego, CA, Mar. 9–12). IEEE Press, Piscataway, NJ, 1981, 439–449.

23. Zeller, A. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the Seventh European Software Engineering Conference* (held jointly with the Seventh ACM SIGSOFT International Symposium on Foundations of Software Engineering) (Toulouse, France, Sept. 6–10), Springer, London, 1999, 253–267.

Dalal Alrajeh (dalal.alrajeh@imperial.ac.uk) is a junior research fellow in the Department of Computing at Imperial College London, U.K.

Jeff Kramer (j.kramer@imperial.ac.uk) is a professor of distributed computing in the Department at Computing of Imperial College London, U.K.

Alessandra Russo (a.russo@imperial.ac.uk) is a reader in applied computational logic in the Department of Computing at Imperial College London, U.K.

Sebastian Uchitel (s.uchitel@imperial.ac.uk) is a reader in software engineering in the Department of Computing at Imperial College London, U.K., and an ad-honorem professor in the Departamento de Computation and **the National Scientific and Technical Research Council??**, or CONICET, at the University of Buenos Aires, Argentina.

Pull Quotes

The marriage of model checking and logic-based learning thus provides automated support for specification verification, diagnosis, and repair, reducing human effort and potentially producing a more robust product.

Model checking automatically detects errors in the formal description, and learning carries out the diagnosis and repair tasks for the identified errors, resulting in a correctly revised description.

Machine-learning approaches that are not logic-based have been used in conjunction with theorem proving to find useful premises that help prove a new conjecture based on previously solved mathematical problems.[2]