From One To Many: Checking A Set Of Models

Rohit Dureja and Kristin Yvonne Rozier
Iowa State University, Ames, USA

***Abstract*—Whether the objective is to narrow in on the final system design, check capabilities of system versions with varying features, or regression testing to make a design more robust, several models of the system under development have to be analyzed. Model checking can compare different models; however, applying model checking off-the-shelf may not scale due to the large size of the design space for today's complex systems. There is a need to develop new algorithms that intelligently use inherent properties of models in a design space to increase scalability of checking the complete model-set. We report on our investigation of the *model-set checking problem*, highlight preliminary results, and discuss ongoing work and future research directions.**

I. Introduction

The design of complex systems often requires analyzing several models of the system under development. The set of models constitute the design space of the system. Models in a set may represent different design options for the system, features, or bug fixes; different models differ in terms of core capabilities, behavioral implementations, and component configurations. The design space of a modern microprocessor with configurable cache size, number of registers, and pipeline depth, etc., is one such example. Model checking can be used to aid system development via a thorough comparison of the set of models. In the classical approach, each model in the set is checked one-by-one against a set of properties representing system requirements. However, for large and complex design spaces, such an approach can be inefficient and even fail to scale to handle the combinatorial size of the design space. Nevertheless, model checking remains the most widely used method in industry when dealing with such systems [1, 2].

Model-Set Checking Problem. Given a model M and a property φ, a classical model checker checks whether $M \models \varphi$? If the answer is *yes*, the checker returns a proof, otherwise, a counterexample is returned. We lift classical one-model checking to model-set checking. The *model-set checking problem* is "given a set of competing models for a system and a property, check whether each model in the set satisfies the property". Stated formally, for a set of models $\mathcal{M}$, and property φ, check for each $M \in \mathcal{M}$ whether $M \models \varphi$.

Related Work. Product line verification techniques, e.g., with Software Product Lines (SPL), also verify models describing large design spaces [3]. The several *instances* of feature transition systems (FTS) [4] describe a set of models. Our work allows models in a set, that cannot be combined as a FTS, to be checked. Model-sets are also generated in *regression verification* [5], where a new version of a design is re-verified with the same (or similar) property. *Parameter synthesis* [6] can generate configurations for models that satisfy a property, whereas, we are interested in all models in a set. The parameterized model checking problem (PMCP) [7] deals with infinite families of homogeneous systems. In our case, the models are finite and heterogeneous.

Thanks to NSF CAREER Award CNS-1552934 for supporting this work.

II. Preliminary Results

An important observation is that different models in the design space are related, i.e., they have overlapping state spaces. Fig. 1 shows the reachable state space for a model-set $\mathcal{M} = \{M_1, M_2, M_3, M_4\}$ as a Venn diagram. In the classical scenario, a model checking algorithm does not take advantage of this information and ends-up re-verifying overlapping state spaces across models. For large models this is wasteful as every model checking run re-explores already known reachable states. Therefore, as the number of models grow, learning and reusing information from solving related models becomes very important for future checking efforts; like reusing variable ordering in BDD-based model checking. For example, assuming M_1 is checked before M_2 in Fig. 1, the checker run for M_2 can reuse the already explored and verified state space of M_1.

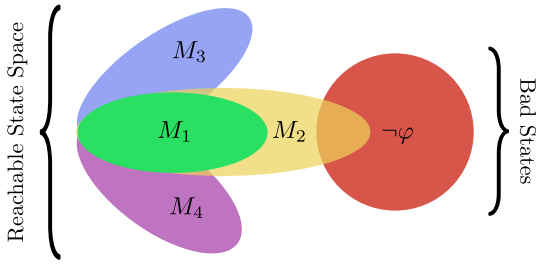

Fig. 1. Venn diagram representation of reachable states for a set of models $\mathcal{M} = \{M_1, M_2, M_3, M_4\}$ and bad states $\neg\varphi$. Model $M_2 \not\models \varphi$ since reachable states of M_2 intersect bad states $\neg\varphi$.

In [8], we extend one of the fastest bit-level verification methods, IC3 [9], to deal with a set of models. The algorithm, FuseIC3, automatically reuses information from earlier model-checking runs to minimize time spent in exploring the state space in common between related models. Given a set of models and a safety property, FuseIC3 sequentially checks each models by reusing information: reachable state approximations, counterexamples, and invariants, learned in earlier runs to reduce the set's total checking time. When the stored information cannot be reused directly, FuseIC3 repairs and patches the information using an efficient algorithm. It adds "just enough" extra information to the saved reachable states to enable reuse. Our experiments on real-life challenging

benchmarks demonstrate that FuseIC3 is a median 1.75× (average 4.39×) faster than checking each model individually.

III. Ongoing and Future Work

Fig. 2 shows the verification workflow for checking a set of models. Given a set of models $\mathcal{M}$, a property φ, a model-set checking algorithm checks if each model in $\mathcal{M}$ satisfies property φ by reusing information learned during earlier checker runs. The output is a report containing results for every model.

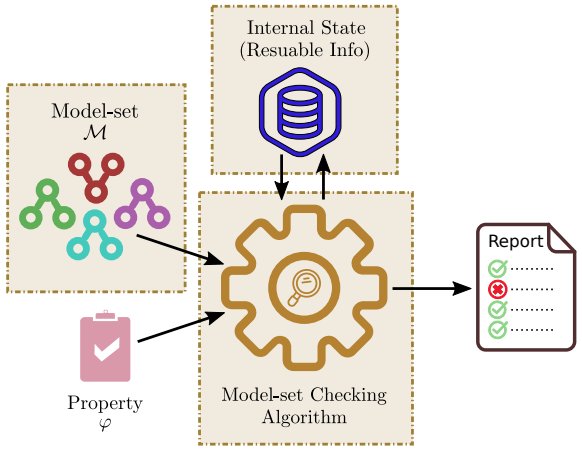

Fig. 2. Verification work-flow for model-set checking. Given set of models $\mathcal{M}$, property φ, and internal state, a model-set algorithm checks if φ holds for models in $\mathcal{M}$. The shaded regions indicate avenues for future research.

We identify three avenues of research in the model-set verification workflow in Fig. 2. Experiments using FuseIC3 indicate that algorithm performance is influenced by the order the models are checked in, and how much stored information is reused. Two heuristics that may improve performance are:

1. Checking Order for Models in a Set

   Each model in a set is defined over the same set of state variables. A transition relation for a model M over Boolean variables x, y, and z is of the form $(x'=\ldots \wedge y'=\ldots \wedge z'=\ldots)$, where primed variables denote next state. There are two points to consider:

   i) Related models may have the same next state assignment for some of their variables.
   ii) Each next state assignment is a Boolean circuit that allows us to determine overlap between two assignments by organizing the Boolean formula in a canonical form.

   For example, using a Karnaugh map, the number of similar groupings is proportional to the degree of overlap. Both metrics can be computed in linear time for two models. We organize the models as a weighted undirected graph. Edge weight between two models is proportional to the overlap between them, with the first metric contributing more weight than the second. An approximate polynomial-time solution to the *nearest-neighbor* Hamiltonian Path Problem on the complete graph gives a possible ordering in which the models can be checked.

2. Ranking Information based on Importance and Relevance

   The state approximations reused in FuseIC3 are Boolean formulas in CNF, and the algorithm tries to selectively repair them. A lazy approach to repair may reduce the number of clauses used in a run of FuseIC3. We remove the violating clauses from the state approximation. When a bad state is found in the blocking phase, we use one of the violating clauses from the predecessor frame to block the bad state. The choice is determined by:

   i) Number of literals common between the bad state and the violating clause.
   ii) Number of frames the violating clause appears in the last checked model.

   The chosen clause is then repaired and the algorithm continues. In other words, we only repair clauses that block a bad state in the future, instead of repairing them prematurely.

We plan to incorporate all theoretical advances in a tool for checking large design spaces [10]. Since checking large design spaces is becoming commonplace, we plan to develop more model-sets and make them publicly available as research benchmarks.

IV. Conclusion

The design of complex systems often requires analyzing several models of a system under test and model-set checking can aid development via a thorough comparison of the models. Our preliminary results demonstrate that model-set algorithms that make use of the related information between models, outperform one-model checking algorithms. We identify future work along three facets of the verification workflow and describe two heuristics that may improve performance of model-set checking algorithms. Finally, model-set checking can benefit greatly from the availability of a portfolio of algorithms and tools that implement them.

References

[1] M. Bozzano, A. Cimatti, A. Fernandes Pires, D. Jones, G. Kimberly, T. Petri, R. Robinson, and S. Tonetta, "Formal design and safety analysis of AIR6110 wheel brake system," in *CAV*, 2015.

[2] M. Gario, A. Cimatti, C. Mattarei, S. Tonetta, and K. Y. Rozier, "Model checking at scale: Automated air traffic control design space exploration," in *CAV*, 2016.

[3] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE*, 2010, pp. 335–344.

[4] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1069–1089, 2013.

[5] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *ICSM*, 2009, pp. 115–124.

[6] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta, "Parameter synthesis with IC3," in *FMCAD*, 2013, pp. 165–168.

[7] E. A. Emerson and V. Kahlon, "Reducing model checking of the many to the few," in *CADE*, 2000, pp. 236–254.

[8] R. Dureja and K. Y. Rozier, "FuseIC3: An algorithm for checking large design spaces," in *FMCAD*, 2017.

[9] A. R. Bradley, "SAT-Based Model Checking without Unrolling," in *VMCAI*, 2011, pp. 70–87.

[10] R. Dureja and K. Y. Rozier, "Nexus: A model checker for large design spaces," (ongoing work).