Learning-Based Assume-Guarantee Verification (Tool Paper)

Dimitra Giannakopoulou[1] and Corina S. Păsăreanu[2]

[1] RIACS
[2] QSS, NASA Ames, Moffett Field, CA 94035-1000, USA
{dimitra, pcorina}@email.arc.nasa.gov

1 Introduction

Despite significant advances in the development of model checking, it remains a difficult task in the hands of experts to make it scale to the size of industrial systems. A key step in achieving scalability is to "divide-and-conquer", that is, to break up the verification of a system into smaller tasks that involve the verification of its components. Assume-guarantee reasoning [9, 11] is a widespread "divide-and-conquer" approach that uses assumptions when checking individual components of a system. Assumptions essentially encode expectations that each component has from the rest the system in order to operate correctly. Coming up with the right assumptions is typically a non-trivial manual process, which limits the applicability of this type of reasoning in practice.

Over the last few years, we have developed a collection of techniques and a supporting toolset, for performing assume-guarantee reasoning of software in an automated fashion. Our techniques are applicable both at the level of design models, and at the level of actual source code. In the heart of these techniques lies a framework that uses an off-the-shelf learning algorithm for regular languages, namely L* [1], to compute assumptions automatically.

The rest of the paper is organized as follows. Section 2 is a high-level description of our techniques for learning-based assume-guarantee reasoning of software. Section 3 discusses the tool support for our techniques and experimental results obtained from the application of our approach to some industrial size case studies, and we conclude the paper with Section 4.

2 Learning-Based Assume-Guarantee Reasoning

Analysis of Finite State Models. At the design level, our techniques target models described as labeled transition systems (LTSs) with blocking communication. We check safety properties expressed as finite state machines that describe the legal sequences of actions that a system can perform. We reason about assume-guarantee formulas $\langle A\rangle M\langle P\rangle$, where M is a component, P is a property and A is an assumption on M's environment. The formula is true if whenever M is part of a system that satisfies A, then the system must also guarantee P.

$\dfrac{\begin{array}{l} 1: \langle A_1\rangle M_1\langle P\rangle \\ 2: \langle true\rangle M_2\langle A_1\rangle \end{array}}{\langle true\rangle M_1 \parallel M_2\langle P\rangle}$	$\dfrac{\begin{array}{l} 1: \langle A_1\rangle M_1\langle P\rangle \\ 2: \langle A_2\rangle M_2\langle P\rangle \\ 3: C(A_1, A_2, P) \end{array}}{M_1 \parallel M_2 \models P}$	$\dfrac{\begin{array}{l} 1..n: \ \langle A_i\rangle M_i\langle P\rangle \\ n+1: C(A_1, \cdots A_n, P) \end{array}}{\langle true\rangle M_1 \parallel \cdots \parallel M_n\langle P\rangle}$
(a)	(b)	(c)

Fig. 1. Assume-guarantee rules

Our framework is equipped with a collection of assume-guarantee rules which are sound and complete [2]. Incomplete rules can also be incorporated. The simplest (non-symmetric) assume guarantee rule (see Figure 1 (a)) establishes that property P holds for the composition of two models M_1 and M_2. In [6], we present an approach that uses this rule to perform assume-guarantee reasoning in an incremental and fully automatic fashion. The approach iterates a process based on gradually *learning* an assumption that is strong enough for M_1 to satisfy P but weak enough to be an abstraction of M_2's behavior.

The framework also handles symmetric rules [2]. These rules are instances of the rule pattern presented in Figure 1 (b). $C(A_1, A_2, P)$ represents some logical condition that involves the two assumptions and the property. For example, an instance of this rule states as the third premise that $\overline{A_1} \parallel \overline{A_2} \models P$. Here $\overline{A_1}$ and $\overline{A_2}$ denote the complement automata. Intuitively, this premise ensures that the possible common traces of M_1 and M_2, which are ruled out by the two assumptions, satisfy the property.

The approach extends to reasoning about n components. For the non-symmetric rule, we can decompose the system into two parts M_1 and $M_2' = M_2 \parallel \cdots \parallel M_n$, and apply the approach recursively for checking Premise 2. The generalization for symmetric rules follows the pattern of Figure 1 (c); its use in the context of learning-based assume-guarantee verification is illustrated in Figure 2.

The input models M_1, ..., M_n are created by the user or extracted from source code, using automated abstraction techniques, as discussed later in this section. At each iteration, L* generates approximate assumptions A_1, ... A_n. Model checking is then used to determine whether $\langle A_i\rangle M_i\langle P\rangle$ holds for each $i = 1..n$. If the result of any of these checks is false, then L* uses the returned counterexample to refine the corresponding assumption. The refinement process iterates until we obtain assumptions that are appropriate for showing that the first n premises hold. The last premise is then checked to discharge the assumptions; if it holds, then, according to the compositional rule, $M_1 \parallel \cdots \parallel M_n \models P$. Otherwise, the obtained counterexample is analyzed to see if it corresponds to a real error, or it is spurious, in which case it is used to refine the assumptions. The counterexample analysis is performed component wise.

For finite state systems, the iterative learning process terminates and it yields minimal assumptions [6]. In our experience, the generated assumptions are usually orders of magnitude smaller than the analyzed components, and the cost of learning-based assume-guarantee verification is small as compared to non-compositional model checking. This is often the case for well designed software,

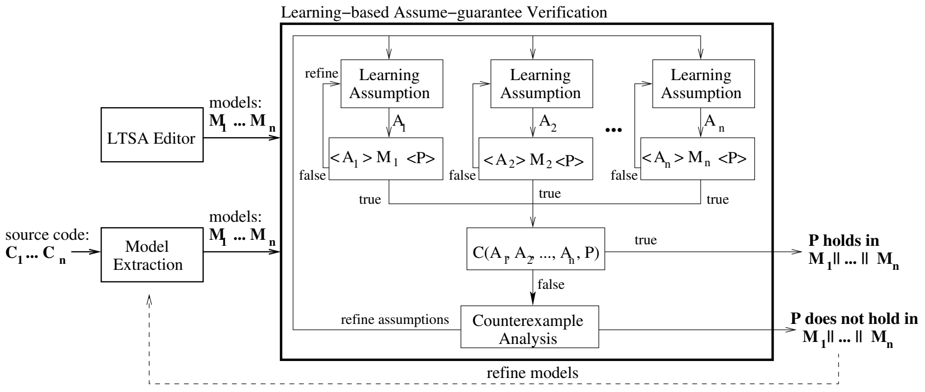


Fig. 2. Learning based assume-guarantee verification

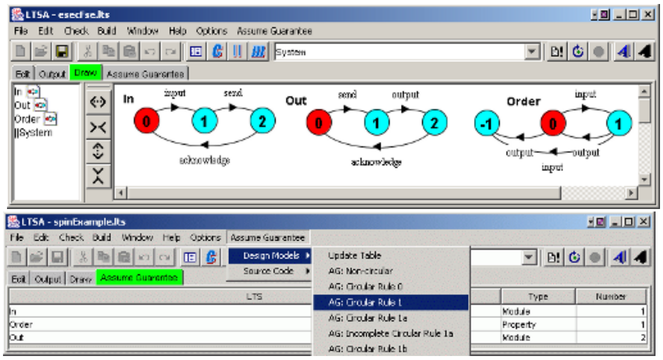
where the interfaces between components are usually small. However, there may be cases where no single rule or no particular system decomposition yields small assumptions. Our framework partially alleviates this problem, by providing a collection of rules that the user can select and experiment with. The decomposition is still a manual process.

Analysis of Source Code. Assume first a top-down software development process, where one creates and debugs design models, which are then used to guide the development of source code (possibly in a (semi-) automatic way by code synthesis). In such a setting, the assumptions created at the design level can be used to check source code in an assume guarantee style, as presented in [8].

For cases where design models are not available, we have recently extended our framework with a component for model extraction from source code (see Figure 2). The framework is iterative: extracted models are analyzed in an assume-guarantee style, and when the analysis detects spurious errors due to the abstraction of the source code, the models are refined automatically. The extended framework advocates a clear separation between model extraction and model analysis, which facilitates the incorporation of existing well-engineered tools into it. For example, we have integrated our framework with the Magic tool that extracts finite-state models from C code using automated predicate abstraction and refinement [4]. Other tools that build finite-state models of software could also be used (e.g. Bandera for Java [7]).

3 Tool Support

Implementation. The techniques presented in the previous section have been implemented in the context of the LTSA tool [10]. The LTSA supports model checking of a system based on its architecture. It features graphical display, animation and simulation of LTSs (see Figure 3). Its input language “FSP” is



LTS	Type	Number
In	Module	1
Order	Property	1
Out	Module	2

Fig. 3. LTSA GUI including Assume-Guarantee Plugin

a process-algebra style notation with LTS semantics. The LTSA has an extensible architecture which allows extra features to be added by means of plugins [5].

Our initial learning framework [6] was implemented within the core of the LTSA tool. This implementation is efficient because it can directly manipulate the internal data structures of the LTSA. However, such a solution is not sustainable, because it is hard to synchronize our code development with that of the LTSA. For this reason, we proceeded by implementing our extensions to the LTSA as the `Assume-Guarantee` plugin.

The `Assume-Guarantee` plugin extends the LTSA with a menu and a tab (see Figure 3). The menu provides options for analyzing `Design Models` and `Source Code` (uses Magic to extract models), and for selecting assume-guarantee rules. For the case of design models, all the processes in the specification are displayed in the tab, so that the user may select which components and properties participate in an assume-guarantee proof. For source code, these choices are currently hard-coded, but we intend to improve on this in the future.

Note that, at the design level, there is a significant performance overhead incurred by the plug-in implementation, which is not present in the original, non plug-in implementation. This is due to the fact that plug-ins communicate with the LTSA by placing FSP descriptions of LTSs in the Edit tab. In the future, we expect the LTSA developers to expose LTSs as objects which will enable us to do a more efficient implementation.

Experience. Within a project at NASA Ames, we have applied our techniques to the design and code of the K9 Rover Executive. The design models consist of approximately 700 lines of FSP code. The code we analyzed is about 7K lines of Java, translated from 10K lines of C++ code. Some results are shown in the tables below (monolithic refers to non-compositional verification).

We have also applied our integrated tool-set with the Magic model extractor to the verification of various safety properties of OpenSSL version 0.9.6c which has about 74,000 lines of C code. Our approach achieved two orders of magnitude space reduction when compared to Magic's non-compositional analysis [3]. Symmetric rules did consistently better than the non-symmetric one in this example.

Iteration	$\|A_i\|$	States	Transitions
1	1	294	1,548
2	2	269	1,560
3	3	**541**	**3,066**
4	5	12	69
5	6	474	2,706

Application of learning to the design of the K9 Rover Executive. Global state space: 3,630 states and 34,653 transitions. Largest state space computed by our approach: 541 states and 3,066 transitions (iteration 3). We achieve an order of magnitude space reduction.

System	States	Transitions	Memory	Time
Monolithic	183,132	425,641	952.85Mb	12m,24
Premise 1	53,215	117,756	255.96Mb	4m,49
Premise 2	13,884	20,601	118.97Mb	1m,16

Checking K9 code with JPF [12]. Use of design assumptions with the rule in Fig. 1 (a) yields a 3-fold space reduction.

4 Conclusions and Future Work

We presented a framework and its associated tool for learning-based assume-guarantee verification of software models and implementations. Our experience so far indicates that the approach has the potential of scaling to industrial size applications, especially when combined with abstraction.

Our tool is extensible: new assume-guarantee rules can be easily incorporated, and alternative tools for model extraction can be interfaced with it. Moreover, our framework is general; it relies on standard features of model checking, and could therefore be introduced in other model checking tools. For example, Magic has recently been extended to directly support learning-based assume guarantee reasoning [3]. We are also planning an implementation of our framework for the Spin model checker, in the context of a new NASA project.

Acknowledgements

We thank Howard Barringer and Jamieson Cobleigh for their contributions to our techniques and Sagar Chaki for helping with the Magic integration.

References

1. D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, Nov. 1987.
2. H. Barringer, D. Giannakopoulou, and C. S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proc. SAVCBS Workshop*, 2003.
3. S. Chaki, E. Clarke, D. Giannakopoulou, and C. Păsăreanu. Abstraction and assume-guarantee reasoning for automated software verification. *RIACS TR 05.02*, October 2004.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. *IEEE TSE*, 30(6):388–402, June 2004.
5. R. Chatley, S. Eisenbach, and J. Magee. Magicbeans: a Platform for Deploying Plugin Components. In *Proc. of Component Deployment (CD 2004).*

6. J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. of 9th TACAS*, pages 331–346, Apr. 2003.
7. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE'00.*
8. D. Giannakopoulou, C. S. Păsăreanu, and J. M. Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proc. of ICSE 2004.*
9. C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Trans. on Prog. Lang. and Sys.*, 5(4):596–619, Oct. 1983.
10. J. Magee and J. Kramer. *Concurrency: State Models & Java Programs.* John Wiley & Sons, 1999.
11. A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144, 1985.
12. W. Visser, K. Havelund, G. Brat, and S.-J. Park. Model checking programs. In *Proc. of the Fifteenth IEEE Int. Conf. on Auto. Soft. Eng.*, pages 3–12, Sept. 2000.