RESY: Requirement Synthesis for Compositional Model Checking*

Bernd Finkbeiner, Hans-Jörg Peter, and Sven Schewe

Universität des Saarlandes
66123 Saarbrücken, Germany
{finkbeiner,peter,schewe}@cs.uni-sb.de

Abstract. The requirement synthesis tool RESY automatically computes environment assumptions for compositional model checking. Given a process M in a multi-process PROMELA program, an abstraction refinement loop computes a coarse equivalence relation on the states of the environment, collapsing two states if the environment of M can either force the occurrence of an error from both states or from neither state. RESY supports three different operation modes: assumption generation, compositional model checking, and front-end to the model checker SPIN. In *assumption generation* mode, RESY minimizes the size of the assumption; small assumptions are useful for program documentation and as certificates for re-verification. In *compositional model checking* mode, RESY terminates as soon as the property is proven or disproven, independently of the size of the assumption. In *front-end* mode, RESY terminates when the size of the assumption falls below a specified threshold, and calls SPIN with the simplified verification problem.

1 Requirement Synthesis

RESY is a tool for the automatic synthesis of requirement automata for safety properties. Requirement automata represent the assumptions an environment makes on the behavior of a component. Typical applications include *program documentation* [1], where the synthesized requirements help the user to understand the interaction of the program components; *program certification* [2], where the synthesized requirements simplify the re-verification of the system (possibly by a different user and a different tool); and *compositional model checking* [3], where the requirement is synthesized and used during the *same* model checking run, in order to avoid the construction of the full product state space.

RESY implements the requirement synthesis algorithm presented in [4]. Given a system $M \| E$, which consists of a process M and its environment E, RESY computes an equivalence relation on the states of M, collapsing two states if E can either force the occurrence of an error from both states or from neither state.

* This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

C.R. Ramakrishnan and J. Rehof (Eds.): TACAS 2008, LNCS 4963, pp. 463–466, 2008.

The requirement automaton is the quotient of M with respect to the equivalence relation.

Key advantages of this approach are that the generated requirement automaton is small (RESY's equivalence is much coarser than language-based equivalences like bisimulation), inexpensive to compute (RESY is often dramatically faster than L*-based requirement learning), and easy to re-verify (implementation and requirement are related by a simple homomorphism).

2 Generating Requirements from Abstractions

Computing the equivalence relation requires two traversals of the state space. In a *forward* traversal, we identify states of the process M that are all either reachable or unreachable, depending on the state in the environment E they are combined with. In a *backward* traversal, we identify states of M that either all have or all do not have a path to the error, depending again on the state in E they are combined with.

To avoid the expansion of the full state graph, RESY considers abstractions of E. The abstractions are computed in an automatic abstraction refinement loop that, starting with the trivial abstraction, incrementally increases the size of the abstraction.

The abstraction $\mathcal{E}$ of the environment is a modal transition system that is defined by an equivalence relation $\simeq$ on E. Replacing E with its abstraction introduces the possibility that two states of M both lead to an error when composed with $\mathcal{E}$, but only one of them leads to an error when composed with E. RESY therefore distinguishes situations that *may* lead to an error (i.e., when the error is reached in the composition with $\mathcal{E}$ but not necessarily in E) from situations that *must* lead to an error (both in composition with $\mathcal{E}$ and in composition with E). Merging two states of M is safe in two cases: (1) if they *both must* lead to an error, and (2) if *neither* of them *may* lead to an error.

The environment abstraction identifies *must* and *may* transitions. In the backward analysis, for example, a transition $([v], a, [v'])$ is a *must* transition if, for all states $w \simeq v$, there is a state $w' \simeq v'$ such that (w, a, w') is a transition of E. Reachability on *must* transitions is a sufficient criterion for reachability in the concrete system; unreachability on *may* transitions is a sufficient criterion for unreachability in the concrete system.

In each abstraction refinement step, RESY uses a heuristic to pick some *may* transition $([v], \sigma, [v'])$ of the forward or backward analysis that is not also a *must* transition, and splits the equivalence class $[v']$ (respectively $[v]$), distinguishing states that either have or do not have the incoming (respectively outgoing) transition in E. By default, RESY picks forward and backward transitions that are closest to the initial state and the error, respectively.

RESY recognizes situations in which further refinements of the environment abstraction will no longer lead to a reduction of the requirement automaton. Depending on RESY's operation mode, the refinement loop may also be interrupted earlier, yielding a sound but not necessarily minimal requirement automaton.

3 Operation Modes of RESY

The input to RESY is a PROMELA program that specifies a distributed system as a parallel composition of modules, and a specification automaton for the safety property. RESY can be executed in the following modes:

- *Assumption generation.* In this mode, RESY minimizes the size of the requirement automaton. This mode is most useful if the automaton is to be used as a certificate.
- *Compositional model checking.* In this mode, RESY terminates as soon as the property is proven or disproven, independently of the size of the requirement automaton generated so far. This mode is most useful if RESY is to be used as a stand-alone model checker.
- *SPIN front-end.* In this mode, RESY also terminates if the size of the requirement automaton falls below a user-defined threshold (for example, 10% of the states of the program M). If the property has not been proven or disproven at this point, RESY replaces M with the requirement automaton and calls SPIN [5] with the modified PROMELA program.

4 Results

Table 1 shows the performance of RESY on a range of benchmarks, including the sliding window protocol (SW), an elevator controller (Elevator), a production cell (Prodcell), and an industrial document flow (workflow). For each benchmark, the table shows the time and memory usage of the assumption generation mode and compares the performance of the compositional model checking mode (CMC) and the SPIN front-end mode using a threshold of 10% (R10%) with the performance of the model checker SPIN alone (SPIN).

The *sliding window* protocol is parameterized by the buffer and window sizes. Property A, B, and C are valid properties (e.g., "the protocol does not invent messages"). Property D ("the receiver never produces any output") does not hold. The *elevator* benchmark is parameterized by the number of floors. (The property states that a door is never open when no elevator is present.) In the *production cell*, two concurrent programs control a plant. The benchmark is parameterized by the number of components of the plant (which may include robot arms, a press, lifts, and grippers). The property requires that there is no arm within the press when it starts working. The *workflow* benchmark models an industrial document flow. It is parameterized by the number of participants.

The results in Table 1 show that compositional model checking with RESY often improves over monolithic model checking with SPIN by more than an order of magnitude. Computing the minimal requirement automaton typically does not add significant cost. The minimal requirement automaton is always much smaller than the original process and often small enough to be presented to the user.

Availability. RESY and the benchmarks used in this paper are available online at `http://react.cs.uni-sb.de/resy`.

Table 1. Experimental results of RESY on a range of benchmarks. The table shows the performance of the *assumption generation* mode, and compares the performance of the verification modes *compositional model checking* (CMC) and SPIN *front-end* with a threshold of 10% (R10%) to the performance of SPIN. The time (t) and memory usage (m) is given in milliseconds and megabytes, respectively; the sizes of the process M, environment E, and requirement automaton $\mathcal{A}$ are given as the number of states. All benchmarks were measured on an Intel Pentium M processor 2.13 GHz.

	model size		assumption generation			verification					
						CMC		R10%		SPIN	
	M	E	t	m	$\mathcal{A}$	t	m	t	m	t	m
SW 2/1/A	48	26	86	0.4	3	86	0.3	86	0.4	1149	2.7
SW 3/1/A	256	120	2880	2.6	6	2877	2.4	3960	2.7	9017	3.9
SW 3/2/A	256	120	3882	2.7	5	3872	2.5	5366	2.8	9657	4.1
SW 2/1/B	48	26	86	0.4	28	49	0.3	86	0.4	1148	2.7
SW 3/1/B	256	120	3818	2.9	107	1722	1.9	3818	2.9	9050	3.9
SW 3/2/B	256	120	4979	3.0	209	598	1.7	4979	3.0	9613	4.1
SW 2/1/C	48	26	44	0.3	5	53	0.3	44	0.3	1148	2.7
SW 3/1/C	256	120	985	2.2	9	985	2.8	1958	2.7	9069	4.0
SW 3/2/C	256	120	1524	2.2	9	1523	3.0	2978	2.8	9640	4.1
SW 2/1/D	48	26	22	0.4	1	22	0.2	22	0.4	1148	2.7
SW 3/1/D	256	120	84	2.3	1	84	1.4	84	2.3	8890	3.6
SW 3/2/D	256	120	89	2.4	1	89	1.5	89	2.4	9406	3.7
Elevator 2	48	18	68	0.2	11	61	0.1	68	0.2	636	2.6
Elevator 3	192	30	414	1.0	15	407	0.5	414	1.0	860	2.6
Elevator 4	768	42	2633	4.9	22	2615	2.1	5113	4.9	1152	2.6
Prodcell 2	12	12	23	0.1	5	23	0.1	23	0.1	519	2.6
Prodcell 3	24	24	73	0.2	6	65	0.2	73	0.2	681	2.6
Prodcell 4	40	24	111	0.4	6	118	0.3	802	2.6	803	2.6
Prodcell 5	40	40	296	0.6	6	296	0.5	296	0.6	897	2.6
Prodcell 6	40	48	486	0.7	6	486	0.6	486	0.7	973	2.6
Prodcell 7	72	48	902	1.1	6	906	0.8	1831	2.6	1158	2.7
Workflow 2	64	11	35	0.2	4	31	0.1	35	0.2	526	2.6
Workflow 3	512	16	441	2.5	8	50	0.5	441	2.5	619	2.6
Workflow 4	4096	25	20294	62.9	16	409	4.1	20294	62.9	849	2.7

References

1. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Proc. ASE, pp. 3–12. IEEE Computer Society, Los Alamitos (2002)
2. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001)
3. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
4. Finkbeiner, B., Schewe, S., Brill, M.: Automatic synthesis of assumptions for compositional model checking. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 143–158. Springer, Heidelberg (2006)
5. Holzmann, G.: The Spin Model Checker, Primer and Reference Manual. Addison-Wesley, Reading (2003)