RATSY– A new Requirements Analysis Tool with Synthesis*

Roderick Bloem[1], Alessandro Cimatti[2], Karin Greimel[1], Georg Hofferek[1], Robert Könighofer[1], Marco Roveri[2], Viktor Schuppan[2], and Richard Seeber[1]

[1]Graz University of Technology, Austria
[2]Fondazione Bruno Kessler, Trento, Italy

Abstract. Formal specifications play an increasingly important role in system design-flows. Yet, they are not always easy to deal with. In this paper we present RATSY, a successor of the Requirements Analysis Tool RAT. RATSY extends RAT in several ways. First, it includes a new graphical user interface to specify system properties as simple Büchi word automata. Second, it can help debug incorrect specifications by means of a game-based approach. Third, it allows correct-by-construction synthesis of systems from their temporal properties. These new features and their seamless integration assist in property-based design processes.

1 Introduction

Several (recent) trends in designing and implementing complex digital systems necessitate the existence of a formal specification for the system at hand. For example, specifications can be used to unambiguously communicate design intents and interface assumptions between collaborating designers. They can also be used to formally verify implementations by means of a model checker. Moreover, a complete formal specification may be used to automatically synthesize an implementation using tools like Lily [6], Anzu [7], or as shown in [11]. Formal specifications are also created, sold, and used as third-party verification IPs [4].

For some of these use cases it is of interest to create and analyze the formal specification stand-alone, i.e., without a corresponding implementation, or before such an implementation is ready. The tool RAT [2] supports these tasks by allowing the user to write a specification in PSL syntax, to analyze it on a trace level, and to check if it is realizable, i.e., if a conforming system exists. However, RAT has some shortcomings when used for system design. Figure 1 depicts a typical property-based design flow. Some informal design intent is turned into a formal specification, which is then refined in several iterations involving simulation and debugging. Finally, an implementation is derived from the specification, ideally using correct-by-construction synthesis. The user faces several problems when putting this design flow into practice. First, it is hard to express the design intent in a formal language. Second, our experience shows

* This work was supported by EU grants 217069 (COCONUT) and 248613 (DIAMOND) as well as Provincia Autonoma di Trento grant EMTELOS.

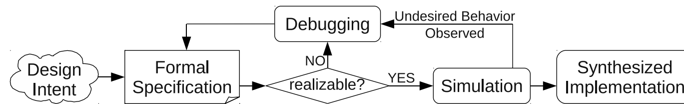

Fig. 1. A typical property-based design flow.

that formal specifications for complex designs are hardly ever written correctly on the first try. Thus, there is a need for proper support in debugging. Finally, it must be possible to synthesize an implementation from the specification.

We present the tool RATSY, an extension of RAT which provides several new features to assist the user in a property-based design flow. First, a graphical user interface for drawing Büchi automata has been added. These automata are an easy-to-understand way to specify system properties. Second, the debugging approach presented in [8] has been integrated. It aids in debugging unrealizable specifications and in refining specifications that allow undesired behavior. Third, synthesis functionality has been added. Finally, an additional realizability algorithm [10, 5] has been implemented, handling a strictly larger subset of PSL than the one in RAT [9]. Together with the analysis features inherited from RAT, this yields a powerful tool with full support for property-based design processes.

RATSY is available at `http://rat.fbk.eu/ratsy/`. The following sections will detail the improvements and new features of RATSY.

2 Automaton Editor

The automaton editor provides an intuitive interface to specify system properties as Büchi automata. The graphical representation makes creating and especially maintaining specifications easier and less error-prone. Automata are restricted to be deterministic and complete to allow for more efficient synthesis.[1] Completeness is ensured by providing an implicit "dead state" as the default destination of transitions. When transitions are added or changed, other transition conditions are updated by the tool to maintain determinism. Automata can be drawn once and instantiated multiple times, with template parameters allowing for different instantiations. For use with other features of RATSY, PSL formulas are generated automatically from the automata. Finally, the automata are used to visualize state information during simulation and debugging (see next section).

3 Simulation and Debugging

RATSY implements the ideas presented in [8] to test and debug formal specifications. First, it allows the user to test realizable specifications. An implementation is synthesized and the user can simulate it. Second, the tool provides an easy-to-use method to rule out undesired behavior observed during simulation. The user

[1] This is usually not a limiting factor since specifications used in practice tend to be in this class [9]; otherwise the designer can fall back to entering formulas in PSL.

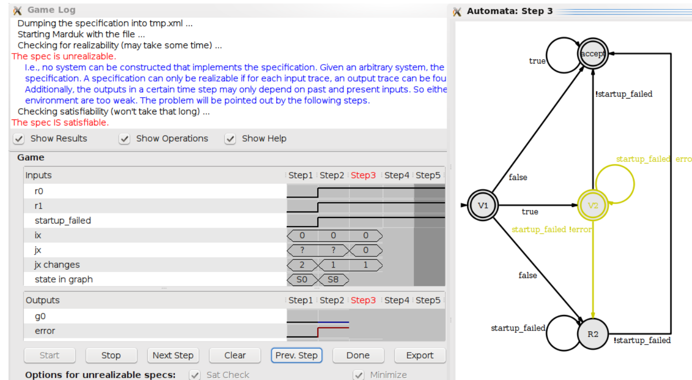

Fig. 2. A part of RATSY's GUI.

can simply modify an incorrect simulation trace to match the design intent. The tool then turns this modified trace into a property which enforces the desired response to the inputs, thereby eliminating the undesired behavior. Third, the user can debug unrealizable specifications. Following [8], the user plays a diagnostic game in the role of the system against a counterstrategy or a countertrace synthesized from an unrealizable core of the specification. The diagnostic game illustrates where the specification is too restrictive to be realizable.

Figure 2 shows a part of the screen when playing a diagnostic game to debug unrealizability. Testing a realizable specification looks similar. In every step of the game the counterstrategy determines values for the input signals, and the user sets values for the output signals in the game window or via the automata window. In every automaton of the specification, the current state, as well as transitions that can still be taken, are highlighted. The user can traverse a certain transition by clicking it. All restrictions on signal values associated with that transition are then applied. This integration with the automaton editor greatly increases the usability and helps the user to keep track of what is going on.

4 Technical Aspects

RATSY itself is implemented in Python. The symbolic algorithms rely on CUDD [12] and NuSMV [3], which are accessed through a SWIG-generated [1] wrapper. The synthesis functionality is based on a Python reimplementation of Anzu [7] with some minor implementation-specific improvements. The synthesis algorithm [9] handles specifications given in *Generalized Reactivity (1)* (GR(1)) format. By means of the NuSMV parser, RATSY can perform several syntactic transformations on its own in order to turn a specification into the required format. Furthermore, the NuSMV library automatically encodes multi-valued variables to Boolean signals. RATSY generates circuits in BLIF and Verilog format. If syntactic transformation into GR(1) fails, but succeeds into LTL, then a preliminary implementation of an algorithm along the lines of [10, 5] can be used

to determine realizability; debugging and synthesis are not yet available in that case. The conversion into non-deterministic Büchi automata required by [10, 5] is performed via a (slightly adapted) version of Wring [13] from Lily [6]. RATSY performs similar to Anzu [7, 8] when operating on GR(1) specifications, and can decide most of the examples that Acacia [5] can for full LTL.

5 Conclusions and Future Work

RATSY enhances the analysis features of RAT with a game-based debugging approach for specifications. Furthermore, it eases specifying properties by representing them as Büchi automata, which can be edited via a graphical user interface. Once the user is satisfied with the result of debugging and analyzing her specification, she can synthesize an implementation with just a few clicks. All the new features integrate seamlessly with the well-established analysis features of RAT. Thus, RATSY is a powerful tool to support property-based design.

In the future, we plan to implement a wider variety of output formats for synthesis. Furthermore, we will continue work on improving the size of the synthesized circuits, as well as the time needed to perform synthesis. Concerning debugging, we plan to combine the already implemented approach with model-based diagnosis techniques. This should further simplify the localization of errors.

References

1. D.M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proc. 4th USENIX Tcl/Tk Workshop*, pages 129–139, 1996.
2. R. Bloem, R. Cavada, I. Pill, M. Roveri, and A. Tchaltsev. Rat: A tool for the formal analysis of requirements. In *CAV*, pages 263–267, 2007.
3. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *CAV*, pages 359–364, 2002.
4. S. Dellacherie. Automatic bus-protocol verification using assertions. In *GSPx'04*.
5. E. Filiot, N. Jin, and J. Raskin. An antichain algorithm for LTL realizability. In *CAV*, pages 263–277, 2009.
6. B. Jobstmann and R. Bloem. Optimizations for LTL synthesis. In *FMCAD*, pages 117–124, 2006.
7. B. Jobstmann, S. Galler, M. Weiglhofer, and R. Bloem. Anzu: A tool for property synthesis. In *CAV*, pages 258–262, 2007.
8. R. Könighofer, G. Hofferek, and R. Bloem. Debugging formal specifications using simple counterstrategies. In *FMCAD*, pages 152–159, 2009.
9. N. Piterman, A. Pnueli, and Y. Sa´ar. Synthesis of reactive(1) designs. In *VMCAI*, pages 364–380. Springer, 2006. LNCS 3855.
10. S. Schewe and B. Finkbeiner. Bounded synthesis. In *ATVA*, pages 474–488, 2007.
11. S. Sohail and F. Somenzi. Safety first: A two-stage algorithm for LTL games. In *FMCAD*, pages 77–84, 2009.
12. F. Somenzi. *CUDD: CU Decision Diagram Package.* University of Colorado at Boulder, `ftp://vlsi.colorado.edu/pub/`.
13. F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *CAV*, pages 248–263, 2000.