Science of Computer Programming 83 (2014) 56–79

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Synthesis of hierarchical systems

Benjamin Aminof [a], Fabio Mogavero [b,*], Aniello Murano [b]

[a] Hebrew University, Jerusalem 91904, Israel
[b] Università degli Studi di Napoli "Federico II", 80126 Napoli, Italy

HIGHLIGHTS

- We describe how to synthesize a hierarchical system from a library of components.
- We propose a procedure that in rounds given a specification returns a correct system.
- We manage both branching-time (mu-Calculus) and linear-time specification languages.
- The resulting system is hierarchical and reuses components previously synthesized.
- The obtained complexity is not worst than that of classic flat synthesis procedure.

ARTICLE INFO

Article history:
Received 4 May 2012
Received in revised form 1 July 2013
Accepted 2 July 2013
Available online 23 July 2013

Keywords:
Hierarchical systems
mu-Calculus
Temporal logics
Parity games
Synthesis

ABSTRACT

In automated synthesis, given a specification, we automatically create a system that is guaranteed to satisfy the specification. In the classical temporal synthesis algorithms, one usually creates a "flat" system "from scratch". However, real-life software and hardware systems are usually created using preexisting libraries of reusable components, and are not "flat" since repeated sub-systems are described only once.

In this work we describe an algorithm for the synthesis of a hierarchical system from a library of hierarchical components, which follows the "bottom-up" approach to system design. Our algorithm works by synthesizing in many rounds, when at each round the system designer provides the specification of the currently desired module, which is then automatically synthesized using the initial library and the previously constructed modules. To ensure that the synthesized module actually takes advantage of the available high-level modules, we guide the algorithm by enforcing certain modularity criteria.

We show that the synthesis of a hierarchical system from a library of hierarchical components is EXPTIME-complete for μ-calculus, and 2EXPTIME-complete for LTL, both in the cases of complete and incomplete information. Thus, in all cases, it is not harder than the classical synthesis problem (of synthesizing flat systems "from scratch"), even though the synthesized hierarchical system may be exponentially smaller than a flat one.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

In formal verification and design, *synthesis* is the automated construction of a system from its specification. The basic idea is simple and appealing: instead of developing a system and then verifying that it is correct w.r.t. its specification, we use an automated procedure that, given a specification, builds a system that is correct by construction.

The first formulation of synthesis goes back to Church [20]. Later works on synthesis considered first *closed systems*, where the system is extracted from a constructive proof that the specification is satisfiable [28,41]. In the late 1980s, Pnueli

* Corresponding author.
E-mail address: fm@fabiomogavero.com (F. Mogavero).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.07.001

and Rosner [47] realized that such a synthesis paradigm is not of much interest when applied to *open systems* [31] (also called *reactive systems* [16,37]). Differently from closed systems, an open system interacts with an external environment and its correctness depends on whether it satisfies the specification with respect to all allowable environments. If we apply the techniques of [28,41] to open systems, we obtain a system that is correct only with respect to some specific environments. In [47], Pnueli and Rosner argued that the right way to approach synthesis of open systems is to consider the framework as a possibly infinite game between the environment and the system. A correct system can be then viewed as a winning strategy in this game, and synthesizing a system amounts to finding such a strategy.

The Pnueli and Rosner idea can be summarized as follows. Given sets Σ_I and Σ_O of inputs and outputs, respectively (usually, $\Sigma_I = 2^I$ and $\Sigma_O = 2^O$, where I is a set of input signals supplied by the environment and O is a set of output signals), one can view a system as a strategy $P : \Sigma_I^* \to \Sigma_O$ that maps a finite sequence of sets of input signals (i.e., the history of the actions of the environment so far) into a set of current output signals. When P interacts with an environment that generates infinite input sequences, it associates with each input sequence an infinite computation over $\Sigma_I \cup \Sigma_O$. Though the system P is deterministic, it induces a *computation tree*. The branches of the tree correspond to external nondeterminism, caused by different possible inputs. Thus, the tree has a fixed branching degree $|\Sigma_I|$, and it embodies all the possible inputs (and hence also computations) of P. When we synthesize P from a linear temporal logic formula φ we require φ to hold in all the paths of P's computation tree. However, in order to impose possibility requirements on P we have to use a branching-time logic like μ-calculus. Given a branching specification φ over $\Sigma_I \cup \Sigma_O$, realizability of φ is the problem of determining whether there exists a system P whose computation tree satisfies φ. Correct synthesis of φ then amounts to constructing such a P. The above synthesis problem for linear-time temporal logic (LTL) specifications was addressed in [47], and for μ-calculus specifications in [26]. In both cases, the traditional algorithm for finding the desired P works by constructing an appropriate *computation tree-automaton* that accepts trees that satisfy the specification formula, and then looking for a finitely-representable witness to the non-emptiness of this automaton. Such a witness can be easily viewed as a finite-state system P realizing the specification.

In spite of the rich theory developed for system synthesis in the last two decades, little of this theory has been lifted to practice. In fact, apart from very recent results [14,24,51], the main classical approaches to tackle synthesis in practice are either to use heuristics (e.g., [30]) or to restrict to simple specifications (e.g., [44]). Some people argue that this is because the synthesis problem is very expensive compared to model-checking [36]. There is, however, something misleading in this perception: while the complexity of synthesis is given with respect to the specification only, the complexity of model-checking is given also with respect to a program, which can be very large. A common thread in almost all of the works concerning synthesis is the assumption that the system is to be built from "scratch". Obviously, real-world systems are rarely constructed this way, but rather by utilizing many preexisting reusable components, i.e., a *library*. Using standard preexisting components is sometimes unavoidable (for example, access to hardware resources is usually under the control of the operating system, which must be "reused"), and many times has other benefits (apart from saving time and effort, which may seem to be less of a problem in a setting of automatic – as opposed to manual – synthesis), such as maintaining a common code base, and abstracting away low level details that are already handled by the preexisting components. Another reason that may account for the limited use of synthesis in practice is that many designers find it extremely difficult and/or unnatural to write a complex specification in temporal logic. Indeed, a very common practice in the hardware industry is to consider a model of the desired hardware written in a high level programming language like ANSI-C to be a specification (a.k.a. "golden model") [45]. Moreover, even if a specification is written in temporal logic, the synthesized system is usually monolithic and looks very unnatural from the system designer's point of view. Indeed, in classical temporal synthesis algorithms one usually creates in one step a "flat" system, i.e., a system in which sub-systems may be repeated many times. On the contrary, real-life software and hardware systems are built step by step and are hierarchical (or even recursive) having repeated sub-systems (such as sub-routines) described only once. While hierarchical systems may be exponentially more succinct than flat ones, it has been shown that the cost of solving questions about them (like model-checking) are in many cases not exponentially higher [6,7,29]. Hierarchical systems can also be seen as a special case of recursive systems [3,4], where the nesting of calls to sub-systems is bounded. However, having no bound on the nesting of calls gives rise to infinite-state systems, and this may result in a higher complexity, especially when we have a bound that is small with respect to the rest of the system.

Consider for example the problem of synthesizing a 60-minutes chronograph displaying elapsed time in minutes and seconds. A naive solution, which is the one created by a traditional synthesis approach, is to create a transducer that contains at least 3600 explicit states, one for each successive clock signal. However, an alternative is to design a hierarchical transducer, composed of two different machines: one counts from 0 to 59 minutes (the *minutes-machine*), and the other counts from 0 to 59 seconds (the *seconds-machine*) – see Example 3.1 for a detailed description. In particular, by means of 60 special states, named *boxes* or *super-states*, the minutes-machine calls 60 times the seconds-machine. This hierarchical machine is arguably more natural, and it is definitely more succinct since it has an order of magnitude less states and boxes than the flat one with 3600 states. Once the 60-minutes chronograph design process has been completed the resulting transducer can be added to a library of components for future use, for example in a 24-hours chronograph. Then, it is enough to build a new machine (*hours-machine*) that counts from 0 to 24 hours using boxes to call the 60-minutes transducer found in the library.

In this work, we provide a uniform algorithm, for different temporal logics, for the synthesis of a hierarchical system from a library of hierarchical systems, which mimics the "bottom-up" approach to system design, where one builds a system

Download English Version:

https://daneshyari.com/en/article/433368

Download Persian Version:

https://daneshyari.com/article/433368

Daneshyari.com