Synthesis from Component Libraries with Costs

Guy Avni and Orna Kupferman

School of Computer Science and Engineering, The Hebrew University, Israel

Abstract. *Synthesis* is the automated construction of a system from its specification. In real life, hardware and software systems are rarely constructed from scratch. Rather, a system is typically constructed from a library of components. Lustig and Vardi formalized this intuition and studied LTL synthesis from component libraries. In real life, designers seek optimal systems. In this paper we add optimality considerations to the setting. We distinguish between quality considerations (for example, size – the smaller a system is, the better it is), and pricing (for example, the payment to the company who manufactured the component). We study the problem of designing systems with minimal quality-cost and price. A key point is that while the quality cost is individual – the choices of a designer are independent of choices made by other designers that use the same library, pricing gives rise to a resource-allocation game – designers that use the same component share its price, with the share being proportional to the number of uses (a component can be used several times in a design). We study both closed and open settings, and in both we solve the problem of finding an optimal design. In a setting with multiple designers, we also study the game-theoretic problems of the induced resource-allocation game.

1 Introduction

Synthesis is the automated construction of a system from its specification. The classical approach to synthesis is to extract a system from a proof that the specification is satisfiable. In the late 1980s, researchers realized that the classical approach to synthesis is well suited to *closed* systems, but not to *open* (also called *reactive*) systems [1,24]. A reactive system interacts with its environment, and a correct system should have a *strategy* to satisfy the specification with respect to all environments. It turns out that the existence of such a strategy is stronger than satisfiability, and is termed *reliability*.

In spite of the rich theory developed for synthesis, in both the closed and open settings, little of this theory has been reduced to practice. This is in contrast with verification algorithms, which are extensively applied in practice. We distinguish between algorithmic and conceptual reasons for the little impact of synthesis in practice. The algorithmic reasons include the high complexity of the synthesis problem (PSPACE-complete in the closed setting [28] and 2EXPTIME-complete in the open setting [24], for specifications in LTL) as well as the intricacy of the algorithms in the open setting – the traditional approach involves determinization of automata on infinite words [27] and a solution of parity games [19].

We find the argument about the algorithmic challenge less compelling. First, experience with verification shows that even nonelementary algorithms can be practical,

since the worst-case complexity does not arise often. For example, while the model-checking problem for specifications in second-order logic has nonelementary complexity, the model-checking tool MONA [14] successfully verifies many specifications given in second-order logic. Furthermore, in some sense, synthesis is not harder than verification: the complexity of synthesis is given with respect to the specification only, whereas the complexity of verification is given with respect to the specification and the system, which is typically much larger than the specification. About the intercity of the algorithms, in the last decade we have seen quite many alternatives to the traditional approach – Safraless algorithms that avoid determinization and parity games, and reduce synthesis to problems that are simpler and are amenable to optimizations and symbolic implementations [17,21,22].

The arguments about the conceptual and methodological reasons are more compelling. We see here three main challenges, relevant in both the closed and open settings. First, unlike verification, where a specification can be decomposed into sub-specifications, each can be checked independently, in synthesis the starting point is one comprehensive specification. This inability to decompose or evolve the specification is related to the second challenge. In practice, we rarely construct systems from scratch or from one comprehensive specification. Rather, systems are constructed from existing components. This is true for both hardware systems, where we see IP cores or design libraries, and software systems, where web APIs and libraries of functions and objects are common. Third, while in verification we only automate the check of the system, automating its design is by far more risky and unpredictable – there are typically many ways to satisfy a satisfiable or realizable specification, and designers will be willing to give up manual design only if they can count on the automated synthesis tool to construct systems of comparable quality. Traditional synthesis algorithms do not attempt to address the quality issue.

In this paper we continue earlier efforts to cope with the above conceptual challenges. Our contribution extends both the setting and the results of earlier work. The realization that design of systems proceeds by composition of underlying components is not new to the verification community. For example, [18] proposed a framework for component-based modelling that uses an abstract layered model of components, and [12] initiated a series of works on interface theories for component-based design, possibly with a reuse of components in a library [13]. The need to consider components is more evident in the context of software, where, for example, recursion is possible, so components have to be equipped with mechanisms for call and return [4]. The setting and technical details, however, are different from these in the synthesis problem we consider here. The closer to our work here is [23], which studied LTL synthesis from reusable component libraries. Lustig and Vardi studied two notions of component composition. In the first notion, termed data-flow composition, components are cascaded so that the outputs of one component are fed to other components. In the second notion, termed control-flow composition, the composition is flat and control flows among the different components. The second notion, which turns out to be the decidable one [23], is particularly suitable for modelling web-service orchestration, where users are typically offered services and interact with different parties [3].

Let us turn now to the quality issue. Traditional formal methods are based on a Boolean satisfaction notion: a system satisfies, or not, a given specification. The richness of today's systems, however, calls for specification formalisms that are *multi-valued*. The multi-valued setting arises directly in probabilistic and weighted systems and arises indirectly in applications where multi-valued satisfaction is used in order to model quantitative properties of the system like its size, security level, or quality. Reasoning about quantitative properties of systems is an active area of research in recent years, yielding quantitative specification formalisms and algorithms [11,16,10,2,9]. In quantitative reasoning, the Boolean satisfaction notion is refined and one can talk about the cost, or reward, of using a system, or, in our component-based setting, the cost of using a component from the library.

In order to capture a wide set of scenarios in practice, we associate with each component in the library two costs: a *quality cost* and a *construction cost*. The quality cost, as describes above, concerns the performance of the component and is paid each time the component is used. The construction cost is the cost of adding the component to the library. Thus, a design that uses a component pays its construction cost once. When several designs use the same component, they share its construction cost. This corresponds to real-life scenarios, where users pay, for example, for web-services, and indeed their price is influenced by the market demand.

We study synthesis from component libraries with costs in the closed and open settings. In both settings, the specification is given by means of a deterministic automaton $\mathcal{S}$ on finite words (DFA).[1] In the closed setting, the specification is a regular language over some alphabet Σ and the library consists of box-DFAs (that is, DFAs with exit states) over Σ. In the open setting, the specification $\mathcal{S}$ is over sets I and O of input and output signals, and the library consists of box-I/O-transducers. The boxes are black, in the sense that a design that uses components from the library does not see Σ (or $I \cup O$) nor it sees the behaviour inside the components. Rather, the mode of operation is as in the control-flow composition of [23]: the design gives control to one of the components in the library. It then sees only the exit state through which the component completes its computation and relinquishes control. Based on this information, the design decides which component gets control next, and so on.

In more technical details, the synthesis problem gets as input the specification $\mathcal{S}$ as well as a library $\mathcal{L}$ of components $\mathcal{B}_1, \ldots, \mathcal{B}_n$. The goal is to return a correct design – a transducer $\mathcal{D}$ that reads the exit states of the components and outputs the next component to gain control. In the closed setting, correctness means that the language over Σ that is generated by the composition defined by $\mathcal{D}$ is equal to the language of $\mathcal{S}$. In the open setting, correctness means that the interaction of the composition defined by $\mathcal{D}$ with all input sequences generates a computation over $I \cup O$ that is in the language of $\mathcal{S}$.

We first study the problem without cost and reduce it to the solution of a two-player safety game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$. In the closed setting, the game is of full information and the problem

[1] It is possible to extend our results to specifications in LTL. We prefer to work with deterministic automata, as this setting isolates the complexity and technical challenges of the design problem and avoids the domination of the doubly-exponential complexity of going from LTL to deterministic automata.

can be solved in polynomial time. In the open setting, the flexibility that the design have in responding to different input sequences introduces partial information to the game, and the problem is EXPTIME-complete. We note that in [23], where the open setting was studied and the specification is given by means of an LTL formula, the complexity is 2EXPTIME-complete, thus one could have expected our complexity to be only polynomial. We prove, however, hardness in EXPTIME, showing that it is not just the need to transfer the LTL formula to a deterministic formalism that leads to the high complexity.

We then turn to integrate cost to the story. As explained above, there are two types of costs associated with each component $\mathcal{B}_i$ in $\mathcal{L}$. The first type, quality cost, can be studied for each design in isolation. We show that even there, the combinatorial setting is not simple. While for the closed setting an optimal design can be induced from a memoryless strategy of the designer in the game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$, making the problem of finding an optimal design NP-complete, seeking designs of optimal cost may require sophisticated compositions in the open setting. In particular, we show that optimal designs may be exponentially larger than other correct designs[2], and that an optimal design may not be induced by a memoryless strategy in $\mathcal{G}_{\mathcal{L},\mathcal{S}}$. We are still able to bound the size of an optimal transducer by the size of $\mathcal{G}_{\mathcal{L},\mathcal{S}}$, and show that the optimal synthesis problem is NEXPTIME-complete.

The second type of cost, namely construction cost, depends not only on choices made by the designer, but also on choices made by designers of other specifications that use the library. Indeed, recall that the construction cost of a component is shared by designers that use this component, with the share being proportional to the number of uses (a component can be used several times in a design). Hence, the setting gives rise to a *resource-allocation game* [26,15]. Unlike traditional resource-allocation games, where players' strategies are sets of resources, here each strategy is a multiset – the components a designer needs. As has been the case in [7], the setting of multisets makes the game less stable. We show that the game is not guaranteed to have a *Nash Equilibrium* (NE), and that the problem of deciding whether an NE exists is Σ_2^P-complete. We then turn to the more algorithmic related problems and show that the problems of finding an optimal design given the choices of the other designers (a.k.a. the *best-response* problem, in algorithmic game theory) and of finding designs that minimize the total cost for all specifications (a.k.a. the *social optimum*) are both NP-complete.

Due to lack of space, many proofs and examples are missing in this version. They can be found in the full version, in the authors' URLs.

2 Preliminaries

Automata, transducers, and boxes A *deterministic finite automaton* (DFA, for short) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is an alphabet, Q is a set of states, $\delta : Q \times \Sigma \to Q$ is a partial transition function, $q_0 \in Q$ is an initial states, and $F \subseteq Q$ is a set of accepting states. We extend δ to words in an expected way, thus $\delta^* : Q \times \Sigma^* \to Q$ is such that for $q \in Q$, we have $\delta^*(q, \epsilon) = q$ and for $w \in \Sigma^*$ and $\sigma \in \Sigma$, we have $\delta^*(q, w \cdot \sigma) = \delta(\delta^*(q, w), \sigma)$. When $q = q_0$, we sometimes omit it, thus $\delta^*(w)$ is the

[2] Recall that "optimal" here refers to the quality-cost function.

state that $\mathcal{A}$ reaches after reading w. We assume that all states are reachable from q_0, thus for every $q \in Q$ there exists a word $w \in \Sigma^*$ such that $\delta^*(w) = q$. We refer to the *size* of $\mathcal{A}$, denoted $|\mathcal{A}|$, as the number of its states.

The *run* of $\mathcal{A}$ on a word $w = w_1, \ldots w_n \in \Sigma^*$ is the sequence of states $r = r_0, r_1, \ldots, r_n$ such that $r_0 = q_0$ and for every $0 \leq i \leq n-1$ we have $r_{i+1} = \delta(r_i, w_{i+1})$. The run r is accepting iff $r_n \in F$. The *language* of $\mathcal{A}$, denoted $L(\mathcal{A})$, is the set of words $w \in \Sigma^*$ such that the run of $\mathcal{A}$ on w is accepting, or, equivalently, $\delta^*(w) \in F$. For $q \in Q$, we denote by $L(\mathcal{A}^q)$ the language of the DFA that is the same as $\mathcal{A}$ only with initial state q.

A *transducer* models an interaction between a system and its environment. It is similar to a DFA except that in addition to Σ, which is referred to as the input alphabet, denoted Σ_I, there is an output alphabet, denoted Σ_O, and rather than being classified to accepting or rejecting, each state is labeled by a letter from Σ_O[3]. Formally, a transducer is a tuple $\mathcal{T} = \langle \Sigma_I, \Sigma_O, Q, q_0, \delta, \nu \rangle$, where Σ_I is an input alphabet, Σ_O is an output alphabet, Q, q_0, and $\delta : Q \times \Sigma_I \to Q$ are as in a DFA, and $\nu : Q \to \Sigma_O$ is an output function. We require $\mathcal{T}$ to be *receptive*. That is, δ is complete, so for every input word $w \in \Sigma_I^*$, there is a run of $\mathcal{T}$ on w. Consider an input word $w = w_1, \ldots, w_n \in \Sigma_I^*$. Let $r = r_0, \ldots, r_n$ be the run of $\mathcal{T}$ on w. The *computation of $\mathcal{T}$ in w* is then $\sigma_1, \ldots, \sigma_n \in (\Sigma_I \times \Sigma_O)^*$, where for $1 \leq i \leq n$, we have $\sigma_i = \langle w_i, \nu(r_{i-1}) \rangle$. We define the language of $\mathcal{T}$, denoted $L(\mathcal{T})$, as the set of all its computations. For a specification $L \subseteq (\Sigma_I \times \Sigma_O)^*$, we say that $\mathcal{T}$ *realizes* L iff $L(\mathcal{T}) \subseteq L$. Thus, no matter what the input sequence is, the interaction of $\mathcal{T}$ with the environment generates a computation that satisfies the specification.

By adding *exit states* to DFAs and transducers, we can view them as components from which we can compose systems. Formally, we consider two types of components. Closed components are modeled by *box-DFAs* and open components are modeled by *box-transducers*. A box-DFA augments a DFA by a set of exit states. Thus, a box-DFA is a tuple $\langle \Sigma, Q, \delta, q_0, F, E \rangle$, where $E \subseteq Q$ is a nonempty set of exit states. There are no outgoing transitions from an exit state. Also, the initial state cannot be an exit state and exit states are not accepting. Thus, $q_0 \notin E$ and $F \cap E = \emptyset$. Box-transducers are defined similarly, and their exit states are not labeled, thus $\nu : Q \setminus E \to \Sigma_O$.

Component libraries A *component library* is a collection of boxes $\mathcal{L} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$. We say that $\mathcal{L}$ is a *closed library* if the boxes are box-DFAs, and is an *open library* if the boxes are box-transducers. Let $[n] = \{1, \ldots, n\}$. In the first case, for $i \in [n]$, let $\mathcal{B}_i = \langle \Sigma, C_i, \delta_i, c_i^0, F_i, E_i \rangle$. In the second case, $\mathcal{B}_i = \langle \Sigma_I, \Sigma_O, C_i, \delta_i, c_i^0, \nu_i, E_i \rangle$. Note that all boxes in $\mathcal{L}$ share the same alphabet (input and output alphabet, in the case of transducers). We assume that the states of the components are disjoint, thus for every $i \neq j \in [n]$, we have $C_i \cap C_j = \emptyset$. We use the following abbreviations $\mathcal{C} = \bigcup_{i \in [n]} C_i$, $\mathcal{C}_0 = \bigcup_{i \in [n]} \{c_i^0\}$, $\mathcal{F} = \bigcup_{i \in [n]} F_i$, and $\mathcal{E} = \bigcup_{i \in [n]} E_i$. We define the *size* of $\mathcal{L}$ as $|\mathcal{C}|$.

We start by describing the intuition for composition of closed libraries. A *design* is a recipe to compose the components of a library $\mathcal{L}$ (allowing multiple uses) into a DFA. A run of the design on a word starts in an initial state of one of the components in $\mathcal{L}$. We say that this component has the initial *control*. When a component is in control, the

[3] These transducers are sometimes referred to as *Moore machines*.

run uses its states, follows its transition function, and if the run ends, it is accepting iff it ends in one of the components' accepting states. A component relinquishes control when the run reaches one of its exit states. It is then the design's duty to assign control to the next component, which gains control through its initial state.

Formally, a design is a transducer $\mathcal{D}$ with input alphabet $\mathcal{E}$ and output alphabet $[n]$. We can think of $\mathcal{D}$ as running beside the components. When a component reaches an exit state e, then $\mathcal{D}$ reads the input letter e, proceeds to its next state, and outputs the index of the component to gain control next. Note that $\mathcal{D}$ does not read the alphabet Σ and has no information about the states that the component visits. It only sees which exit state has been reached.

Consider a design $\mathcal{D} = \langle \mathcal{E}, [n], D, \delta, d^0, \nu \rangle$ and a closed library $\mathcal{L}$. We formalize the behavior of $\mathcal{D}$ by means of the *composition DFA* $\mathcal{A}_{\mathcal{L},\mathcal{D}}$ that simulates the run of $\mathcal{D}$ along with the runs of the box-DFAs. Formally, $\mathcal{A}_{\mathcal{L},\mathcal{D}} = \langle \Sigma, Q_{\mathcal{L},\mathcal{D}}, \delta_{\mathcal{L},\mathcal{D}}, q^0_{\mathcal{L},\mathcal{D}}, F_{\mathcal{L},\mathcal{D}} \rangle$ is defined as follows. The set of states $Q_{\mathcal{L},\mathcal{D}} \subseteq (\mathcal{C} \setminus \mathcal{E}) \times D$ consists of pairs of a *component state* from $\mathcal{C}$ and an *design state* from S. The component states are consistent with ν, thus $Q_{\mathcal{L},\mathcal{D}} = \bigcup_{i \in [n]} (C_i \setminus E_i) \times \{q : \nu(q) = i\}$. In exit states, the composition immediately moves to the initial state of the next component, which is why the component states of $\mathcal{A}_{\mathcal{L},\mathcal{D}}$ do not include $\mathcal{E}$. Consider a state $\langle c, q \rangle \in Q_{\mathcal{L},\mathcal{D}}$ and a letter $\sigma \in \Sigma$. Let $i \in [n]$ be such that $c \in C_i$. When a run of $\mathcal{A}_{\mathcal{L},\mathcal{D}}$ reaches the state $\langle c, q \rangle$, the component $\mathcal{B}_i$ is in control. Recall that c is not an exit state. Let $c' = \delta_i(c, \sigma)$. If $c' \notin E_i$, then $\mathcal{B}_i$ does not relinquish control after reading σ and $\delta_{\mathcal{L},\mathcal{D}}(\langle c, q \rangle, \sigma) = \langle c', q \rangle$. If $c' \in E_i$, then $\mathcal{B}_i$ relinquishes control through c', and it is the design's task to choose the next component to gain control. Let $q' = \delta(q, c')$ and let $j = \nu(q')$. Then, $\mathcal{B}_j$ is the next component to gain control (possibly $j = i$). Accordingly, we advance $\mathcal{D}$ to q' and continue to the initial state of $\mathcal{B}_j$. Formally, $\delta_{\mathcal{L},\mathcal{D}}(\langle c, q \rangle, \sigma) = \langle c^0_j, q' \rangle$. (Recall that $c^0_j \notin E_j$, so the new state is in $Q_{\mathcal{L},\mathcal{D}}$.) Note also that a visit in c' is skipped. The component that gains initial control is chosen according to $\nu(d^0)$. Thus, $q^0_{\mathcal{L},\mathcal{D}} = \langle c^0_j, d^0 \rangle$, where $j = \nu(d^0)$. Finally, the accepting states of $\mathcal{A}_{\mathcal{L},\mathcal{D}}$ are these in which the component state is accepting, thus $F_{\mathcal{L},\mathcal{D}} = \mathcal{F} \times D$.

The definition of a composition for an open library is similar. There, the composition is a transducer $\mathcal{T}_{\mathcal{L},\mathcal{D}} = \langle \Sigma_I, \Sigma_O, Q_{\mathcal{L},\mathcal{D}}, \delta_{\mathcal{L},\mathcal{D}}, q^0_{\mathcal{L},\mathcal{D}}, \nu_{\mathcal{L},\mathcal{D}} \rangle$, where $Q_{\mathcal{L},\mathcal{D}}$, $q^0_{\mathcal{L},\mathcal{D}}$, and $\delta_{\mathcal{L},\mathcal{D}}$ are as in the closed setting, except that $\delta_{\mathcal{L},\mathcal{D}}$ reads letters in Σ_I, and $\nu_{\mathcal{L},\mathcal{D}}(\langle c, q \rangle) = \nu_i(c)$, for $i \in [n]$ such that $c \in C_i$.

3 The Design Problem

The *design problem* gets as input a component library $\mathcal{L}$ and a specification that is given by means of a DFA $\mathcal{S}$. The problem is to decide whether there exists a correct design for $\mathcal{S}$ using the components in $\mathcal{L}$. In the closed setting, a design $\mathcal{D}$ is correct if $L(\mathcal{A}_{\mathcal{L},\mathcal{D}}) = L(\mathcal{S})$. In the open setting, $\mathcal{D}$ is correct if the transducer $\mathcal{T}_{\mathcal{L},\mathcal{D}}$ realizes $\mathcal{S}$. Our solution to the design problem reduces it to the problem of finding the winner in a turn-based two-player game, defined below.

A *turn-based two-player game* is played on an arena $\langle V, \Delta, V_0, \alpha \rangle$, where $V = V_1 \cup V_2$ is a set of vertices that are partitioned between Player 1 and Player 2, $\Delta \subseteq V \times V$ is a set of directed edges, $V_0 \subseteq V$ is a set of initial vertices, and α is an objective for Player 1, specifying a subset of V^ω. We consider here *safety games*, where $\alpha \subseteq V$ is a

set of vertices that are *safe* for Player 1. The game is played as follows. Initially, Player 1 places a token on one of the vertices in V_0. Assume the token is placed on a vertex $v \in V$ at the beginning of a round. The player that owns v is the player that moves the token to the next vertex, where the legal vertices to continue to are $\{v' \in V : \langle v, v' \rangle \in \Delta\}$. The outcome of the game is a *play* $\pi \in V^\omega$. The play is winning for Player 1 if for every $i \geq 1$, we have $\pi_i \in \alpha$. Otherwise, Player 2 wins.

A *strategy* for Player i, for $i \in \{1, 2\}$, is a recipe that, given a prefix of a play, tells the player what his next move should be. Thus, it is a function $f_i : V^* \cdot V_i \to V$ such that for every play $\pi \cdot v \in V^*$ with $v \in V_i$, we have $\langle v, f_i(\pi \cdot v) \rangle \in \Delta$. Since Player 1 moves first, we require that $f_1(\epsilon)$ is defined and is in V_0. For strategies f_1 and f_2 for players 1 and 2 respectively, the play $out(f_1, f_2) \in V^\omega$ is the unique play that is the outcome the game when the players follow their strategies. A strategy f_i for Player i is *memoryless* if it depends only in the current vertex, thus it is a function $f_i : V_i \to V$.

A strategy is *winning* for a player if by using it he wins against every strategy of the other player. Formally, a strategy f_1 is winning for Player 1 iff for every strategy f_2 for Player 2, Player 1 wins the play $out(f_1, f_2)$. The definition for Player 2 is dual. It is well known that safety games are *determined*, namely, exactly one player has a winning strategy, and admits *memoryless* strategies, namely, Player i has a winning strategy iff he has a memoryless winning strategy. Deciding the winner of a safety game can done in linear time.

Solving the design problem We describe the intuition of our solution for the design problems. Given a library $\mathcal{L}$ and a specification $\mathcal{S}$ we construct a safety game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ such that Player 1 wins $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ iff there is a correct design for $\mathcal{S}$ using the components in $\mathcal{L}$. Intuitively, Player 1's goal is to construct a correct design, thus he chooses the components to gain control. Player 2 challenges the design that Player 1 chooses, thus he chooses a word (over Σ in the closed setting and over $\Sigma_I \times \Sigma_O$ in the open setting) and wins if his word is a witness for the incorrectness of Player 1's design.

Closed designs The input to the closed-design problem is a closed-library $\mathcal{L}$ and a DFA $\mathcal{S}$ over the alphabet Σ. The goal is to find a correct design $\mathcal{D}$. Recall that $\mathcal{D}$ is correct if the DFA $\mathcal{A}_{\mathcal{L},\mathcal{D}}$ that is constructed from $\mathcal{L}$ using $\mathcal{D}$ satisfies $L(\mathcal{A}_{\mathcal{L},\mathcal{D}}) = L(\mathcal{S})$. We assume that $\mathcal{S}$ is the minimal DFA for the language $L(\mathcal{S})$.

Theorem 1. *The closed-design problem can be solved in polynomial time.*

Proof: Given a closed-library $\mathcal{L}$ and a DFA $\mathcal{S} = \langle \Sigma, S, \delta_S, s^0, F_S \rangle$, we describe a safety game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ such that Player 1 wins $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ iff there is a design of $\mathcal{S}$ using components from $\mathcal{L}$. Recall that $\mathcal{L}$ consists of box-DFAs $\mathcal{B}_i = \langle \Sigma, C_i, \delta_i, c_i^0, F_i, E_i \rangle$, for $i \in [n]$, and that we use $\mathcal{C}$, $\mathcal{C}_0$, $\mathcal{E}$, and $\mathcal{F}$ to denote the union of all states, initial states, exit states, and accepting states in all the components of $\mathcal{L}$. The number of vertices in $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ is $|(\mathcal{C}_0 \cup \mathcal{E}) \times S|$ and it can be constructed in polynomial time. Since solving safety games can be done in linear time, the theorem follows.

We define $\mathcal{G}_{\mathcal{L},\mathcal{S}} = \langle V, E, V_0, \alpha \rangle$. First, $V = (\mathcal{C}_0 \cup \mathcal{E}) \times S$ and $V_0 = \mathcal{C}_0 \times \{s^0\}$. Recall that Player 1 moves when it is time to decide the next (or first) component to gain control. Accordingly, $V_1 = \mathcal{E} \times S$. Also, Player 2 challenges the design suggested

by Player 1 and chooses the word that is processed in a component that gains control, so $V_2 = \mathcal{C}_0 \times S$.

Consider a vertex $\langle e, s\rangle \in V_1$. Player 1 selects the next component to gain control. This component gains control through its initial state. Accordingly, E contains edges $\langle\langle e, s\rangle, \langle c_i^0, s\rangle\rangle$, for every $i \in [n]$. Note that since no letter is read when control is passed, we do not advance the state in $\mathcal{S}$. Consider a vertex $v = \langle c_i^0, s\rangle \in V_2$. Player 2 selects the word that is read in the component $\mathcal{B}_i$, or equivalently, he selects the exit state from which $\mathcal{B}_i$ relinquishes control. Thus, E contains an edge $\langle\langle c_i^0, s\rangle, \langle e, s'\rangle\rangle$ iff there exists a word $u \in \Sigma^*$ such that $\delta_i^*(u) = e$ and $\delta_{\mathcal{S}}^*(s, u) = s'$.

We now turn to define the winning condition. All the vertices in V_1 are in α. A vertex $v \in V_2$ is not in α if it is possible to extend the word traversed for reaching v to a witness for the incorrectness of $\mathcal{D}$. Accordingly, a vertex $\langle c_i^0, s\rangle$ is not in α if one of the following holds. First ("the suffix witness"), there is a finite word that is read inside the current component and witnesses the incorrectness. Formally, there is $u \in \Sigma^*$ such that $\delta_i^*(u) \in F_i$ and $\delta_{\mathcal{S}}^*(s, u) \notin F_{\mathcal{S}}$, or $\delta_i^*(u) \in C_i \setminus (F_i \cup E_i)$ and $\delta_{\mathcal{S}}^*(s, u) \in F_{\mathcal{S}}$. Second ("the infix witness"), there are two words that reach the same exit state of the current component yet the behavior of $\mathcal{S}$ along them is different. Formally, there exist words $u, u' \in \Sigma^*$ such that $\delta_i^*(u) = \delta_i^*(u') \in E_i$ and $\delta_{\mathcal{S}}^*(s, u) \neq \delta_{\mathcal{S}}^*(s, u')$. Intuitively, the minimality of $\mathcal{S}$ enables us to extend either u or u' to an incorrectness witness. Given $\mathcal{L}$ and $\mathcal{S}$, the game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ can be constructed in polynomial time.

In the full version we prove that there is a correct design iff Player 1 wins $\mathcal{G}_{\mathcal{L},\mathcal{S}}$. ■

We continue to study the open setting. Recall that there, the input is a DFA $\mathcal{S}$ over the alphabet $\Sigma_I \times \Sigma_O$ and an open library $\mathcal{L}$. The goal is to find a correct design $\mathcal{D}$ or return that no such design exists, where $\mathcal{D}$ is correct if the composition transducer $\mathcal{T}_{\mathcal{L},\mathcal{D}}$ realizes $L(\mathcal{S})$.

Lustig and Vardi [23] studied the design problem in a setting in which the specification is given by means of an LTL formula. They showed that the problem is 2EXPTIME-complete. Given an LTL formula one can construct a deterministic parity automaton that recognizes the language of words that satisfy the formula. The size of the automaton is doubly-exponential in the size of the formula. Thus, one might guess that the design problem in a setting in which the specification is given by means of a DFA would be solvable in polynomial time. We show that this is not the case and that the problem is EXPTIME-complete. As in [23], our upper bound is based on the ability to "summarize" the activity inside the components. Starting with an LTL formula, the solution in [23] has to combine the complexity involved in the translation of the LTL formula into an automaton with the complexity of finding a design, which is done by going throughout a universal word automaton that is expanded to a tree automaton. Starting with a deterministic automaton, our solution directly uses games: Given an open-library $\mathcal{L}$ and a DFA $\mathcal{S}$, we describe a safety game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ such that Player 1 wins $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ iff there is a design for $\mathcal{S}$ using components from $\mathcal{L}$. The number of vertices in $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ is exponential in S and $\mathcal{C}$. Since solving safety games can be done in linear time, membership in EXPTIME follows. The interesting contribution, however, is the lower bound, showing that the problem is EXPTIME-hard even when the specification is given by means of a deterministic automaton. For that, we describe a reduction from the problem of deciding

whether Player 1 has a winning strategy in a *partial-information safety game*, known to be EXPTIME-complete [8].

Partial-information games are a variant of the *full-information* games defined above in which Player 1 has imperfect information [25]. That is, Player 1 is unaware of the location on which the token is placed and is only aware of the observation it is in. Accordingly, in his turn, Player 1 cannot select the next location to move the token to. Instead, the edges in the game are labeled with actions, denoted Γ. In each round, Player 1 selects an action and Player 2 resolves nondeterminism and chooses the next location the token moves to. In our reduction, the library $\mathcal{L}$ consists of box-transducers $\mathcal{B}_a$, one for every action $a \in \Gamma$. The exit states of the components correspond to the observations in $\mathcal{O}$. That is, when a component exits through an observation $L_i \in \mathcal{O}$, the design decides which component $\mathcal{B}_a \in \mathcal{L}$ gains control, which corresponds to a Player 1 strategy that chooses the action $a \in \Gamma$ from the observation L_i. The full definition of partial-information games as well as the upper and lower bounds are described in the full version.

Theorem 2. *The open-design problem is EXPTIME-complete.*

4 Libraries with Costs

Given a library and a specification, there are possibly many, in fact infinitely many, designs that are solutions to the design problem. As a trivial example, assume that $L(\mathcal{S}) = a^*$ and that the library contains a component $\mathcal{B}$ that traverses the letter a (that is, $\mathcal{B}$ consists of an accepting initial state that has an a-transition to an exist state). An optimal design for $\mathcal{S}$ uses $\mathcal{B}$ once: it has a single state with a self loop in which $\mathcal{B}$ is called. Other designs can use $\mathcal{B}$ arbitrarily many numbers. When we wrote "optimal" above, we assumed that the smaller the design is, the better it is. In this section we would like to formalize the notion of optimality and add to the composition picture different costs that components in the libraries may have.

In order to capture a wide set of scenarios in practice, we associate with each component in $\mathcal{L}$ two costs: a *construction cost* and a *quality cost*. The costs are given by the functions $c\text{-}cost, q\text{-}cost : \mathcal{L} \to \mathbb{R}^+ \cup \{0\}$, respectively. The construction cost of a component is the cost of adding it to the library. Thus, a design that uses a component pays its construction cost once, and (as would be the case in Section 5), when several designs use a component, they share its construction cost. The quality cost measures the performance of the component, and involves, for example, its number of states or security level. Accordingly, a design pays the quality cost of a component every time it uses it, and the fact the component is used by other designs is not important.[4]

Formally, consider a library $\mathcal{L} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ and a design $\mathcal{D} = \langle [n], E, D, d^0, \delta, \nu \rangle$. The number of times $\mathcal{D}$ uses a component B_i is $nused(\mathcal{D}, \mathcal{B}_i) = |\{d \in D : \nu(d) = i\}|$.

[4] One might consider a different quality-cost model, which takes into an account the cost of *computations*. The cost of a design is then the maximal or expected cost of its computations. Such a cost model is appropriate for measures like the running time or other complexity measures. We take here a global approach, which is appropriate for measures like the number of states or security level.

The set of components that are used in $\mathcal{D}$, is $used(\mathcal{D}) = \{\mathcal{B}_i : nused(\mathcal{D}, \mathcal{B}_i) \geq 1\}$. The cost of a design is then $cost(\mathcal{D}) = \sum_{\mathcal{B} \in used(\mathcal{D})} c\text{-}cost(\mathcal{B}) + nused(\mathcal{D}, \mathcal{B}) \cdot q\text{-}cost(\mathcal{B})$.

We state the problem of finding the cheapest design as a decision problem. For a specification DFA $\mathcal{S}$, a library $\mathcal{L}$, and a threshold μ, we say that an input $\langle \mathcal{S}, \mathcal{L}, \mu \rangle$ is in BCD (standing for "bounded cost design") iff there exists a correct design $\mathcal{D}$ such that $cost(\mathcal{D}) \leq \mu$. In this section we study the BCD problem in a setting with a single user. Thus, decisions are independent of other users of the library, which, recall, may influence the construction cost.

In section 3, we reduced the design problem to the problem of the solution of a safety game. In particular, we showed how a winning strategy in the game induces a correct design. Note that while we know that safety games admits memoryless strategies, there is no guarantee that memoryless strategies are guaranteed to lead to optimal designs. We first study this point and show that, surprisingly, while memoryless strategies are sufficient for obtaining an optimal design in the closed setting, this is not the case in the open setting. The source of the difference is the fact that the language of a design in the open setting may be strictly contained in the language of the specification. The approximation may enable the user to generate a design that is more complex and is still cheaper in terms of cost. This is related to the fact that over approximating the language of a DFA may result in exponentially bigger DFAs [5]. We are still able to bound the size of the cheapest design by the size of the game.

4.1 On the optimality and non-optimality of memoryless strategies

Consider a closed library $\mathcal{L}$ and a DFA $\mathcal{S}$. Recall that a correct design for $\mathcal{S}$ from components in $\mathcal{L}$ is induced by a winning strategy of Player 1 in the game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ (see Theorem 1). If the winning strategy is not memoryless, we can trim it to a memoryless one and obtain a design whose state space is a subset of the design induced by the original strategy. Since the design has no flexibility with respect to the language of $\mathcal{S}$, we cannot do better. Hence the following lemma.

Lemma 1. *Consider a closed library $\mathcal{L}$ and a DFA $\mathcal{S}$. For every $\mu \geq 0$, if there is a correct design $\mathcal{D}$ with $cost(\mathcal{D}) \leq \mu$, then there is a correct design $\mathcal{D}'$ induced by a memoryless strategy for Player 1 in $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ such that $cost(\mathcal{D}') \leq \mu$.*

While Lemma 1 seems intuitive, it does not hold in the setting of open systems. There, a design has the freedom to generate a language that is a subset of $L(\mathcal{S})$, as long as it stays receptive. This flexibility allows the design to generate a language that need not be related to the structure of the game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$, which may significantly reduce its cost. Formally, we have the following.

Lemma 2. *There is an open library $\mathcal{L}$ and a family of DFAs $\mathcal{S}_n$ such that $\mathcal{S}_n$ has a correct design $\mathcal{D}_n$ with cost 1 but every correct design for $\mathcal{S}_n$ that is induced by a memoryless strategy for Player 1 in $\mathcal{G}_{\mathcal{L},\mathcal{S}_n}$ has cost n.*

Proof: We define $\mathcal{S}_n = \langle \Sigma_I \times \Sigma_O, S_n, \delta_{\mathcal{S}_n}, s^0_0, F_{\mathcal{S}_n} \rangle$, where $\Sigma_I = \{\tilde{0}, \tilde{1}, \#\}$, $\Sigma_O = \{0, 1, \_\}$, and S_n, $\delta_{\mathcal{S}_n}$ and $F_{\mathcal{S}_n}$ are as follows. Essentially, after reading a prefix of i #'s, for $1 \leq i \leq n$, the design should arrange its outputs so that the i-th and $(n+i)$-th

letters agree (they are $\tilde{0}$ and 0, or are $\tilde{1}$ and 1). One method to do it is to count the number of $\#$'s and then check the corresponding indices. Another method is to keep track of all the first n input letters and make sure that they repeat. The key idea is that while in the second method we strengthen the specification (agreement is checked with respect to all i's, ignoring the length of the $\#$-prefix), it is still receptive, which is exactly the flexibility that the open setting allows. We define $\mathcal{S}_n$ and the library $\mathcal{L}$ so that the structure of $\mathcal{G}_{\mathcal{L},\mathcal{S}_n}$ supports the first method, but counting each $\#$ has a cost of 1. Consequently, a memoryless strategy of Player 1 in $\mathcal{G}_{\mathcal{L},\mathcal{S}_n}$ induces a design that counts, and is therefore of cost n, whereas an optimal design follows the second method, and since it does not count the number of $\#$'s, its cost is only 1.

We can now describe the DFA $\mathcal{S}_n$ in more detail. It consists of n chains, sharing an accepting sink s_{acc} and a rejecting sink s_{rej}. For $0 \leq i \leq n-1$, we describe the i-th chain, which is depicted in Figure 1. When describing $\delta_{\mathcal{S}_n}$, for ease of presentation, we sometimes omit the letter in Σ_I or Σ_O and we mean that every letter in the respective alphabet is allowed. For $0 \leq i < n-1$, we define $\delta_{\mathcal{S}_n}(s_0^i, \#) = s_0^{i+1}$ and $\delta_{\mathcal{S}_n}(s_0^n, \#) = s_0^n$. Note that words of the form $\#^i a_1 \tilde{1} a_2 b 0$ or $\#^i a_1 \tilde{0} a_2 b 1$ are not in $L(\mathcal{S}_n)$, where if $0 \leq i \leq n-1$, then $a_1 \in (\tilde{0}+\tilde{1})^i$, $a_2 \in (\tilde{0}+\tilde{1})^{n-i-1}$, and $b \in (0+1)^i$, and if $i > n-1$ then the lengths of a_1, a_2, and b are $n-1$, 0, and $n-1$, respectively. We require that after reading a word in $\#^*(\tilde{0}+\tilde{1})^n$ there is an output of n letters in $\{0,1\}$. Thus, for $n \leq j \leq n+i+1$, we define $\delta_{\mathcal{S}_n}(s_j^{i,0}, \_) = \delta_{\mathcal{S}_n}(s_j^{i,1}, \_) = s_{rej}$. Also, $\mathcal{S}_n$ accepts every word that has a $\#$ after the initial prefix of $\#$ letters. Thus, for $1 \leq j \leq i$, we define $\delta_{\mathcal{S}_n}(s_j^i, \#) = s_{acc}$, and for $i+1 \leq j \leq n+i+1$ and $t \in \{0,1\}$ we define $\delta_{\mathcal{S}_n}(s_j^{i,t}, \#) = s_{acc}$.

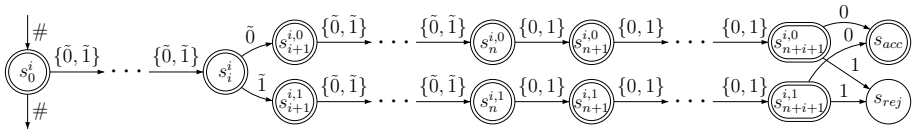


Fig. 1. A description of the i-th chain of the specification $\mathcal{S}_n$.

The library $\mathcal{L}$ is depicted in Figure 2. The quality and construction costs of all the components is 0, except for $\mathcal{B}_1$ which has $q\text{-}cost(\mathcal{B}_1) = 1$ and $c\text{-}cost(\mathcal{B}_1) = 0$.

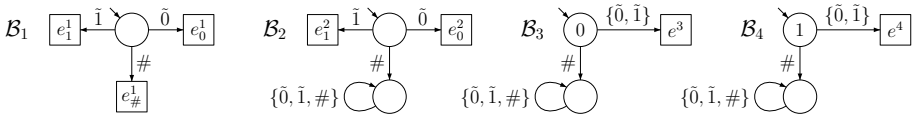


Fig. 2. The library $\mathcal{L}$. Exit states are square nodes and the output of a state is written in the node.

In the full version we prove that every correct design must cost at least 1 and describe such a design, which, as explained above, does not track the number of $\#$'s that are read and can thus use $\mathcal{B}_1$ only once. On the other hand, a design that corresponds to a winning memoryless strategy in $\mathcal{G}_{\mathcal{L},\mathcal{S}}$ uses $\mathcal{B}_1$ n times, thus it costs n, and we are done. ■

4.2 Solving the BCD problem

Theorem 3. *The BCD problem is NP-complete for closed designs.*

Proof: Consider an input $\langle \mathcal{S}, \mathcal{L}, \mu \rangle$ to the BCD problem. By Lemma 1, we can restrict the search for a correct design $\mathcal{D}$ with $cost(\mathcal{D}) \leq \mu$ to these induced by a memoryless strategy for Player 1 in $\mathcal{G}_{\mathcal{L},\mathcal{S}}$. By the definition of the game $\mathcal{G}_{\mathcal{L},\mathcal{S}}$, such a design has at most $|\mathcal{C}_0 \times S|$ states. Since checking if a design is correct and calculating its cost can be done in polynomial time, membership in NP follows. For the lower bound we show a reduction from SET-COVER, which we describe in the full version. ■

We turn to study the open setting, which is significantly harder than the closed one. For the upper bound, we first show that while we cannot restrict attention to designs induced by memoryless strategies, we can still bound the size of optimal designs:

Theorem 4. *For an open library $\mathcal{L}$ with ℓ components and a specification $\mathcal{S}$ with n states, a cheapest correct design $\mathcal{D}$ has at most $\binom{n}{n/2} \cdot \ell$ states.*

Proof: Given $\mathcal{S}$ and $\mathcal{L}$, assume towards contradiction that the cheapest smallest design $\mathcal{D}$ for $\mathcal{S}$ using the components in $\mathcal{L}$ has more than $\binom{n}{n/2} \cdot \ell$ states.

Consider a word $w \in L(\mathcal{T}_{\mathcal{L},\mathcal{D}})$. Let $\mathcal{B}_{i_1}, \ldots, \mathcal{B}_{i_m} \in \mathcal{L}$ be the components that are traversed in the run r of $\mathcal{T}_{\mathcal{L},\mathcal{D}}$ that induces w. Let $w = w_1 \cdot \ldots \cdot w_m$, where, for $1 \leq j \leq m$, the word w_j is induced in the component $\mathcal{B}_{i_j}$. We say that w is suffix-less if $w_m = \epsilon$, thus r ends in the initial state of the last component to gain control. We denote by $\pi_w(\mathcal{D}) = e_{i_1}, \ldots, e_{i_{m-1}} \in \mathcal{E}^*$ the sequence of exit states that r visits.

For a state $d \in D$, we define the set $S_d \subseteq S$ so that $s \in S_d$ iff there exists a suffix-less word $w \in (\Sigma_I \times \Sigma_O)^*$ such that $\delta^*_{\mathcal{S}}(w) = s$ and $\delta^*_{\mathcal{D}}(\pi_w(\mathcal{D})) = d$. Since $\mathcal{D}$ has more than $\binom{n}{n/2} \cdot \ell$ states, there is a component $\mathcal{B}_i \in \mathcal{L}$ such that the set $D' \subseteq \mathcal{L}$ of states that are labeled with $\mathcal{B}_i$ is larger than $\binom{n}{n/2}$. Thus, there must be two states $d, d' \in D'$ that have $S_{d'} \subseteq S_d$. Note that $\nu(d) = \nu(d') = i$.

In the full version we show that we can construct a new correct design $\mathcal{D}'$ by merging d' into d. Since for every component $\mathcal{B} \in \mathcal{L}$, we have $nused(\mathcal{D}, \mathcal{B}) \geq nused(\mathcal{D}', \mathcal{B})$, it follows that $cost(\mathcal{D}) \geq cost(\mathcal{D}')$. Moreover, $\mathcal{D}'$ has less states than $\mathcal{D}$, and we have reached a contradiction. ■

Before we turn to the lower bound, we argue that the exponential blow-up proven in Theorem 4 cannot be avoided:

Theorem 5. *For every $n \geq 1$, there is an open library $\mathcal{L}$ and specification $\mathcal{S}_n$ such that the size of $\mathcal{L}$ is constant, the size of $\mathcal{S}_n$ is $O(n^2)$, and every cheapest correct design for $\mathcal{S}_n$ that uses components from $\mathcal{L}$ has at least 2^n states.*

Proof: Consider the specification $\mathcal{S}_n$ and library $\mathcal{L}$ that are described in Lemma 2. As detailed in the full version, every correct design that costs 1 cannot count #'s and should thus remember vectors in $\{\tilde{0}, \tilde{1}\}^n$. ■

Theorem 6. *The BCD problem for open libraries is NEXPTIME-complete.*

Proof: Membership in NEXPTIME follows from Theorem 4 and the fact we can check the correctness of a design and calculate its cost in polynomial time. For the lower

bound, in the full version, we describe a reduction from the problem of exponential tiling to the BCD problem for open libraries. The idea behind the reduction is as follows. Consider an input $\langle T, V, H, n\rangle$ to EXP-TILING, where $T = \{t_1, \ldots, t_m\}$ is a set of tiles, $V, H \subseteq T \times T$ are *vertical* and *horizontal* relations, respectively, and $n \in \mathbb{N}$ is an index given in unary. We say that $\langle T, V, H, n\rangle \in$ EXP-TILING if it is possible to fill a $2^n \times 2^n$ square with the tiles in T that respects the two relations.

Given an input $\langle T, V, H, n\rangle$, we construct an input $\langle \mathcal{L}, \mathcal{S}, k\rangle$ to the open-BCD problem such that there is an exponential tiling iff there is a correct design $\mathcal{D}$ with $cost(\mathcal{D}) \leq 2^{2n+1} + 1$. The idea behind the reduction is similar to that of Lemma 2. We define $\Sigma_I = \{\tilde{0}, \tilde{1}, \#, c, v, h, \_\}$ and $\Sigma_O = \{0, 1, \_\} \cup T$. For $x \in \{0,1\}^n$, we use $\tilde{x}$ to refer to the $\{\tilde{0}, \tilde{1}\}$ copy of x. The library $\mathcal{L}$ has the same components as in Lemma 2 with an additional *tile component* $\mathcal{B}_t$ for every $t \in T$. The component $\mathcal{B}_t$ outputs t in its initial state, and when reading c, v, or h, it relinquishes control. When reading every other letter, it enters an accepting sink. The construction costs of the components in $\mathcal{L}$ is 0. We define $q\text{-}cost(\mathcal{B}_1) = 2^{2n} + 1$, and $q\text{-}cost(\mathcal{B}_t) = 1$ for all $t \in T$. The other components' quality cost is 0.

Consider a correct design $\mathcal{D}$ with $cost(\mathcal{D}) \leq 2^{2n+1} + 1$. We define $\mathcal{S}$ so that a correct design must use $\mathcal{B}_1$ at least once, thus $\mathcal{D}$ uses it exactly once. Intuitively, $a \cdot b$, for $a, b \in \{0,1\}^n$, can be thought of as two coordinates in a $2^n \times 2^n$ square. We define $\mathcal{S}$ so that after reading the word $\tilde{a} \cdot \tilde{b} \in \{\tilde{0}, \tilde{1}\}^{2n}$, a component is output, which can be thought of as the tile in the (a, b) coordinate in the square. The next letter that can be read is either c, v, or h. Then, $\mathcal{S}$ enforces that the output is $a \cdot b$, $(a+1) \cdot b$, and $a \cdot (b+1)$, respectively. Thus, we show that $\mathcal{D}$ uses exactly 2^{2n} tile components and the tiling that it induces is legal. ■

5 Libraries with Costs and Multiple Users

In this section we study the setting in which several designers, each with his own specification, use the library. The construction cost of a component is now shared by the designers that use it, with the share being proportional to the number of times the component is used. For example, if $c\text{-}cost(\mathcal{B}) = 8$ and there are two designers, one that uses $\mathcal{B}$ once and a second that uses $\mathcal{B}$ three times, then the construction costs of $\mathcal{B}$ of the two designers are 2 and 6, respectively. The quality cost of a component is not shared. Thus, the cost a designer pays for a design depends on the choices of the other users and he has an incentive to share the construction costs of components with other designers. We model this setting as a multi-player game, which we dub *component library games* (CLGs, for short). The game can be thought of as a one-round game in which each player (user) selects a design that is correct according to his specification. In this section we focus on closed designs.

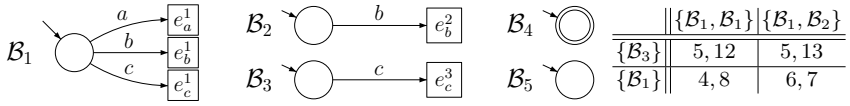
Formally, a CLG is a tuple $\langle \mathcal{L}, \mathcal{S}_1, \ldots, \mathcal{S}_k\rangle$, where $\mathcal{L}$ is a closed component library and, for $1 \leq i \leq k$, the DFA $\mathcal{S}_i$ is a specification for Player i. A strategy of Player i is a design that is correct with respect to $\mathcal{S}_i$. We refer to a choice of designs for all the players as a *strategy profile*. Consider a profile $P = \langle \mathcal{D}_1, \ldots, \mathcal{D}_k\rangle$ and a component $\mathcal{B} \in \mathcal{L}$. The construction cost of $\mathcal{B}$ is split proportionally between the players that use it. Formally, for $1 \leq i \leq k$, recall that we use $nused(\mathcal{B}, \mathcal{D}_i)$ to denote the number of times $\mathcal{D}_i$ uses $\mathcal{B}$. For a profile P, let $nused(\mathcal{B}, P)$ denote the number of times $\mathcal{B}$ is used by all the designs

in P. Thus, $nused(\mathcal{B}, P) = \sum_{1 \leq i \leq k} nused(\mathcal{B}, \mathcal{D}_i)$. Then, the construction cost that Player i pays in P for $\mathcal{B}$ is $c\text{-}cost_i(P, \mathcal{B}) = c\text{-}cost(\mathcal{B}) \cdot \frac{nused(\mathcal{B}, \mathcal{D}_i)}{nused(\mathcal{B}, P)}$. Since the quality costs of the components is not shared, it is calculated as in Section 4. Thus, the cost Player i pays in profile P, denoted $cost_i(P)$ is $\sum_{\mathcal{B} \in \mathcal{L}} c\text{-}cost_i(P, \mathcal{B}) + nused(\mathcal{B}, \mathcal{D}_i) \cdot q\text{-}cost(\mathcal{D}_i)$. We define the cost of a profile P, denoted $cost(P)$, as $\sum_{i \in [k]} cost_i(P)$.

For a profile P and a correct design $\mathcal{D}$ for Player i, let $P[i \leftarrow \mathcal{D}]$ denote the profile obtained from P by replacing the choice of design of Player i by $\mathcal{D}$. A profile P is a *Nash equilibrium* (NE) if no Player i can benefit by unilaterally deviating from his choice in P to a different design; i.e., for every Player i and every correct design $\mathcal{D}$ with respect to $\mathcal{S}_i$, it holds that $cost_i(P[i \leftarrow \mathcal{D}]) \geq cost_i(P)$.

Theorem 7. *There is a CLG with no NE.*

Proof: We adapt the example for multiset cost-sharing games from [6] to CLGs. Consider the two-player CLG over the alphabet $\Sigma = \{a, b, c\}$ in which Player 1 and 2's specifications are (the single word) languages $\{ab\}$ and $\{c\}$, respectively. The library is depicted in Figure 3, where the quality costs of all components is 0, $c\text{-}cost(\mathcal{B}_1) = 12$, $c\text{-}cost(\mathcal{B}_2) = 5$, $c\text{-}cost(\mathcal{B}_3) = 1$, and $c\text{-}cost(\mathcal{B}_4) = c\text{-}cost(\mathcal{B}_5) = 0$. Both players have two correct designs. For Player 1, the first design uses $\mathcal{B}_1$ twice and the second design uses $\mathcal{B}_1$ once and $\mathcal{B}_2$ once. There are also uses of $\mathcal{B}_4$ and $\mathcal{B}_5$, but since they can be used for free, we do not include them in the calculations. For Player 2, the first design uses $\mathcal{B}_2$ once, and the second design uses $\mathcal{B}_1$ once. The table in Figure 3 shows the players' costs in the four possible CLG's profiles, and indeed none of the profiles is a NE. ■



	$\{\mathcal{B}_1, \mathcal{B}_1\}$	$\{\mathcal{B}_1, \mathcal{B}_2\}$
$\{\mathcal{B}_3\}$	$5, 12$	$5, 13$
$\{\mathcal{B}_1\}$	$4, 8$	$6, 7$

Fig. 3. The library of the CLG with no NE, and the costs of the players in its profiles.

We study computational problems for CLGs. The most basic problem is the *best-response problem* (BR problem, for short). Given a profile P and $i \in [k]$, find the cheapest correct design for Player i with respect to the other players' choices in P. Apart from its practical importance, it is an important ingredient in the solutions to the other problems we study. The next problem we study is finding the *social optimum* (SO, for short), namely the profile that minimizes the total cost of all players; thus the one obtained when the players obey some centralized authority. For both the BR and SO problems, we study the decision (rather than search) variants, where the input includes a threshold μ. Finally, since CLGs are not guaranteed to have a NE, we study the problem of deciding whether a given CLG has a NE. We term this problem $\exists$NE.

Note that the BCD problem studied in Section 4 is a special case of BRP when there is only one player. Also, in a setting with a single player, the SO and BR problems coincide, thus the lower bound of Theorem 3 applies to them. In Lemma 1 we showed that if there is a correct design $\mathcal{D}$ with $cost(\mathcal{D}) \leq \mu$, then there is also a correct design $\mathcal{D}'$, based on a memoryless strategy and hence having polynomially many states, such

that for every component $\mathcal{B}$, we have $nused(\mathcal{D}', \mathcal{B}) \leq nused(\mathcal{D}, \mathcal{B})$. The arguments there apply in the more general case of CLGs. Thus, we have the following.

Theorem 8. *The BR and SO problems are NP-complete.*

We continue to study the ∃NE problem. We show that ∃NE is complete for Σ_2^P – the second level of the polynomial hierarchy. Namely, decision problems solvable in polynomial time by a nondeterministic Turing machine augmented by an *oracle* for an NP-complete problem.

Theorem 9. *The ∃NE problem is Σ_2^P-complete.*

Proof: The full proof can be found in the full version. We describe its idea in the following. For the upper bound, we describe a nondeterministic Turing machine with an oracle to SBR problem – the strict version of the BR problem, where we seek a design whose cost is strictly smaller than μ. Given a CLG $\mathcal{G} = \langle \mathcal{L}, \mathcal{S}_1, \ldots, \mathcal{S}_k \rangle$, we guess a profile $P = \langle \mathcal{D}_1, \ldots, \mathcal{D}_k \rangle$, where for $1 \leq i \leq k$, the design $\mathcal{D}_i$ has at most $|\mathcal{C}_0 \times S_i|$ states, where S_i are the states of $\mathcal{S}_i$. We check whether the designs are correct, and use the oracle to check whether there is a player that can benefit from deviating from P. For the lower bound, we show a reduction from the complement of the Π_2^P-complete problem *min-max vertex cover* [20]. ■

6 Discussion

Traditional synthesis algorithms assumed that the system is constructed from scratch. Previous work adjusted synthesis algorithms to a reality in which systems are constructed from component libraries. We adjust the algorithms further, formalize the notions of quality and cost and seek systems of high quality and low cost. We argue that one should distinguish between quality considerations, which are independent of uses of the library by other designs, and pricing considerations, which depend on uses of the library by other designs.

Once we add multiple library users to the story, synthesis is modeled by a resource-allocation game and involves ideas and techniques form algorithmic game theory. In particular, different models for sharing the price of components can be taken. Recall that in our model, users share the price of a component, with the share being proportional to the number of uses. In some settings, a *uniform sharing rule* may fit better, which also makes the game more stable. In other settings, a more appropriate sharing rule would be the one used in *congestion games* – the more a component is used, the higher is its price, reflecting, for example, a higher load. Moreover, synthesis of different specifications in different times gives rise to *dynamic allocation* of components, and synthesis of collections of specifications by different users gives rise to *coalitions* in the games. These notions are well studied in algorithmic game theory and enable an even better modeling of the rich settings in which traditional synthesis is applied.

References

1. M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 25th ICALP*, LNCS 372, pages 1–17. Springer, 1989.

2. S. Almagor, U. Boker, and O. Kupferman. Formalizing and reasoning about quality. In *Proc. 40th ICALP*, LNCS 7966, pages 15 – 27. Springer, 2013.
3. G. Alonso, F. Casati, H.A. Kuno, and V. Machiraju. Web Services - Concepts, Architectures and Applications. *Data-Centric Systems and Applications*. Springer, 2004.
4. R. Alur, K. Etessami, and P. Madhusudan. A temporal logic of nested calls and returns. In *Proc. 10th TACAS*, LNCS 2725, pages 67–79. Springer, 2004.
5. G. Avni and O. Kupferman. When does abstraction help? *IPL*, 113:901–905, 2013.
6. G. Avni, O. Kupferman, and T. Tamir. Congestion and cost-sharing games with multisets of resources. *Submitted*, 2014.
7. G. Avni, O. Kupferman, and T. Tamir. Network-formation games with regular objectives. In *Proc. 17th FoSSaCS*, LNCS 8412, pages 119–133. Springer, 2014.
8. D. Berwanger and L. Doyen. On the power of imperfect information. In *Proc. 28th TST& TCS*, pages 73–82, 2008.
9. A. Bohy, V. Bruyère, E. Filiot, and J-F. Raskin. Synthesis from LTL specifications with mean-payoff objectives. In *Proc. 19th TACAS*, LNCS 7795, pages 169–184. Springer, 2013.
10. U. Boker, K. Chatterjee, T.A. Henzinger, and O. Kupferman. Temporal specifications with accumulative values. In *Proc. 26th LICS*, pages 43–52, 2011.
11. L. de Alfaro, M. Faella, T.A. Henzinger, R. Majumdar, and M. Stoelinga. Model checking discounted temporal properties. *TCS*, 345(1):139–170, 2005.
12. L. de Alfaro and T.A. Henzinger. Interface theories for component-based design. In *Proc. 1st EMSOFT*, LNCS 2211, pages 148–165. Springer, 2001.
13. L. Doyen, T. A. Henzinger, B. Jobstmann, and T. Petrov. Interface theories with component reuse. In *Proc. 8th EMSOFT*, pages 79–88, 2008.
14. J. Elgaard, N. Klarlund, and A. Möller. Mona 1.x: new techniques for WS1S and WS2S. In *Proc. 10th CAV*, LNCS 1427, pages 516–520. Springer, 1998.
15. A. Fabrikant, C. Papadimitriou, and K. Talwar. The complexity of pure nash equilibria. In *Proc. 36th STOC*, pages 604–612, 2004.
16. M. Faella, A. Legay, and M. Stoelinga. Model checking quantitative linear time logic. *ENTCS*, 220(3):61–77, 2008.
17. E. Filiot, N. Jin, and J.-F. Raskin. Antichains and compositional algorithms for LTL synthesis. *FMSD*, 39(3):261–296, 2011.
18. G. Gößler and J. Sifakis. Composition for component-based modeling. *Sci. Comput. Program.*, 55(1-3):161–183, 2005.
19. M. Jurdzinski. Small progress measures for solving parity games. In *Proc. 17th STACS*, LNCS 1770, pages 290–301. Springer, 2000.
20. K-I. Ko and C-L. Lin. On the complexity of min-max optimization problems and their approximation. In *Minimax and Applications*, volume 4 of *Nonconvex Optimization and Its Applications*, pages 219–239. Springer, 1995.
21. O. Kupferman, N. Piterman, and M.Y. Vardi. Safraless compositional synthesis. In *Proc. 18th CAV*, LNCS 4144, pages 31–44. Springer, 2006.
22. O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *Proc. 46th FOCS*, pages 531–540, 2005.
23. Y. Lustig and M.Y. Vardi. Synthesis from component libraries. *STTT* 15:603–618, 2013.
24. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th POPL*, pages 179–190, 1989.
25. J.-F. Raskin, K. Chatterjee, L. Doyen, and T. Henzinger. Algorithms for ω-regular games with imperfect information. *LMCS*, 3(3), 2007.
26. T. Roughgarden and E. Tardos. How bad is selfish routing? *JACM*, 49(2):236–259, 2002.
27. S. Safra. On the complexity of ω-automata. In *Proc. 29th FOCS*, pages 319–327, 1988.
28. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *JACM*, 32:733–749, 1985.