INVITED PAPER

Component-Based Design by Solving Language Equations

The authors review the existing formalisms, algorithmic solutions, and design automation tools to specify and synthesize unknown components in compositional finite-state systems.

By Tiziano Villa, Alexandre Petrenko, Nina Yevtushenko, Alan Mishchenko, and Robert Brayton, *Fellow, IEEE*

ABSTRACT | An important step in the design of a complex system is its decomposition into a number of interacting components, of which some are given (known) and some need to be synthesized (unknown). Then a basic task in the design flow is to synthesize an unknown component that when combined with the known part of the system (the context) satisfies a given specification. This problem arises in several applications ranging from sequential synthesis to the design of discrete controllers. There are different formulations of the problem, depending on the formal models to specify the system and its components, the composition operators, and the conformance relations of the composed system versus the specification. Various behavioral models have been studied in the literature, e.g., finite state machines and automata, omega-automata, process algebras; various forms of synchronous and asynchronous (interleaving/parallel) composition have been considered; the conformance relations include language containment and equality, and notions of simulation. In this paper we give an overview of the problem (a.k.a., the unkown component problem, or submodule construction, etc.), and we focus on its reduction to solving equations over languages, as a key technology for supporting synthesis of compositional systems. We survey the state-of-art and highlight open problems requiring further investigation.

KEYWORDS | Component-based design; decomposition of finite automata and state machines; finite automata and state machines; parallel and synchronous language equations; synthesis of unknown component

I. INTRODUCTION

Component-based system development is perceived as a key technology for designing hardware and software systems in a cost- and time-effective manner, e.g., when a system cannot be built completely from the available modules and some "glue" component needs to be added to satisfy the overall behavioral specification of the system (called a service specification in the context of distributed reactive systems [1]), or a component needs to be replaced for some reason. This paradigm has been endorsed by many industries.

Software industry uses a component-based software development approach to develop software systems by selecting appropriate off-the-shelf components and then assembling them with a well-defined architecture. The concept of building software from prebuilt components arose by analogy with the way that hardware is now designed and built, using "off-the-shelf" modules. The concept of a software component was introduced at the first software engineering conference in 1968 (in the keynote speech "Mass-Produced Software Components" by Doug McIlroy, as reported in [2]). The fact that components hold such an esteemed place in software engineering history should come as no surprise: componentization is a fundamental engineering principle. Top-down approaches decompose large systems into smaller parts, components, and bottom-up approaches compose smaller parts, components, into larger systems. Since 1968, components have played a role in both software engineering research and practice [3].

Manuscript received January 15, 2015; revised May 5, 2015; accepted June 14, 2015. Date of publication August 14, 2015; date of current version October 26, 2015. This work was supported in part by the NSERC under Grant RGPIN/194381-2012, by the Project #739 (GOSZDANIE Russian Federation), and by the EU Commission through the EU Project FP7-ICT-223844 CON4COORD.

T. Villa is with the Department of Computer Science, University of Verona, Verona 37100, Italy (e-mail: tiziano.villa@univr.it).
A. Petrenko is with the Computer Research Institute of Montreal (CRIM), Montreal, QC H3N 7M3 Canada (e-mail: Alexandre.Petrenko@crim.ca).
N. Yevtushenko is with the Department of Radiophysics, Tomsk State University, Tomsk 634050, Russian Federation (e-mail: yevtushenko@sibmail.com).
A. Mishchenko and **R. Brayton** are with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA 94720 USA (e-mail: alanmi@berkeley.edu; brayton@berkeley.edu).

Digital Object Identifier: 10.1109/JPROC.2015.2450937

One often-used definition of software components is "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard" [4]. COTS (commercial off-the-shelf) components, product-line components, and open source components are examples of components in use by the software industry. The component-based software development approach has shown considerable successes in several application domains, among them distributed and web-based systems, desktop and graphical applications [5].

In spite of a big interest from software practitioners who are using *ad hoc* technologies, component-based software development remains mostly a manual process, though in recent years several software development tools have appeared that provide visual programming based on components. Compared to these methodologies, model-driven (and model-based in particular) approaches can raise the automation of the process of software component-based development to a new level. However, the use of formal executable models has received relatively little attention from the research community of software component-based development. Some attempts have been made to formally specify component models, but each defines a particular component model intended for a particular purpose with different semantics, if at all [6]. Formal methods have been applied successfully to the verification of medium-sized programs in protocol and hardware design [7], [8]. However, their application to component-based software development requires more research.

Component-based design plays an increasing role also in hardware synthesis. Consider the synthesis of sequential circuits, which consists of replacing a given circuit representation by an optimized one. The optimized representation may be better in terms of area and delay. Computing such an optimized representation is often a challenge. The challenge is even greater when designing networks of finite state machines (FSMs), where the task is to synthesize/resynthesize the component FSMs in the network, for instance optimizing the single components until the network cannot be simplified further. In order to avoid constructing the FSM that specifies the behavior of the whole network, we may follow a windowing strategy which considers just pairs of connected FSMs to find a replacement for a component of each pair. This calls for a component-based design approach to handle the synthesis of an FSM embedded in a larger FSM environment, with the goal to find the set of all FSMs that can replace the current FSM without changing the external behavior of the whole system; this set represents the complete sequential flexibility of the FSM with respect to its environment. Problems related to the computation and exploitation of the sequential flexibility have been addressed in the past with various techniques in different logic synthesis applications [9], [10].

A general scenario for component-based design of hardware and software systems may be abstracted as follows. Given a global system specified say by an FSM S, called the global FSM, and a library of components represented by FSMs, called context FSMs, implement S as a modular system using instances of the available components; if needed, synthesize also an additional component (not in the library) to complete the system S. For that purpose, we define an iterative greedy procedure based on repeating an operation of "decomposition/division" as follows: given the composition of already instantiated components, compute the quotient of the current specification with respect to all the candidate context components of the library, and select the best one according to a suitable cost function (e.g., the minimum number of states). Then include the selected component into the modular system being built, and update the new specification to be matched. The iteration terminates either when the selected context FSM implements the current specification and so S is decomposed entirely into components of available types; or when further use of any of the available context components does not reduce the cost of the current specification, and so in this case any FSM that implements the current specification is an additional component completing the system S.

This is a generic synthesis scenario that may be instantiated into more specialized ones. The key problem is the availability of a "decomposition/division" algorithm that computes the unknown component (quotient) that combined with a given context yields a given specification. This problem is known by different names (e.g., unknown component problem in the monograph [10], submodule construction [11], etc.) and it has been addressed by different research communities with various formal models and algorithms, as it will be briefly surveyed in Section II. In Section III, we will describe its reduction to solving equations over languages. In Section IV, we will describe algorithms to solve effectively the equations over languages corresponding to finite automata (and finite state machines). In Section V we will mention the frontiers of current research, highlight open issues calling for more investigation and draw conclusions.

II. A RETROSPECTIVE LOOK AT THE UNKOWN COMPONENT PROBLEM AND EQUATION SOLVING

The earliest proposals for casting a general scenario for component-based design in a formal setting include the work of Cerny and Marin [12] in the context of combinational logical circuits, and the work of Merlin and Bochmann [11] in the context of parallel systems. The latter reads "The usual approach to the design of parallel systems involves the important step of dividing the overall system into a number of separate submodules which operate in parallel and interact in some well-defined way.

This paper presents a new approach which assists in the elaboration of the submodule specifications during each single step of the refinement process. Given a complete specification of a given module and the specifications of some submodules, the method described below provides the specification of an additional submodule that, together with the other submodules, will provide a system that satisfies the specification of the given module." This work formulates an equation in terms of a process language, represented by a transition system, which is a finite automaton where each state is accepting. The authors use a protocol design problem to illustrate the proposed solution. The proposed approach for finding the largest solution to the equation has been further extended by Bochmann himself [13] and other researchers to obtain solutions with the desired properties, such as safety and liveness and solutions restricted to FSM languages, which need to be input-progressive and to avoid divergence, see [10] for details and references. A number of authors consider the equations under various relations between processes and consider process models, such as modal transition systems and CCS, e.g., see [14] and [15]. Some research is further focused on modal specifications as automata whose transitions are typed with *may* and *must* modalities, as in [16] and [17]. It has to be mentioned that equations can also be formulated using not only classical Moore and Mealy FSMs, but also a more general model of Input/Output Automata [18], called interface automata in [19].

Following Bochmann, several researchers contributed to the equation solving theory in the context of the protocol design problem, in particular, protocol conversion problem, see, e.g., [20]–[26]. The research community addressing the protocol design problem relies on general techniques for solving equations, while considering various properties of interactions between components, specific to distributed systems. For instance, in [26] they discuss the design of converters for VLSI on-chip protocols, by synthesizing an output transducer as an FSM which is a submachine of the product automaton (i.e., all the machines have the same set of actions) consisting of legal states in terms of data dependency.

A similar observation extends to the research community focusing on the supervisor synthesis problem in the context of control theory [27]. In supervisory control a controller (or supervisor) restricts the behavior of a plant by dynamically disabling some of the controllable events after the execution of each event with the goal to match a desired behavior. An extensive literature has been developed since the seminal work by Ramadge and Wonham [28], [29], studying control under complete and partial observation, centralized and distributed control and also control of nonterminating behaviors [30]–[34]. The approach based on language equations developed in this paper generalizes the results obtained in supervisory control to arbitrary compositions of components in equations, not limited to a loop composition of plant and controller. The supervisory control problem in its basic form is posed as a special case of model matching for FSMs in [35], [36]. In this work, we focus on equation solving considering language containment and equality, while the above work uses simulation relations. Simulation relations in general are stronger (more restrictive) than language containment, further constraining the set of possible solutions.

The problem of equation solving is also addressed by the community focusing on the design of asynchronous sequential circuits that are delay-insensitive, see [37]–[39]. Here, the equations are specialized to a designated composition operator modelling asynchronous communication between processes, called parallel composition in [10] and asynchronous composition here (the corresponding type of equations is also called asynchronous equations in [40]).

Equations specialized to synchronous communications have a long history in the synchronous sequential synthesis community, where the goal is the optimization of a sequential circuit by exploiting the flexibility due to its modular structure and its environment (see [41]–[43]). Special cases of computing the partial or full combinational or sequential flexibility in a circuit were addressed since the 70s (see [12] and [44]), resulting in different approaches described under various names: computation of sequential input and output don't care sequences as in [45]–[47], hierarchical optimization in [48], testing strategies and redundancy identification and removal for interacting FSMs as in [49] and [50], computation of flexibility with the E-machine in [51]. For a thorough presentation of the previous work we refer to [9] and [10].

A final note about the classification of these problems in the hierarchy of computational complexity classes: the authors of [52] proved that assuming modular systems and specifications modeled as deterministic finite automata interacting with parallel composition, many problems related to synthesis and verification of supervisory controllers are PSPACE-complete, which implies that synthesis and verification problems for more general models and topologies are not easier.

We conclude this quick retrospective look by mentioning how a brainstorming event organized by Robert Brayton at Cadence Berkeley Labs in February 1998 triggered a further development of the theory of unknown components based on equation solving. The authors of this paper, representing in fact different research communities, were challenged by the diversity of formal approaches to the common unknown component problem. They recognized that there was a need for a common formal framework for treating various aspects of the problem occurring in component-based design of hardware and software systems, based on language equation approaches. It was decided then to consolidate the efforts in developing such a framework. The collaboration led to a series of contributions [53]–[59], including the research

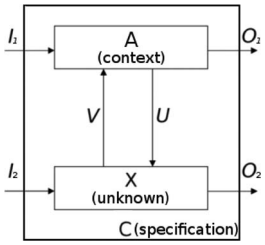

Fig. 1. ***General topology.***

monograph [10] and this paper. Some of these contributions are reviewed in the subsequent sections.

III. FINDING THE UNKNOWN COMPONENTS BY SOLVING INEQUATIONS AND EQUATIONS OVER LANGUAGES

As anticipated in the introduction, in this section we define equations and inequations over languages, first with respect to abstract composition operators and then with respect to concrete ones; in particular, we describe general closed-form solutions first and then restricted solutions.

As reference topology, consider the scheme in Fig. 1, where the composition of two interconnected components (the context or plant A, and the unknown component X) defines a language or behaviour contained in or equal to the one defined by the specification C. The context and the unknown components interact through internal signals, and exchange information with the environment by means of external input and output channels. Restricted topologies simplify this general scheme, by removing some interconnections.

An alphabet is a finite set of symbols. The set of all finite strings over a fixed alphabet X is denoted by $X^\star$. $X^\star$ includes the empty string ϵ. A subset $L \subseteq X^\star$ is called a **language** over alphabet X.

If L, L_1, L_2 are languages and X is an alphabet, some standard operations on languages are: union $L_1 \cup L_2$, concatenation $L_1 L_2$, Kleene closure $L^\star = \cup_{i=0}^{\infty} L^i$ (and positive Kleene closure $L^+ = \cup_{i=1}^{\infty} L^i$), intersection $L_1 \cap L_2$, complement $\overline{L} = X^\star \setminus L$, and difference $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$.

Furthermore, we recall the notions of substitution and homomorphism of languages [60]. A **substitution** f is a mapping of an alphabet Σ onto subsets of $\Delta^\star$ for some alphabet Δ. The substitution f is extended to strings by setting $f(\epsilon) = \{\epsilon\}$ and $f(xa) = f(x)f(a)$, and then to languages by setting $f(L) = \{f(\alpha) \mid \alpha \in L\}$. In the sequel, instead of the functional notation $f(L)$, we will use the suffix notation L_f, as in $L_\top$ and $L_\bot$, or even in $L_{\top\Delta}$ and $L_{\bot\Delta}$ where we introduce explicitly the alphabet Δ of the image language under the substitutions $f = \top$ or $f = \bot$. An **homomorphism** h is a substitution such that $h(a)$ is a single string for each symbol a in the alphabet Σ.

Given the disjoint alphabets I, U, O, a language L_1 over alphabet $I \circ U$ and a language L_2 over alphabet $U \circ O$, consider the following symbols to represent operators defined over alphabets and languages:

- $\circ$ denotes an operator between alphabets;
- $\odot$ denotes a **composition operator** between languages, defined as

$$L_1 \odot_{I\circ O} L_2 = \left[(L_1)_{\top O} \cap (L_2)_{\top I}\right]_{\bot I\circ O}$$

where $\top$ and $\bot$ denote language substitutions operators in suffix notation, as introduced above.

A. Language Equations Under Abstract Composition

We define language inequations and equations, whose solutions we want to characterize.

Definition 3.1: Given the disjoint alphabets I, U, O, a language A over alphabet $I \circ U$ and a language C over alphabet $I \circ O$, we define the **language inequation**

$$A \odot_{I\circ O} X \subseteq C \tag{1}$$

and the **language equation**

$$A \odot_{I\circ O} X = C \tag{2}$$

with respect to the unknown language X over alphabet $U \circ O$.

Notice that for simplicity, when referring to both "language inequations" and "language equations" in this exposition we may lump them together under the term "language equations." To make explicit the alphabets from which each language and operator depends, the previous language inequation should be written as

$$A_{I\circ U} \odot_{I\circ O} X_{U\circ O} \subseteq C_{I\circ O} \tag{3}$$

and the language equation as

$$A_{I\circ U} \odot_{I\circ O} X_{U\circ O} = C_{I\circ O}. \tag{4}$$

In the sequel, for ease of parsing the notation, we will annotate explicitly only the language subscripts deemed essential to avoid ambiguity. In this section we will describe a closed-form solution of such language inequations with respect to an abstract composition operator $\odot$ under the least restrictive known algebraic conditions.

Definition 3.2: Given the disjoint alphabets I, U, O, a language A over alphabet $I \circ U$ and a language C over alphabet $I \circ O$, language B over alphabet $U \circ O$ is called a **solution** of the inequation $A \odot_{I\circ O} X \subseteq C$ iff $A \odot_{I\circ O} B \subseteq C$. A solution is called the **largest solution** if it contains any other solution. $B = \emptyset$ is the trivial solution.

The following theorem states requirements on the substitution operators $\top$ and $\bot$ that allow writing the largest solution in the closed-form $S = \overline{A \odot_{U\circ O} \overline{C}}$.

Theorem 3.1: If the substitution operators $\top$ and $\bot$ are such that:

*H*1: given disjoint alphabets Z, Y and language L over Z, $(L_{\top Y})_{\bot Z} = L$;

*H*2: given disjoint alphabets Z, Y and languages L_1, L_2 over $Y \circ Z$, if $L_1 = (L_{1\bot Z})_{\top Y}$ or $L_2 = (L_{2\bot Z})_{\top Y}$ then $(L_1 \cap L_2)_{\bot Z} = L_{1\,\bot Z} \cap L_{2\,\bot Z}$;

*H*3: given disjoint alphabets Z, Y and language L over $Y \circ Z$, $L_{\bot Z} = \emptyset \Leftrightarrow L = \emptyset$;

then there exists the largest solution of the inequation $A \odot X \subseteq C$ and this solution is the language

$$S = \overline{A \odot \overline{C}}. \tag{5}$$

Any language contained in S is a solution of the language inequation. If the language $S = \overline{A \odot \overline{C}}$ is empty, then the inequation has a single trivial solution, namely the empty language. ⋈

Proof: Consider a string $\alpha \in (U \circ O)^\star$, then α is in the largest solution of $A \odot X \subseteq C$ iff $A \odot \{\alpha\} \subseteq C$ and the following chain of equivalences follows:

$$\begin{aligned}
A \odot \{\alpha\} \subseteq C \quad &\Leftrightarrow \\
(A_{\top O} \cap \{\alpha\}_{\top I})_{\bot I\circ O} \cap \overline{C} = \emptyset \quad &\Leftrightarrow \\
\textit{by Hyp. } H1: \ \overline{C} = (\overline{C}_{\top U})_{\bot I\circ O} & \\
(A_{\top O} \cap \{\alpha\}_{\top I})_{\bot I\circ O} \cap (\overline{C}_{\top U})_{\bot I\circ O} = \emptyset \quad &\Leftrightarrow \\
\textit{by Hyp. } H2: \ \cap \textit{ and } \bot \textit{ commute} & \\
(A_{\top O} \cap \{\alpha\}_{\top I} \cap \overline{C}_{\top U})_{\bot I\circ O} = \emptyset \quad &\Leftrightarrow \\
\textit{by Hyp. } H3: \ Y = U, \ X = I \circ O & \\
A_{\top O} \cap \{\alpha\}_{\top I} \cap \overline{C}_{\top U} = \emptyset \quad &\Leftrightarrow \\
\textit{by Hyp. } H3: \ Y = I, X = U \circ O & \\
(A_{\top O} \cap \{\alpha\}_{\top I} \cap \overline{C}_{\top U})_{\bot U\circ O} = \emptyset \quad &\Leftrightarrow \\
\textit{by Hyp. } H2: \ \cap \textit{ and } \bot \textit{ commute} & \\
(\{\alpha\}_{\top I})_{\bot U\circ O} \cap (A_{\top O} \cap \overline{C}_{\top U})_{\bot U\circ O} = \emptyset \quad &\Leftrightarrow \\
\textit{by Hyp. } H1: \ (\{\alpha\}_{\top I})_{\bot U\circ O} = \{\alpha\} & \\
\{\alpha\} \cap (A_{\top O} \cap \overline{C}_{\top U})_{\bot U\circ O} = \emptyset \quad &\Leftrightarrow \\
\alpha \notin (A_{\top O} \cap \overline{C}_{\top U})_{\bot U\circ O} \quad &\Leftrightarrow \\
\alpha \in \overline{(A_{\top O} \cap \overline{C}_{\top U})_{\bot U\circ O}} \quad &\Leftrightarrow \\
\alpha \in \overline{A \odot \overline{C}}. &
\end{aligned}$$

Therefore the largest solution of the language inequation $A \odot X \subseteq C$ is given by the language $S = \overline{A \odot \overline{C}}$.

Corollary 3.1: If $S \odot \overline{A \odot \overline{C}} = C$, then S is the largest solution of the language equation $A \odot X = C$. A subset of S may not be a solution of the language equation.

If $S \odot \overline{A \odot \overline{C}} \subset C$, then the language equation is unsolvable and the language $D = S \odot \overline{A \odot \overline{C}}$ is the largest subset of C such that the language equation $A \odot X = D$ is solvable. ⋈

B. Language Equations Under Concrete Composition: Synchronous and Asynchronous Operators

Consider two systems A and B with associated languages $L(A)$ and $L(B)$. The systems communicate with each other by a channel U and with the environment by channels I and O. We introduce two concrete composition operators that describe the external behaviour of the composition of $L(A)$ and $L(B)$: synchronous composition (studied, e.g., in [54]) and asynchronous (a.k.a. as interleaving or parallel) composition (studied, e.g., in [40] and [53]).

1) Synchronous Inequations and Equations: To define synchronous composition, consider the following operations on languages.

1) Given a language L over alphabet $X \times V$, consider the homomorphism $p : X \times V \to V^\star$ defined as
$$p((x, v)) = v$$
then the language
$$L_{\downarrow V} = \{p(\alpha) \mid \alpha \in L\}$$
over alphabet V is the **projection** of language L to alphabet V, or V-projection of L. By definition of substitution $p(\epsilon) = \epsilon$.
2) Given a language L over alphabet X and an alphabet V, consider the substitution $l : X \to 2^{(X\times V)^\star}$ defined as
$$l(x) = \{(x, v) \mid v \in V\}$$
then the language
$$L_{\uparrow V} = \{l(\alpha) \mid \alpha \in L\}$$
over alphabet $X \times V$ is the **lifting** of language L to alphabet V, or V-lifting of L. By definition of substitution $l(\epsilon) = \{\epsilon\}$.

By definition $\emptyset_{\downarrow V} = \emptyset$, $\emptyset_{\uparrow V} = \emptyset$. As an example of lifting, consider $X = \{u, v\}$, $L = \{uv\}$, $V = \{x, y\}$, then $L_{\uparrow V} = \{(u, x)(v, x), (u, x)(v, y), (u, y)(v, x), (u, y)(v, y)\}$.

The given substitution operators change a language and its alphabet of definition; in particular the operators $\uparrow$ and $\downarrow$ vary the components that are present in the Cartesian product defining the language alphabet. We assume that each component has a fixed position in the Cartesian product. For instance, let language L_1 be defined over alphabet I and language L_2 be defined over alphabet O, then language $L_{1\uparrow O}$ is defined over alphabet $I \times O$ and also language $L_{2\uparrow I}$ is defined over alphabet $I \times O$, assuming an ordering of alphabets.

Definition 3.3: Given the alphabets I, U, O, language L_1 over $I \times U$ and language L_2 over $U \times O$, the **synchronous composition** of languages L_1 and L_2 is the language[1] $[(L_1)_{\uparrow O} \cap (L_2)_{\uparrow I}]_{\downarrow I \times O}$, denoted by $L_1 \bullet_{I \times O} L_2$, defined over $I \times O$.

Notice that each language may be defined by the Cartesian product of other languages, according to the overall composition topology.

So the synchronous composition operator $\bullet$ is a concrete instantiation of the abstract operator $\odot$, where the alphabet operator $\circ$ becomes $\times$, and the language substitution operators $\top$ and $\bot$ become, respectively, $\uparrow$ and $\downarrow$. One can prove that Hyp. H1, H2, and H3 of Th. 3.1 hold for the substitution operators $\uparrow$ and $\downarrow$ (see [10], respectively, Prop. 2.1(a) p. 11, Prop. 2.3(d) p. 12, and Prop. 2.7(b) p. 14). Therefore, as a corollary of Th. 3.1, we deduce the solution of the synchronous language inequation

$$A \bullet X \subseteq C. \tag{6}$$

Theorem 3.2: Given the alphabets I, U, O, a language A over alphabet $I \times U$ and a language C over alphabet $I \times O$, the largest solution S over alphabet $U \times O$ of the inequation $A \bullet X \subseteq C$ is the language $S = \overline{A \bullet \overline{C}}$. $\bowtie$

Proof: The proof follows from Th. 3.1, because Hyp. H1, H2, and H3 hold for the operators $\uparrow$ and $\downarrow$ (see [10], Prop. 2.1(a) p. 11 for H1, Prop. 2.3(d) p. 13 for H2, and Prop. 2.7(b) p. 14 for H3). Notice that the assumptions under which Hyp. H2 holds are verified (in the first instance because $((\overline{C}_{\uparrow U})_{\downarrow I \times O})_{\uparrow U} = \overline{C}_{\uparrow U}$ and in the second instance because $\{\alpha\}_{\uparrow I} = ((\{\alpha\}_{\uparrow I})_{\downarrow U \times O})_{\uparrow I}$).

For convenience of the reader, we spell out in detail by the following corollary the conclusions from Th. 3.2 and Cor. 3.1.

Corollary 3.2: Given the alphabets I, U, O, a language A over alphabet $I \times U$ and a language C over alphabet $I \times O$, language $S \neq \emptyset$ over alphabet $U \times O$ is a solution of the inequation $A \bullet X \subseteq C$ iff $S \subseteq \overline{A \bullet \overline{C}}$. If the language $\overline{A \bullet \overline{C}}$ is empty, then the only solution of the inequation $A \bullet X \subseteq C$ is the empty language.

If $A \bullet \overline{A \bullet \overline{C}} = C$, then the language $S = \overline{A \bullet \overline{C}}$ is the largest solution over $U \times O$ of the equation $A \bullet X = C$. However, not each subset of S inherits the property of being a solution of the language equation.

If $A \bullet \overline{A \bullet \overline{C}} \subset C$, then the equation $A \bullet X = C$ is unsolvable and the language $D = \overline{A \bullet \overline{A \bullet \overline{C}}}$ is the largest subset of C such that the equation $A \bullet X = D$ is solvable over $U \times O$. $\bowtie$

At the moment no better characterization of all the solutions (all subsets of the largest solution) of a language equation is known.

[1]Use the same order $I \times U \times O$ in the languages $(L_1)_{\uparrow O}$ and $(L_2)_{\uparrow I}$.

2) Asynchronous Inequations and Equations: To define asynchronous composition, consider the following operations

1) Given a language L over alphabet $X \cup V$, where $X \cap V = \emptyset$, consider the homomorphism $r : X \cup V \to V^\star$ defined as

$$r(y) = \begin{cases} y & \text{if } y \in V \\ \epsilon & \text{if } y \in X, \end{cases}$$

then the language

$$L_{\Downarrow V} = \{r(\alpha) \mid \alpha \in L\}$$

over alphabet V is the **restriction** of language L to alphabet V, or V-restriction of L, i.e., words in $L_{\Downarrow V}$ are obtained from those in L by deleting all the symbols in X. By definition of substitution $r(\epsilon) = \epsilon$.

2) Given a language L over alphabet X and an alphabet V disjoint from X, consider the mapping $e : X \to 2^{(X \cup V)^\star}$ defined as

$$e(x) = \{\alpha x \beta \mid \alpha, \beta \in V^\star\}$$

then the language

$$L_{\Uparrow V} = \{e(\alpha) \mid \alpha \in L\}$$

over alphabet $X \cup V$ is the **expansion** of language L to alphabet V, or V-expansion of L, i.e., words in $L_{\Uparrow V}$ are obtained from those in L by inserting anywhere in them words from $V^\star$. Notice that e is not a substitution and that $e(\epsilon) = \{\alpha \mid \alpha \in V^\star\}$.

By definition $\emptyset_{\Downarrow V} = \emptyset$, $\emptyset_{\Uparrow V} = \emptyset$. As an example of expansion, consider $X = \{u, v\}$, $L = \{uv\}$, $V = \{x, y\}$, then $L_{\Uparrow V} = \{\{x,y\}^\star u \{x,y\}^\star v \{x,y\}^\star\}$.

Definition 3.4: Given the pairwise disjoint alphabets I, U, O, language L_1 over $I \cup U$ and language L_2 over $U \cup O$, the **asynchronous composition** of languages L_1 and L_2 is the language $[(L_1)_{\Uparrow O} \cap (L_2)_{\Uparrow I}]_{\Downarrow I \cup O}$, denoted by $L_1 \diamond_{I \cup O} L_2$, defined over $I \cup O$.

So the asynchronous composition operator $\diamond$ is a concrete instantiation of the abstract operator $\odot$, where the alphabet operator $\circ$ becomes $\cup$, and the language substitution operators $\top$ and $\bot$ become, respectively, $\Uparrow$ and $\Downarrow$. One can prove that Hyp. H1, H2, and H3 of Th. 3.1 hold for the substitution operators $\Uparrow$ and $\Downarrow$ (see [10], respectively, Prop. 2.1(c) p. 11, Prop. 2.5(d) p. 13, and Prop. 2.7(d) p. 14). Therefore, as a corollary of Th. 3.1, we deduce the solution of the asynchronous language inequation

$$A \diamond X \subseteq C. \tag{7}$$

Theorem 3.3: Given the pairwise disjoint alphabets I, U, O, a language A over alphabet $I \cup U$ and a language C over alphabet $I \cup O$, the largest solution S over alphabet

$U \cup O$ of the inequation $A \diamond X \subseteq C$ is the language $S = A \diamond \overline{C}$. $\bowtie$

Proof: The proof follows from Th. 3.1, because Hyp. H1, H2 and H3 hold for the operators $\Uparrow$ and $\Downarrow$ (see [10, Sec. 2.1.1, p. 9–14]). Notice that the assumptions under which Hyp. H2 holds are verified (in the first instance because $((\overline{C}_{\Uparrow U})_{\Downarrow I \cup O})_{\Uparrow U} = \overline{C}_{\Uparrow U}$ and in the second instance because $\{\alpha\}_{\Uparrow I} = ((\{\alpha\}_{\Uparrow I})_{\Downarrow U \cup O})_{\Uparrow I}$).

A summary of conclusions symmetric to the ones in Cor. 3.2 can be drawn from Th. 3.3.

In summary, there is a complete characterization of the set of solutions of language inequalities, whereas the complete characterization of the set of solutions of language equations needs additional research. More results and discussions on solutions of language equations can be found in [10] and [58].

C. Restricted Solutions of Language Equations

Given a language inequality or equation, not each solution is of interest, but specific applications may dictate a restriction to subsets of solutions. A straightforward restriction may be to enforce that the composed system be nontrivial. Restricted solutions which have been investigated include prefix-closed, progressive and compositionally progressive solutions. We illustrate such restricted solutions for a synchronous language equation and for simplicity we assume that component languages are defined over the Cartesian product of two alphabets.

Definition 3.5: The language L over alphabet A is **prefix-closed** if each prefix of each word is in the language L. A language L over alphabet $A = I \times O$ is **I-progressive** if $\forall \alpha \in A^*\ \forall i \in I\ \exists o \in O\ [\alpha \in L \rightarrow \alpha(i,o) \in L]$. A language L over alphabet $A = I \times O$ is **I-defined** if $L_{\downarrow I} = I^*$.

If a language over $A = I \times O$ is I-progressive then it is also I-defined, but the converse does not hold. A progressive solution ensures that the solution language is complete w.r.t. the alphabet I, i.e., for each string $\beta \in I^*$ there exists a word in the language with the projection β. Progressive solutions are used in logic synthesis, since physical devices usually are input-enabled at each state, and especially are of interest when solving equations over FSMs.

Definition 3.6: Given a language L_1 over alphabet $I \times U$, a language B over alphabet $O \times U$ is I-**compositionally progressive** (w.r.t. the language L_1) if the language $L_{1\uparrow I \times U \times O} \cap B_{\uparrow I \times U \times O}$ is I-progressive.

When a solution of the language equation is compositionally progressive we are guaranteed that the corresponding composition does not fall into a deadlock when I is the set of external inputs submitted by the environment.

Given language L_1 over alphabet $I \times U$ and language L over alphabet $\theta = I \times O$, let X be an unknown language over alphabet $\rho = U \times O$ and $S_\rho = \overline{L_1 \bullet_\rho \overline{L}}$ be the largest solution of the language equation $L_1 \bullet_\theta X = L$. It is interesting to investigate subsets of S_ρ that satisfy further properties, i.e., are prefix-closed, progressive, etc.

If S_ρ is prefix-closed then S_ρ is the largest prefix-closed solution to the equation. However, not each subset of S_ρ inherits this property. If S_ρ is not prefix-closed then denote by $Pref(S_\rho)$ the set obtained from S_ρ by deleting each string that has a prefix not in S_ρ.

Theorem 3.4: If $L_1 \bullet_\theta Pref(S_\rho) = L$ then $Pref(S_\rho)$ is the largest prefix-closed solution to the equation $L_1 \bullet_\theta X = L$. If $L_1 \bullet_\theta Pref(S_\rho) \subset L$, then the equation $L_1 \bullet_\theta X = L$ has no prefix-closed solution. $\bowtie$

If S_ρ is defined over the alphabet $U \times O$ and S_ρ is U-progressive then S_ρ is the largest U-progressive solution to the equation. However, not each subset of S_ρ inherits this property. If S_ρ is not U-progressive then denote by $Prog(S_\rho)$ the subset obtained from S_ρ by deleting each string β such that for some $u \in U$, there is no $o \in O$ for which $\beta(u,o) \in S_\rho$.

Theorem 3.5: If $L_1 \bullet_\theta Prog(S_\rho) = L$ then $Prog(S_\rho)$ is the largest progressive solution to the equation $L_1 \bullet_\theta X = L$. If $L_1 \bullet_\theta Prog(S_\rho) \subset L$, then the equation $L_1 \bullet_\theta X = L$ has no progressive solution. $\bowtie$

In Section IV-B we will discuss compositionally progressive solutions over FSM languages, where it holds that if a synchronous FSM equation (FSM inequality) has a compositionally progressive solution then the FSM equation (FSM inequality) has the largest compositionally progressive solution.

IV. LANGUAGE EQUATIONS OVER REGULAR LANGUAGES

In the previous section we studied the largest solutions of language inequations and equations. In this section we restrict them to a context and specification that are regular languages. For them we are able to set up effective computations that solve them; this is due to fact that regular languages are generated by finite automata, for which we know how to implement the operators required to find the solutions in closed form or by iterative procedures. We will also discuss the special case of regular languages generated by FSMs (in short, FSM languages), and point out some results about their restricted solutions of practical interest.

A. Solving Effectively Regular Language Equations

To solve equations over regular languages we operate on Finite Automata (FA), which are closed under the operations required to compute the largest solution.

Definition 4.1: A **finite automaton** (FA) is a 5-tuple $F = \langle S, \Sigma, \Delta, r, Q \rangle$. S represents the finite state space, Σ

represents the finite alphabet of actions, and $\Delta \subseteq \Sigma \times S \times S$ is the next state relation, such that $(\sigma, p, n) \in \Delta$ iff $n \in S$ is a next state of present state $p \in S$ on action $i \in \Sigma$. The initial or reset state is $r \in S$ and $Q \subseteq S$ is the set of final or accepting states. The automaton F is deterministic, if for each state $s \in S$ and any action $\sigma \in \Sigma$ there exists at most one state $s\prime$, such that $(\sigma, s, s\prime) \in \Delta$, otherwise it is nondeterministic. A variant of FA allows the introduction of ϵ-moves, meaning that $\Delta \subseteq (\Sigma \cup \{\epsilon\}) \times S \times S$.[2]

Well-known results state that each regular language can be generated or represented by a deterministic finite automaton and that regular languages are closed under union, intersection and complementation. Regular languages are also closed under projection, lifting, restriction and expansion. Below we sketch the constructions for the less known operations of projection, lifting, restriction and expansion.

Projection ($\downarrow$) Given FA F that accepts language L over $I \times U$, FA $F_{\downarrow I}$ that accepts language $L_{\downarrow I}$ over I is obtained by replacing each edge $((i,u), s, s')$ in F by the edge (i, s, s').[3]

Lifting ($\uparrow$) Given FA F that accepts language L over I, FA $F_{\uparrow I \times U}$ that accepts language $L_{\uparrow I \times U}$ over $I \times U$ is obtained by replacing each edge (i, s, s') in F by the set of edges $\{((i,u), s, s') : u \in U\}$.

Restriction ($\Downarrow$) Given FA F that accepts language L over A and a nonempty subset V of A, FA $F_{\Downarrow V}$ that accepts language $L_{\Downarrow V}$ over V is obtained by replacing each edge (a, s, s') in F, with $a \in A \setminus V$, by the edge (ε, s, s').[2]

Expansion ($\Uparrow$) Given alphabet A, a nonempty subset V of A, FA F that accepts language L over V, FA $F_{\Uparrow A}$ that accepts language $L_{\Uparrow A}$ over A is obtained by the following procedure: for each state s of FA F, $\forall a \in A \setminus V$, the edge (self-loop) (a, s, s) is added.

Given that the operators used to express the solution of regular language inequations (see Cor. 3.2 and Cor. 3.3) have constructive counterparts on finite automata, we conclude that there is an effective (constructive) way to solve inequations and equations over regular languages.

As an illustration, given a regular language equation $L_C \bullet_\theta X = L$, where L_C is a regular language over alphabet $I \times U$, L is a regular language over $\theta = I \times O$, and the unknown is a regular language X over $\rho = U \times O$, the following algorithm computes the unknown component X:

1) Derive finite automata F_C and F which accept respectively regular languages L_C and L.
2) If F is a nondeterministic automaton then determinize F by the subset construction and obtain the automaton $\overline{F}$ by interchanging the sets of accepting and nonaccepting states of F.
3) Obtain the automaton $F_{C\uparrow I\times U\times O}$ by replacing each label (i,u) with all triples (i,u,o), $o \in O$.
4) Obtain the automaton $\overline{F}_{\uparrow I\times U\times O}$ by replacing each label (i,o) with all triples (i,u,o), $u \in U$.
5) Build the intersection $F_{C\uparrow I\times U\times O} \cap \overline{F}_{\uparrow I\times U\times O}$. The states of the obtained automaton are pairs of states of $F_{C\uparrow I\times U\times O}$ and $\overline{F}_{\uparrow I\times U\times O}$, the initial state is the pair of initial states, and a state of the intersection is accepting if both states of the pair are accepting.
6) Project $F_{C\uparrow I\times U\times O} \cap \overline{F}_{\uparrow I\times U\times O}$ to $\rho = U \times O$ by deleting i from the labels (i,u,o). The obtained automaton in general is nondeterministic; in this case, determinize it by the subset construction and obtain the automaton $F_C \bullet_\rho \overline{F}$ which accepts the language $L_C \bullet_\rho \overline{L}$. Obtain the automaton S_ρ which accepts the regular language $\overline{L_C \bullet_\rho \overline{L}}$ by interchanging the sets of accepting and nonaccepting states of $F_C \bullet_\rho \overline{F}$.
7) Derive the automaton $\left(F_{C\uparrow I\times U\times O} \cap S_{\rho\uparrow I\times U\times O}\right)_{\downarrow I\times O}$. If the automaton accepts the language L then the regular language $\overline{L_C \bullet_\rho \overline{L}}$ is the largest solution over the alphabet $U \times O$ of the equation $L_C \bullet_\theta X = L$. Otherwise, the regular language equation $L_C \bullet_\theta X = L$ has no solution over the alphabet $U \times O$.

We apply the previous procedure to the example shown in Fig. 2, where accepting states are shown in double lines. Automata F_C and F accept respectively the languages L_C and L over alphabets $I \times U$ and $I \times O$. The automaton $\overline{F}$ in Fig. 2(c) accepts the language $\overline{L}$. Fig. 2(d) shows the automaton $F_{C\uparrow I\times U\times O} \cap \overline{F}_{\uparrow I\times U\times O}$ that accepts the intersection $L_{C\uparrow I\times U\times O} \cap \overline{L}_{\uparrow I\times U\times O}$. Fig. 2(e) presents the deterministic automaton that accepts the projection of $L_{C\uparrow I\times U\times O} \cap \overline{L}_{\uparrow I\times U\times O}$ onto the alphabet $U \times O$, i.e., the synchronous composition $L_C \bullet_{U\times O} \overline{L}$, while Fig. 2(f) shows the automaton that accepts the largest solution $S_\rho = \overline{L_C \bullet_\rho \overline{L}}$ of the equation $L_C \bullet_{I\times O} X = L$ over alphabet $U \times O$.

As an example of asynchronous equation over regular languages, consider $L_C \diamond_E X = L$ where L_C is a regular language over alphabet A_C, L is a regular language over alphabet E, and the unknown language X is over alphabet $R \subseteq A_C \cup E$, and let the corresponding finite automata be represented in Fig. 3 (from [58], [61]). The automaton F_C is defined over the alphabet $A_C = \{e_1, e_2, i\}$ and the automaton F over the alphabet $E = \{e_1, e_2, x\}$. The automaton S_R shown in Fig. 3(c) is defined over $R = A_C \cup E = \{e_1, e_2, i, x\}$ and accepts the largest solution $\overline{L_C \diamond_R \overline{L}}$ of the equation $L_C \diamond_E X = L$. Notice that $\overline{F} = \emptyset$, $\overline{F}_{\Uparrow\{i\}} = \emptyset$, $F_{C\Uparrow\{x\}} \cap \overline{F}_{\Uparrow\{i\}} = \emptyset$, $\left(F_{C\Uparrow\{x\}} \cap \overline{F}_{\Uparrow\{i\}}\right)_{\Downarrow R} = \emptyset$, and so $\overline{\left(F_{C\Uparrow\{x\}} \cap \overline{F}_{\Uparrow\{i\}}\right)_{\Downarrow R}} = S_R$, where the language of S_R is $R^\star$.

There are two issues to investigate further:

1) the existence of solutions that are not regular for a regular (synchronous or asynchronous) language equation;
2) the characterization of all the subsets of the largest solution of a regular language equation.

[2]Apply the closure procedure to obtain an equivalent deterministic FA without ε-moves.

[3]Apply the subset construction to obtain an equivalent deterministic FA.

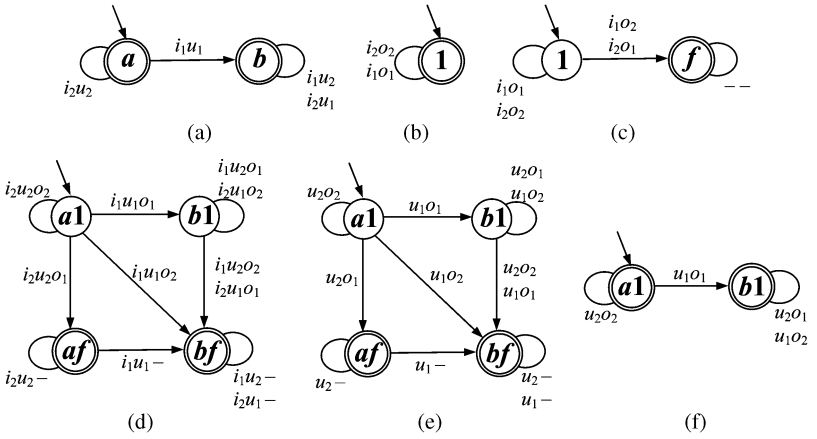

Fig. 2. *Solving a synchronous equation over regular languages: (a) automaton F_C; (b) automaton F; (c) automaton $\overline{F}$; (d) $F_{C\uparrow I\times U\times O} \cap \overline{F}_{\uparrow I\times U\times O}$; (e) deterministic automaton $(F_{C\uparrow I\times U\times O} \cap \overline{F}_{\uparrow I\times U\times O})_{\downarrow U\times O}$ accepting the language $L_C \bullet_{U\times O} \overline{L}$; (f) automaton accepting the largest solution $S_\rho = L_C \bullet_\rho \overline{L}$.*

About the latter question, notice that when solving an inequation we consider sequences and thus any subset of the largest solution is a solution of the inequation; instead, when solving an equation we consider sets of sequences, in place of sequences. The complete characterization of the solutions of a parallel equation over deterministic automata was developed in [61], based on the regular languages associated with the states of the largest solution.

B. Restricted Solutions of Equations Over FSM Languages

FSMs are a formalism widely used in hardware and software design, and they generate a subclass of regular languages. Inequations and equations over FSMs and their associated languages model problems of compositional FSM design. Here we review briefly the basic notions related to FSMs and their languages from [10], and then we mention some results about solutions of FSM languages.

Definition 4.2: A **finite state machine** (FSM) is a 5-tuple $M = \langle S, I, O, T, r\rangle$ where S represents the finite state space, I represents the finite input space, O represents the finite output space and $T \subseteq I \times S \times S \times O$ is the transition relation. On input i, the FSM at present state p may transit to next state n and produce output o iff $(i, p, n, o) \in T$. State $r \in S$ represents the initial or reset state.

If at least one transition is specified for each present state and input pair, the FSM is said to be **complete**, otherwise it is **partial**. An FSM where there is at most one transition for given present state and input is called a deterministic FSM (**DFSM**), otherwise it is a nondeterministic finite state machine (**NDFSM**). An NDFSM is a pseudo nondeterministic FSM (**PNDFSM**) [10] if for each triple $(i, p, o) \in I \times S \times O$, there is at most one state n such that $(i, p, n, o) \in T$.

Definition 4.3: An FSM $M' = \langle S', I', O', T', r'\rangle$ is a **submachine** of FSM $M = \langle S, I, O, T, r\rangle$ if $S' \subseteq S$, $I' \subseteq I$, $O' \subseteq O$, $r' = r$, and T' is a restriction of T to the domain $I' \times S' \times S' \times O'$.

Definition 4.4: A complete FSM is said to be of **Moore** type if $(i, p, n, o) \in T$ implies that for all i' there is n' such that $(i', p, n', o) \in T$.[4]

We now introduce the notion of a language associated to an FSM. This is achieved by looking to the automaton underlying a given FSM. For our purposes, we define two related languages: one over the alphabet $I \times O$ and the other over the alphabet $I \cup O$, naturally associated, respectively, with synchronous and asychronous composition.

For a language over $I \times O$, the automaton coincides with the original FSM where all states are made accepting and the edges carry a label of the type (i, o).

For a language over $I \cup O$, the automaton is obtained from the original FSM, by replacing each edge (i, s, s', o) by

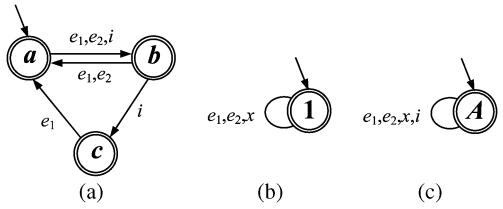

Fig. 3. *Solving an asynchronous equation over regular languages: (a) automaton F_C; (b) automaton F; (c) automaton S_R.*

[4]Notice that this definition allows for NDFSMs of Moore type, contrary to the more common definition of Moore type: for each present state p there is an output o such that all transitions whose present state is p carry the same output o.

the pair of edges $(i, s, (s, i))$ and $(o, (s, i), s')$ where (s, i) is a new node (non-accepting state); all original states are made accepting. The new automaton forces the requirement that an input always precedes an output.

Definition 4.5: A language L is an **FSM language** if there is an FSM M such that the associated automaton $F(M)$ accepts L. Given an FSM $M = \langle S, I, O, T, r\rangle$, one of the two following finite automata may be associated to an FSM (according to whether we assume synchronous or asynchronous composition):

1) $F(M) = \langle S, I \times O, \Delta, r, S\rangle$, where $((i, o), s, s') \in \Delta$ iff $(i, s, s', o) \in T$;
2) $F(M) = \langle S \cup (S \times I), I \cup O, \Delta, r, S\rangle$, where $(i, s, (s, i)) \in \Delta \wedge (o,(s,i), s') \in \Delta$ iff $(i, s, s', o) \in T$.

Definition 4.6: FSM M_A is a **reduction** of FSM M_B, written $M_A \preceq M_B$, iff the language of M_A is contained in the language of M_B. If the languages of M_A and M_B coincide, written $M_A \cong M_B$, then M_A and M_B are **equivalent**.

By definition, an FSM language is a regular language represented by the corresponding automaton. However, it is not the case that each regular language is the language of an FSM, and in [10] we show that for each regular language L over input alphabet I and output alphabet O, there exists the largest subset L^{FSM} of L that is an FSM language. The language of a given FSM over alphabets I and O is contained in L iff it is contained in L^{FSM}.

In order to derive the synchronous (asynchronous) composition of FSMs, we compose the underlying automata (over the alphabet $I \times O$ for the synchronous composition, and over $I \cup O$ for the asynchronous one) and then we transform the obtained automaton into an FSM. When solving the FSM inequation $M_A \bullet M_X \preceq M_C$ or equation $M_A \bullet M_X \cong M_C$, again we solve the equation over the underlying automata. However the largest language solution so obtained is not necessarily an FSM language, so there is an additional step of extracting the largest contained FSM language (generated by a subautomaton of the automaton representing the largest language solution) and transforming its automaton into the corresponding FSM. The detailed procedures for solving synchronous and asynchronous FSM inequations and equations are given in [10]. As in the case of finite automata, an FSM is a solution of an FSM inequation iff the FSM is a reduction of the largest FSM solution (an FSM is a solution of an FSM equation only if the FSM is a reduction of the largest FSM solution).

Notice that the largest FSM solution is unique as a language, but it may be represented by different FSMs (including partial and nondeterministic ones), of which we may choose a unique representative (e.g., a PNDFSM with a minimum number of states).

Next we consider some restricted solutions of FSM inequations and equations which are of theoretical and practical interest.

1) Complete Combinational Solutions: Solutions of equations over FSMs are combinational when the unknown can be replaced by an FSM with a single state (e.g., a combinational winning strategy in a safe game).

A straightforward sufficient condition for the nonexistence of such solutions is based on iterative deleting the states for which there exists an input such that the set of outputs under this input does not intersect with the set of outputs at the initial state (proposed in [62]).

2) Complete and Moore FSM Solutions: We are interested in complete FSMs to model the common case that a hardware/software component is input-enabled, implying that a solution has to be completely specified, since undefined input sequences mean that any output response to this sequence will violate the specification. In [10] Sec 3.1.2, conditions are stated for a regular language (over alphabets $I \times U$ or $I \cup O$) to be the language of a partial or complete FSM, otherwise procedures are given to extract the largest such sublanguage.

Moreover, another requirement of practical interest for FSMs is that each input is followed by an output. For synchronous composition of deterministic FSMs a sufficient condition to enforce such requirement is to have a Moore FSM in each closed loop of the composition. The reason is that a Moore FSM "breaks" the cycle because its outputs depend only on the current state and do not depend directly on the inputs. In [10] p. 43, a procedure is described that given as input an FSM, returns its largest submachine that is a Moore FSM; moreover it is proved that the latter contains (behaviourally) any Moore FSM included in the original FSM.

As an example from [10], given the inequation $M_A \bullet M_X \preceq M_C$, with M_A and M_C shown, respectively, in Fig. 4(a) and 4(b), the largest FSM solution M_X is shown in Fig. 4(c), and the largest Moore FSM solution $Moore(M_X)$ in Fig. 4(d).

In summary, both the procedures for deriving the largest complete FSM solution and the largest Moore FSM solution are based on iterative deleting "bad" states from the largest solution, and thus their complexity is polynomial in the number of states/transitions of the largest solution; if such procedures do not return a solution then such a solution does not exist for the given inequation. Since the set of all complete (or Moore) reductions of the largest solution coincides with the set of reductions of the largest complete (Moore) submachine of the largest solution, any complete (Moore) submachine of the largest complete (Moore) solution is a complete (Moore) solution of the FSM inequality (see Th. 3.9 in [10]). Moreover, any Moore reduction of the largest Moore FSM solution combined with the context FSM yields a complete FSM.

A related notion of *implementability* of interacting machines M_1 and M_2 was reported in [63], where M_1 is implementable with M_2 if there exists a pair of

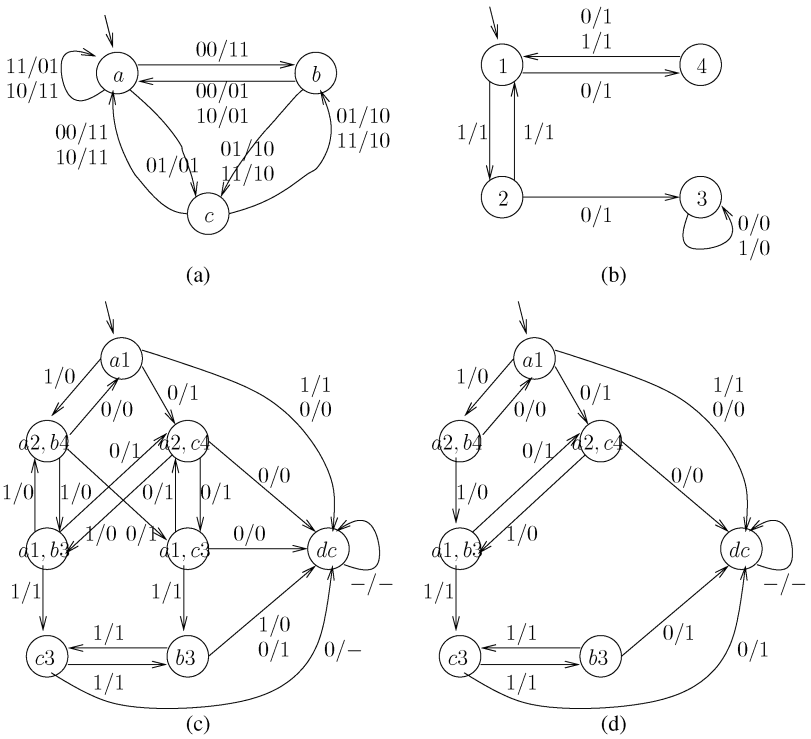

Fig. 4. ***Solving a synchronous inequation over FSMs: largest FSM and largest Moore FSM solutions: (a) FSM M_A; (b) FSM M_C; (c) Largest FSM solution M_X; (d) Largest Moore FSM solution Moore(M_X).***

implementations of M_1 and M_2 such that no combinational loop is created by interconnecting them through the matching variables. A condition for implementability based on the acyclicity of a directed bipartite graph is given; moreover, there is a discussion on how to deal with restricted unimplementable solutions of the largest solution of a synchronous FSM equation. Notice that for deterministic FSMs the assumption that M_1 or M_2 are Moore FSMs is sufficient to guarantee implementability. Related issues were investigated in [64], [65].

3) Compositionally Progressive FSM Solutions: Another restriction of practical interest is that the composition of context and solution be defined for all inputs (input-progressive) or at least for the same set of inputs as the specification, which means that the composed system is required to have no livelocks or deadlocks. The corresponding solutions are called compositionally progressive solutions.

Two procedures for deriving compositionally progressive solutions for synchronous FSM inequations are proposed in [59]. Differently from the largest complete solution and the largest Moore solution, the largest compositionally progressive solution cannot be derived as a submachine of the largest solution of a synchronous FSM inequation. In order to derive the largest compositionally progressive solution as a submachine of the largest solution, the largest solution should be transformed into a so-called *perfect* FSM (the construction was first used in [24] for parallel equations, but the term was coined in [61]). For the composition of the context with a perfect FSM it holds that, for each state (a, m) of the composition, the projection of the language accepted by the state onto the alphabet of the unknown component equals the language accepted by the state m of the perfect FSM. A perfect FSM that is equivalent to the largest solution can be derived by duplicating and splitting states of the largest solution using the following steps: first the composition of the context and largest solution is projected onto the alphabet of the unknown component without merging equivalent states in the projection, and then the language of the obtained automaton is augmented with all the sequences of the language of the largest solution that do not participate in the composition with the given context. The largest compositionally progressive solution is represented by *a submachine of the perfect largest solution*. The same holds for synchronous and asynchronous composition of finite automata [61], [66].

C. Design Automation Support With BALM and BALM-II

Manipulation of finite automata has been extensively studied since the early days of computing. It became more practical to solve problems expressed in terms of automata

theory with the advent of efficient symbolic techniques such as Binary Decision Diagrams (BDDs), satisfiability (SAT) solvers, and AND-INVERTER graphs (AIGs). Based on these techniques, the BALM (Berkeley Automata and Language Manipulation) package aims at providing an experimental environment for efficient manipulation of finite automata in various application domains, e.g., synthesis, verification, control, etc. The environment features the most common automata operations, such as determinization and state minimization, as well as some visualization capabilities. The source code of BALM and its manual are available at [67] and [68]. Recently BALM has been upgraded to BALM-II, adding the specification interfaces and the operations required to solve asynchronous (called parallel in BALM-II) equations. BALM-II subsumes BALM, and its source code and manual are available at [69], [70], to which we refer for extended explanations and examples. The applicability of BALM to finite-state machine synthesis is demonstrated by solving an unknown component problem formulated using language equations.

BALM provides a specialized procedure for resynthesis of sequential circuits [57]. A part of the sequential design to be optimized is selected and temporarily blackboxed. An improved representation of this part can be found by asserting that the behavior of the design with the blackbox is equivalent to the behavior of the original design before blackboxing. As said before, this problem is reduced to language equation solving. A solution of the language equation is computed and converted into a circuit used to replace the blackbox. The resulting circuit structure may improve area and delay and lead to a better implementation in terms of the selected technology. In BALM, this problem is solved using the best known combination of data-structures and algorithms developed for finite automata manipulation. The input is in the form of a circuit. Intermediate computations are based on BDDs representing the partitioned transition relation. The state-transition structure of the solution is obtained in the explicit form as a transition table, with transition conditions and output functions represented compactly using BDDs. The transition structure can be encoded and converted into another circuit that can be used as a missing part in the given circuit. The implementation available in BALM can handle up to millions of states represented implicitly (using BDDs) or explicitly (using the transition table). However, the complementation (determinization) of nondeterministic automata uses explicit enumeration and hence is limited to about 100 K states.

V. CURRENT RESEARCH AND OPEN PROBLEMS

We conclude by pointing out the frontline of research for this design methodology, underlining some interesting open problems.

A. Equations Over ω-Regular Languages

The formulation of the unknown component problem can be extended from the finite regular case to equations over ω-automata. In [10, Ch. 4], it is shown how to extend asynchronous (i.e., parallel) equations to ω-languages described by Büchi automata. Expansion and restriction of ω-languages are defined along with the corresponding ω-expansion and ω-restriction of ω-automata, so that ω-parallel composition of ω-automata can be defined. The largest solution of an asynchronous equation over Büchi automata is then formulated similarly to the finitary regular case.

Moreover, in [10, Ch.18], an extension of BALM is reported to handle synchronous equations over co-Büchi specifications. This extension is motivated by sequential synthesis problems where the objective is to find a strategy, implementable as an FSM, in order to guide a system to a given subset of accepting states with some desirable property and keep it there. Such requirements can be expressed by co-Büchi automata. The synthesis flow is similar to the one for regular (finite-word) automata with the main difference in the last step, where the most general solution is trimmed to obtain FSM solutions that meet the co-Büchi condition. This necessitates a special nonregular method, based on formulating the problem as a SAT instance where each satisfying assignment corresponds to a way of trimming the graph to enforce the desired property.

B. Relations Among Composition Operators

It is an open question whether the conditions H1, H2, and H3 shown in Section III-A to be sufficient to obtain closed-form solutions of language inequations (they had been discussed preliminarily in [71]) are the least restrictive ones, and so are also necessary. Sufficient conditions were discussed also in [72].

Variants of *synchronous composition* are introduced in [73] as *product*, $\times$ (with the comment *sometimes called completely synchronous composition*), and in [74] as *synchronous parallel composition*, $\otimes$. Variants of *parallel composition* are introduced in [73] as *parallel composition*, $\|$ (with the comment *often called synchronous composition*), and in [74] as *interleaving parallel composition*, $\|$; the same operator was called *asynchronous composition* in [40]. These definitions were usually introduced for regular languages and finite automata.

It has also been noticed by Kurshan [74] and Arnold [75] that asynchronous systems can also be modeled with the synchronous interpretation, using null transitions to keep a transition system in the same state for an arbitrary period of time. Kurshan [74] observes that: "While synchronous product often is thought to be a simple-even uninteresting!-type of coordination, it can be shown that, through use of nondeterminism, this conceptually simple coordination serves to model the most general 'asynchronous' coordination, i.e., where processes progress at arbitrary rates relative to one another. In fact the

'interleaving' model, the most common model for asynchrony in the software community, can be viewed as a special case of this synchronous product." More discussion can be found in [76]. A thorough investigation of the composition operators and their relations is still missing.

C. Developing Better Tools

While working on language equations, we tried to achieve the most scalable implementation in order to determine what is the largest size of a problem that can be solved using this methodology. Initially, we used monolithic BDDs to represent the automata, and reached the limits of scalability of BDDs in this sense. Later, we formulated the problem of computing the unknown component using the partitioned representation [57]. This allowed us to increase the size of problems solved, but soon we reached the limit of partitioned BDDs as well. Currently, we are not aware of any method or trick that can increase scalability of the BDD-based implementation for solving language equation problems.

Additionally, we encountered difficulties when trying to find the best solution in the largest set of solutions computed. The difficulty here is that, to evaluate the quality of a solution, it should be extracted, state-assigned, and implemented as a circuit. It is hard to predict the quality of a solution without going through these steps, which is infeasible for all but a relatively small number of solutions. We tried several criteria to extract a good solution, but none of them worked well from the practical stand-point. Nevertheless, when resynthesizing a relatively small component in a circuit, the quality of randomly selected candidates was good enough to lead to some improvements in area.

Looking into the future, we conjecture that more scalable tools for solving language equations can be developed using circuit manipulation and Boolean Satisfiability. For this, a new representation of language automata in the form of circuits should be developed. Next, operations on language automata (such as union, concatenation, intersection) should be carried out by performing circuit transformations. In a similar way, quantification can be done for the circuit representation by cofactoring the circuit and adding new gates. Finally, the set of all solutions should be formulated as a Boolean problem solved using SAT over the resulting circuit. Although the circuit can grow large after all the intermediate circuit transformations, the power of SAT-based methods is that they do not build a complete canonical representation of all solutions as in the case of BDDs. Instead, SAT methods explore a relevant part of the search space to find one feasible solution or prove that none exists. By iterating SAT calls and parameterizing them in different ways, we can find a substantially large subset of solutions satisfying some desirable properties. This is just a blueprint of a future solution, which needs to be researched and developed. However, a similar approach was successful in performing SAT-based don't-care-based optimization of Boolean networks without computing complete don't-cares [77].

Scalability can also be addressed with strategies based on hierarchical decomposition and divide-and-conquer approaches, as in the design of on-chip protocol converters discussed in [26], where each component protocol is represented as a composition of smaller automata, so that there is no need of a monolithic component automaton when deriving a protocol transducer. Finally, we remark that, even though the worst-case computational complexity of these synthesis problems is PSPACE-complete (as mentioned in Section II reporting from [52]), many instances of practical interest may turn out to be more tractable.

D. UCP Versus Other Synthesis Formulations

It is useful to interpret the unknown component problem (UCP) in a logical framework and compare it with other formulations of synthesis problems. It has been pointed out in [78] that UCP for regular languages under synchronous composition can be embedded into WS1S (Weak Second-Order Logic of 1 Successor) formulas, for which there is a decision procedure and tools to compute a minimum deterministic automaton for which a given formula holds [79].

There are two relevant synthesis problems with which to compare UCP: Supervisory control problem (SCP); Reactive synthesis problem (RSP), often known as synthesis from LTL.

The supervisory control problem (SCP) is: given a plant G (modeled by a finite automaton or an ω-automaton or a Petri net), check for the existence of a supervisor S (controller) such that the closed-loop system S/G satisfies a given specification H_{spec} (usually a restriction of G) [73], whose events are classified as uncontrollable or controllable, of which the latter may be disabled by the controller; moreover, find the least restrictive supervisor, if there is one. A common further condition on the supervisor is that it should be nonblocking, i.e., from every reachable state in the closed-loop system there should be a path to an accepting state. If there are many supervisors achieving the goal, the preferred one disables as few events as possible. The theory of supervisory control shows that such supervisor can be found both with a closed-form solution and an iterative computation. The problem can be made more complex by introducing partial observability of the events of the plant. More complex control architectures have been studied (coordinated, distributed, decentralized, hierarchical, see [80]).

The controller topology of SCP is a special case of UCP, however SCP handles also controllable/uncontrollable and observable/unobservable events, whereas in the standard formulation of UCP all events are controllable and observable. Some preliminary discussion on how to model limited controllability in UCP can be found in [10, Ch. 15.3], but the matter requires further investigation. A useful contribution in this direction can be found in [13],

where it is remarked that controllable events can be modeled as outputs and uncontrollable events as inputs.

The reactive synthesis or implementability problem (RSP) (see [81]) is: given an LTL formula ϕ with input and output atomic propositions, check for the existence and, if so, synthesize a controller S as a Moore or Mealy machine such that all behaviours of S satisfy ϕ. Superficially, the two settings (SCP and RSP) are quite different: e.g., the events in SCP are divided as controllable and uncontrollable, whereas in RSP all inputs are uncontrollable; in SCP the plant is given explicitly, and the objective is to find the least restrictive supervisor. However, it has been shown that SCP can be modeled as RSP [82], [83], by defining a version of RSP (called RSCP, reactive synthesis control problem) with the plant modeled as a transition system (see [84]) whose states are partitioned into uncontrollable and controllable. Given the past states up to the current state of the plant, the strategy disables some successors of the current state to restrict the plant modeled by the transition system, and produce the closed-loop system that satisfies the LTL (or CTL or CTL*) specification ϕ. For some CTL formulas ϕ there are unique maximally permissive strategies. Of related interest in this context is also the connection between LTL formulas and equivalent Büchi automata (i.e., recognizing the same ω-language [85]), and the extension of UCP to equations over ω-languages and tractable classes of Büchi automata mentioned in Section V-A.

Roughly speaking, one expects to be able to embed versions of UCP into RSP (as done for SCP versus RSP in [82], [83]), since the former appears to model mainly synthesis for safety properties, however not all the features of UCP appear to be easily mappable. In general, there is space for a cross-fertilization between the two formulations, relying on unifying notions such as the one of *game* (see [86] and [10, Ch. 5.3]), to leverage on the respective strengths in terms of connection topologies, types of composition, expressivity of the specifications. ■

REFERENCES

[1] I. Krueger and R. Mathew, "Component synthesis from service specifications," in *Proc. Scenarios: Models, Transform., Tools*, vol. 3466, *ser. Lecture Notes Comp. Sci.*, S. Leue and T. Syst, Eds., 2005, pp. 255–277. [Online]. Available: http://dx.doi.org/10.1007/11495628_14, Springer Berlin Heidelberg

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming* 2nd ed., Boston, MA, USA: Addison-Wesley, 2002.

[3] I. Crnkovic, J. Stafford, and C. Szyperski, "Software components beyond programming: From routines to services," *IEEE Software*, vol. 28, no. 3, pp. 22–26, May 2011.

[4] G. T. Heineman and W. T. Councill, Eds., Component-based Software Engineering: Putting the Pieces Together. Boston, MA, USA: Addison-Wesley, 2001.

[5] I. Crnkovic, M. Chaudron, and S. Larsson, "Component-based development process and component lifecycle," in *Proc. Int. Conf. Software Eng. Adv.*, Oct. 2006, pp. 44–44.

[6] P. Cox and B. Song, "A formal model for component-based software," in *Proc. EEE Symp. Human-Centric Comput. Languages Environ.*, 2001, pp. 304–311.

[7] O. G. Edmund M. Clarke and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[8] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.

[9] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSMs: Functional Optimization*. Boston, MA, USA: Kluwer Academic Publishers, 1997.

[10] T. Villa, N. Yevtushenko, R. Brayton, A. Mishchenko, A. Petrenko, and A. Sangiovanni-Vincentelli, *The Unknown Component Problem, Theory and Applications*. Berlin, Germany: Springer-Verlag, 2012.

[11] P. Merlin and G. v. Bochmann, "On the construction of submodule specifications and communication protocols," *ACM Trans. Programming Lang. Syst.*, vol. 5, no. 1, pp. 1–25, Jan. 1983.

[12] E. Cerny and M. Marin, "An approach to unified methodology of combinational switching circuits," *IEEE Trans. Comput.*, vol. C-26, no. 8, pp. 745–756, Aug. 1977.

[13] G. Bochmann, "Using logic to solve the submodule construction problem," *Discrete Event Dynamic Syst.*, vol. 23, no. 1, pp. 27–59, 2013. [Online]. Available: http://dx.doi.org/10.1007/s10626-011-0127-6

[14] H. Qin and P. Lewis, "Factorisation of finite state machines under strong and observational equivalences," *Formal Aspects Comput.*, vol. 3, pp. 284–307, Jul.–Sep. 1991.

[15] J. Parrow, "Submodule construction as equation solving CCS," in *Proceedings 7th Conf. Foundations Software Technol. Theoret. Comput. Sci.*, Pune, India, Dec. 17–19, 1987, pp. 103–123. [Online]. Available: http://dx.doi.org/10.1007/3-540-18625-5_46

[16] K. Larsen and L. Xinxin, "Equation solving using modal transition systems," in *Proc. 5th Ann. IEEE Symp. Logic Comput. Sci.*, Jun. 1990, pp. 108–117.

[17] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone, "A modal interface theory for component-based design," *Fundam. Inf.*, vol. 108, no. 1–2, pp. 119–149, Jan. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362088.2362095

[18] N. Lynch and M. Tuttle, "An introduction to input/output automata," *CWI-Quart.*, vol. 2, no. 3, pp. 219–246, Sep. 1989.

[19] L. De Alfaro and T. A. Henzinger, "Interface automata," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 109–120, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/503271.503226

[20] S. S. Lam, "Protocol conversion," *IEEE Trans. Softw. Eng.*, vol. 14, no. 3, pp. 353–362, Mar. 1988. [Online]. Available: http://dx.doi.org/10.1109/32.4655

[21] J. Green and P., "Protocol conversion," *IEEE Trans. Commun.*, vol. 34, no. 3, pp. 257–268, Mar. 1986.

[22] R. Passerone, J. A. Rowson, and A. L. Sangiovanni-Vincentelli, "Automatic synthesis of interfaces between incompatible protocols," in *Proc. Design Autom. Conf.*, 1998, pp. 8–13.

[23] R. Passerone, L. De Alfaro, T. A. Henzinger, and A. L. Sangiovanni-Vincentelli, "Convertibility verification and converter synthesis: Two faces of the same coin," in *Proc. Int. Conf. Comput.-Aided Design*, 2002, pp. 132–139.

[24] R. Kumar, S. Nelvagal, and S. Marcus, "A discrete event systems approach for protocol conversion," *Discrete Event Dynamic Syst.: Theory Appl.*, vol. 7, no. 3, pp. 295–315, Jun. 1997.

[25] H. Hallal, R. Negulescu, and A. Petrenko, "Design of divergence-free protocol converters using supervisory control techniques," in *Proc. 7th IEEE Int. Conf. Electron., Circuits, Syst. (ICECS 2000)*, Dec. 2000, vol. 2, pp. 705–708.

[26] S. Watanabe, K. Seto, Y. Ishikawa, S. Komatsu, and M. Fujita, "Protocol transducer synthesis using divide and conquer approach," in *Proc. Asia and South Pacific Design Autom. Conf. (ASP-DAC'07)*, Jan. 2007, pp. 280–285.

[27] E. Tronci, "Automatic synthesis of controllers from formal specifications," in *Proc. 2nd Int. Conf. Formal Eng. Methods*, Dec. 1998, pp. 134–143.

[28] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Contr. Optimiz.*, vol. 25, no. 1, pp. 206–230, Jan. 1987.

[29] P. Ramadge and W. Wonham, "The control of discrete event systems," *Proc. IEEE*, vol. 77, no. 1, pp. 81–98, Jan. 1989.

[30] R. Kumar, V. Garg, and S. Marcus, "On controllability and normality of discrete event dynamical systems," *Syst. Contr. Lett.*, vol. 17, no. 3, pp. 157–168, Sep. 1991.

[31] R. Kumar and V. Garg, *Modeling and Control of Logical Discrete Event Systems*. New York, NY, USA: Kluwer Academic Publishers, 1995.

[32] A. Overkamp, "Supervisory control using failure semantics and partial specifications," *IEEE Trans. Autom. Control*, vol. 42, no. 4, pp. 498–510, Apr. 1997.

[33] A. Aziz, F. Balarin, R. K. Brayton, M. D. Di Benedetto, A. Saldanha, and A. L. Sangiovanni-Vincentelli, "Supervisory control of finite state machines," in *Proc. Int. Conf. Comput.-Aided Verification*, Jul. 1995, pp. 279–292.

[34] C. Zhou, R. Kumar, and S. Jiang, "Control of nondeterministic discrete event systems for bisimulation equivalence," in *Proc. 2004 Amer. Contr. Conf.*, Jun. 2004, pp. 4488–4492.

[35] G. Barrett and S. Lafortune, "Bisimulation, the supervisory control problem and strong model matching for finite state machines," *Discrete Event Dynamic Syst.: Theory Appl.*, vol. 8, no. 4, pp. 377–429, Dec. 1998.

[36] M. D. Di Benedetto, A. L. Sangiovanni-Vincentelli, and T. Villa, "Model matching for finite state machines," *IEEE Trans. Autom. Control*, vol. 46, no. 11, pp. 1726–1743, Nov. 2001.

[37] W. Chen, J. Udding, and T. Verhoeff, "Networks of communicating processes and their (de-)composition," in *Proc. Math. Program Construction*, vol. 375, *ser. Lecture Notes in Computer Science*, J. van de Snepscheut, Ed., 1989, pp. 174–196. [Online]. Available: http://dx.doi.org/10.1007/3-540-51305-1_10, Springer Berlin Heidelberg

[38] W. Mallon, J. Tijmen, and T. Verhoeff, "Analysis and applications of the XDI model," in *Proc. Int. Symp. Adv. Res. Asynchronous Circuits Syst.*, 1999, pp. 231–242.

[39] R. Negulescu, "Process spaces," in *Proc. 11th Int. Conf. Concurrency Theory (CONCUR 2000)*, vol. 1877, *ser. LNCS*, C. Palamidessi, Ed., 2000, pp. 199–213, Springer-Verlag.

[40] A. Petrenko and N. Yevtushenko, "Solving asynchronous equations.," in *Proc. FORTE*, vol. 135, *ser. IFIP Conf. Proc.*, S. Budkowski, A. R. Cavalli, and E. Najm, Eds., 1998, pp. 231–247, Kluwer.

[41] M. Fujita, Y. Matsunaga, and M. Ciesielski, "Multi-level logic optimization," in *Proc. Logic Synth. Verification*, R. Brayton, S. Hassoun, and T. Sasao, Eds., 2001, pp. 29–63, Kluwer.

[42] E. Sentovich and D. Brand, "Flexibility in logic," in *Proc. Logic Synth. Verification*, R. Brayton, S. Hassoun, and T. Sasao, Eds., 2001, pp. 65–88, Kluwer.

[43] S. Hassoun and T. Villa, "Optimization of synchronous circuits," in *Proc. Logic Synth. Verification*, R. Brayton, S. Hassoun, and T. Sasao, Eds., 2001, pp. 225–253, Kluwer.

[44] J. Kim and M. Newborn, "The simplification of sequential machines with input restrictions," *IRE Trans. Electron. Comput.*, pp. 1440–1443, Dec. 1972.

[45] S. Devadas, "Optimizing interacting finite state machines using sequential don't cares," *IEEE Trans. Comput.-Aided Design*, vol. 10, no. 12, pp. 1473–1484, Dec. 1991.

[46] H.-Y. Wang and R. Brayton, "Input don't care sequences in FSM networks," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1993, pp. 321–328.

[47] H.-Y. Wang and R. Brayton, "Permissible observability relations in FSM networks," in *Proc. Design Autom. Conf.*, Jun. 1994, pp. 677–683.

[48] J.-K. Rho and F. Somenzi, "Don't care sequences and the optimization of interacting finite state machines," *IEEE Trans. Comput.-Aided Design*, vol. 13, no. 7, pp. 865–874, Jul. 1994.

[49] A. Petrenko, N. Yevtushenko, and R. Dssouli, "Testing strategies for communicating finite state machines," in *Proc. IFIP WG 6.1 Int. Workshop Protocol Test Syst.*, T. Mizuno, T. Higashino, and N. Shiratori, Eds., Tokyo, Japan, 1995, pp. 193–208.

[50] F. Ferrandi, F. Fummi, E. Macii, M. Poncino, and D. Sciuto, "Symbolic optimization of interacting controllers based on redundancy identification and removal," *IEEE Trans. Comput.-Aided Design*, vol. 19, no. 7, pp. 760–772, Jul. 2000.

[51] Y. Watanabe and R. Brayton, "The maximum set of permissible behaviors for FSM networks," in *Proc. IEEE Int. Conf. Comput.-Aided Design*, Nov. 1993, pp. 316–320.

[52] K. Rohloff and S. Lafortune, "PSPACE-completeness of modular supervisory control problems," *Discrete Event Dynamic Syst.: Theory Appl.*, vol. 15, no. 2, pp. 145–167, Jun. 2005.

[53] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli, "Solution of parallel language equations for logic synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2001, pp. 103–110.

[54] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli, "Solution of synchronous language equations for logic synthesis," presented at the Biannu. 4th Russian Conf. Foreign Participation Comput.-Aided Technol. Appl. Math., Sep. 2002.

[55] T. Villa, N. Yevtushenko, and S. Zharikova, "Characterization of progressive solutions of a synchronous FSM equation," in *Vestnik, 278, Series Physics*, Sep. 2003, pp. 129–133.

[56] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli, "Equisolvability of series vs. controller's topology in synchronous language equations," in *Proc. Design, Autom. Test Eur. Conf.*, Mar. 2003, pp. 1154–1155.

[57] A. Mishchenko, R. Brayton, J.-H. Jiang, T. Villa, and N. Yevtushenko, "Efficient solution of language equations using partitioned representations," in *Proc. Design, Autom. Test Eur. Conf.*, Mar. 2005, vol. 1, pp. 418–423.

[58] N. Yevtushenko, T. Villa, and S. Zharikova, "Solving language equations over synchronous and parallel composition operators," in *Proc. 1st Int. Workshop Theory Appl. Lang. Equations (TALE 2007)*, M. Kunc and A. Okhotin, Eds., Turku, Finland, Jul. 2007, pp. 14–32.

[59] N. Yevtushenko, T. Villa, R. Brayton, A. Petrenko, and A. Sangiovanni-Vincentelli, "Compositionally progressive solutions of synchronous FSM equations," *Discrete Event Dynamic Syst.*, vol. 18, no. 1, pp. 51–89, Mar. 2008.

[60] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages, Computation*. Boston, MA, USA: Addison-Wesley, 2001.

[61] K. El-Fakih, S. Buffalov, N. Yevtushenko, and G. v. Bochmann, "Progressive solutions to a parallel automata equation," *Theoretical Comput. Sci.*, vol. 362, pp. 17–32, 2006.

[62] S. Tikhomirova Zharikova, "Optimizing Multi Component Discrete Event Systems Based on FSM/Automata Equation Solving," (in Russian)Ph.D. dissertation, Tomsk State Univ., Tomsk, Russia, 2008.

[63] Y. Watanabe, "Logic Optimization of Interacting Components in Synchronous Digital Systems," Ph.D. dissertation, Electron. Re. Lab., Univ. California at Berkeley, Berkeley, CA, USA, Apr. 1994, Tech. Report No. UCB/ERL M94/32.

[64] J. Burch, D. Dill, E. Wolf, and G. DeMicheli, "Modelling hierarchical combinational circuits," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1993, pp. 612–617.

[65] E. Wolf, "Hierarchical Models of Synchronous Circuits for Formal Verification and Substitution," Ph.D. dissertation, Stanford University, Stanford, CA, USA, Sep. 1995, Tech. Rep. CS-TR-95-1557.

[66] N. Yevtushenko, K. El-Fakih, T. Villa, and J.-H. Jiang, "Deriving Compositionally Deadlock Free Components over Synchronous Automata Compositions *Comput. J.*, 2014.

[67] "Software package and documentation," *BALM-RU*. [Online]. Available: http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/balm.html

[68] "Software package and documentation," *BALM*. [Online]. Available: http://embedded.eecs.berkeley.edu/Respep/Research/mvsis/software.html

[69] "Software package and documentation," *BALM-II*. [Online]. Available: http://esd.scienze.univr.it/index.php/it/balm-ii.html

[70] G. Castagnetti, M. Piccolo, T. Villa, N. Yevtushenko, A. Mishchenko, and R. K. Brayton, "Solving Parallel Equations With Balm-ii," EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2011-102, Sep. 2011. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-102.html

[71] N. Yevtushenko, T. Villa, R. Brayton, A. Mishchenko, and A. Sangiovanni-Vincentelli, "Composition operators in language equations," presented at the Int. Workshop Logic Synthesis, Jun. 2004.

[72] L. Kari, "On language equations with invertible operations," *Theor. Comput. Sci.*, vol. 132, no. 1–2, pp. 129–150, Sep. 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90230-5

[73] C. C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems, 2nd ed, Berlin, Germany: Springer-Verlag, 2007.*

[74] R. Kurshan, *Computer-Aided Verification of Coordinating Processes*. Princeton, NJ, USA: Princeton Univ. Press, 1994.

[75] A. Arnold, *Finite Transition Systems: Semantics of Communicating Systems*. Englewood Cliffs, NJ, USA: Prentice Hall, 1994.

[76] R. Kurshan, M. Merritt, A. Orda, and S. Sachs, "Modelling asynchrony with a synchronous model," *Formal Methods Syst. Design*, vol. 15, no. 3, pp. 175–199, Nov. 1999.

[77] A. Mishchenko, R. Brayton, J.-H. R. Jiang, and S. Jang, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 4, pp. 34:1–34:23, Dec. 2011. [Online]. Available: http://www.eecs.berkeley.edu/alanmi/publications/2011/trets11_mfs.pdf

[78] A. Aziz, F. Balarin, R. Brayton, and A. Sangiovanni-Vincentelli, "Sequential synthesis using S1S," *IEEE Trans. Comput.-Aided Design*, vol. 19, no. 10, pp. 1149–1162, Oct. 2000.

[79] N. Klarlund, "Mona & Fido: The logic-automaton connection in practice," in *Proc. 11th Int. Workshop Comput. Sci. Logic (CSL'97)*, vol. 1414, *ser. LNCS*, M. Nielsen and W. Thomas, Eds., 1998, pp. 311–326, Springer-Verlag.

[80] J. van Schuppen and T. Villa, *Coordination Control of Distributed Systems*, Berlin, Germany: Springer-Verlag, 2015. [Online]. Available: http://books.google.it/books?id=I1qaoAEACAAJ

[81] A. Pnueli and R. Rosner, "On the synthesis of a reactive module," in *Proc. 16th ACM SIGPLAN-SIGACT Symp. Principles Program. Lang., ser. POPL'89*, New York, NY, USA, 1989, pp. 179–190. [Online]. Available: http://doi.acm.org/10.1145/75277.75293

[82] R. Ehlers, S. Lafortune, S. Tripakis, and M. Vardi, "Reactive Synthesis vs. Supervisory Control: Bridging the gap," EECS Department, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2013-162, Sep. 2013. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-162.html

[83] R. Ehlers, S. Lafortune, S. Tripakis, and M. Y. Vardi, "Bridging the gap between supervisory control and reactive synthesis: Case of full observation and centralized control," in *Proc. 12th Int. Workshop Discrete Event Syst. (WODES 2014)*, Cachan, France, May 14–16, 2014, pp. 222–227. [Online]. Available: http://dx.doi.org/10.3182/20140514-3-FR-4046.00018

[84] O. Kupferman, P. Madhusudan, P. Thiagarajan, and M. Vardi, "Open systems in reactive environments: Control and synthesis," in *Proc. CONCUR 2000 Concurrency Theory*, vol. 1877, *ser. Lecture Notes in Computer Science*, C. Palamidessi, Ed., 2000, pp. 92–107. [Online]. Available: http://dx.doi.org/10.1007/3-540-44618-4_9, Springer Berlin Heidelberg

[85] M. Y. Vardi and P. Wolper, "Reasoning about infinite computations," *Inf. Comput.*, vol. 115, pp. 1–37, 1994.

[86] S. Krishnan, "ω-Automata, Games and Synthesis," Ph.D. dissertation, EECS Dept., Univ. California, Berkeley, CA, USA, 1998. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1998/3445.html, tech. Report No. UCB/ERL M98/30.

ABOUT THE AUTHORS

Tiziano Villa studied mathematics at the Universities of Milano (B.Sc. in Mathematics), Pisa (Postgraduate Degree in Computational Mathematics), and Cambridge (UK, DAMTP, Mathematical Tripos Part III).

From 1980 to 1985, he worked as a Computer-Aided Design Specialist in the integrated circuits division of the CSELT Labs, Torino, Italy, and then from 1986 to 1996, he was a Research Assistant with the Electronics Research Laboratory, University of California, Berkeley. From 1997 to 2001, he was a Research Scientist with the PARADES Labs, Rome, Italy. From 2002 to 2006, he was an Associate Professor with the Department of Electrical, Industrial, and Mechanical Engineering (DIEGM), Universita' degli Studi di Udine, Italy. Since October 2006, he has been a Full Professor with the Department of Computer Science, Universita' degli Studi di Verona, Italy. His research interests include computer-aided design of digital circuits (especially logic synthesis), formal verification, cyberphysical systems, and automata theory. He coauthored the books: "Synthesis of FSMs: Functional optimization," Kluwer (now Springer), 1997, "Synthesis of FSMs: Logic optimization," Kluwer (now Springer), 1997, and "The Unknown Component Problem: Theory and Applications" (Springer, 2012); he coedited the book "Coordination Control of Distributed Systems" (Springer, 2015).

Dr. Villa was awarded the Tong Leong Lim Predoctoral Prize at the EECS Department of the University of California, Berkeley, in May 1991.

Alexandre Petrenko received the Ph.D. degree from Riga Polytechnic Institute, USSR, in 1974.

He is a Lead Researcher of Computer Research Institute of Montreal, CRIM, Canada, since 1996. Between 1992 and 1996, he was a Visiting Professor at Universite' de Montreal. Until 1992, he was the Director of the department of research in computer networks of the Institute of Electronics and Computer Science in Riga, URSS. Between 1979 and 1982, he worked in Computer Network Task Force of the International Institute for Applied Systems Analysis, IIASA, in Austria. His research interests include automata theory, model-driven software engineering and model based testing. Alexandre has published more than 200 research papers and has given numerous invited talks.

Dr. Petrenko received the Best Paper Award of the following conferences: the IFIP FORTE/PSTV Conference, 1999; the 17th IFIP International Conference on Testing of Communicating Systems, 2005; the IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012. He led numerous industrial research projects in collaboration with companies, such as Siemens, FranceTelecom, SAP, Ericsson, CAE, Bombardier, and GM.

Nina Yevtushenko received the Ph.D. degree in computer science from Saratov State University, Saratov, Russia, in 1983.

Until 1991, she worked as a Senior Researcher with the Siberian Scientific Institute of Physics and Technology. In 1991, Nina Yevtushenko joined Tomsk State University as a Professor and presently, she leads a research team working on the synthesis and analysis of discrete event systems. She stayed as a Visiting Researcher/Professor in the Universite' de Montreal (Canada), the University of Ottawa (Canada), the University of Verona (Italy), and the Institut National des Telecommunications d'Evry (France). She has published more than 100 research papers and has been supervising a number of Russian international research projects. She currently serves as a program committee member for a number of international workshops and conferences. Her research interests include formal methods, automata theory, distributed systems, protocol, and software testing.

Alan Mishchenko received the M.S. degree from the Moscow Institute of Physics and Technology, Moscow, Russia, in 1993, and the Ph.D. degree from the Glushkov Institute of Cybernetics, Kiev, Ukraine, in 1997.

From 1998 to 2002, he was an Intel-sponsored Visiting Scientist at Portland State University, Portland, OR, USA. Since 2002, he has been a Research Engineer in the EECS Department at University of California, Berkeley, CA, USA. His current research interests include developing computationally efficient methods for synthesis and verification.

Dr. Mishchenko was a recipient of the D.O. Pederson TCAD Best Paper Award in 2008, and the SRC Technical Excellence Award in 2011, for his work on ABC.

Robert Brayton (Fellow, IEEE) received the Ph.D. degree in mathematics from the MIT, Cambridge, MA, USA, in 1961.

He was a Member of the Mathematical Sciences Department of the IBM T. J. Watson Research Center, New York, NY, USA. In 1987, he joined the EECS Department at the University of California, Berkeley, CA, USA, where he is currently a Professor in the Graduate School.

Dr. Brayton was a recipient of the IEEE Emanuel R. Piore Award in 2006, the ACM Kanallakis Award in 2006, the European DAA Lifetime Achievement Award in 2006, the EDAC/CEDA Phil Kaufman Award in 2007, the D.O. Pederson Best Paper Award in 2008, the ACM/IEEE A. Richard Newton Technical Impact in EDA Award in 2009, the Iowa State University Distinguished Alumnus Award in 2010, the SRC Technical Excellence Award in 2011, and the ACM/SIGDA Pioneering Achievement Award in 2011. He also held the Buttner Chair and the Cadence Distinguished Professorship of Electrical Engineering at Berkeley. He is a Member of the National Academy of Engineering.