Journal of Visual Languages and Computing 24 (2013) 350–364

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

Vibes: A visual language for specifying behavioral requirements of algorithms ☆, ☆ ☆

Gürcan Güleşir *, Lodewijk Bergmans, Mehmet Akşit, Klaas van den Berg

Department of Computer Science, University of Twente, Germany

ARTICLE INFO

Article history:
Received 29 April 2010
Received in revised form
18 August 2013
Accepted 25 August 2013
Available online 4 September 2013

Keywords:
State-transition diagrams
Formal methods
Software specification
Visual formalisms

ABSTRACT

Manually verifying the behavior of software systems with respect to a set of requirements is a time-consuming and error-prone task. If the verification is automatically performed by a model checker however, time can be saved, and errors can be prevented. To be able to use a model checker, requirements need to be specified using a formal language. Although temporal logic languages are frequently used for this purpose, they are neither commonly considered to have sufficient usability, nor always naturally suited for specifying behavioral requirements of algorithms. Such requirements can be naturally specified as regular language recognizers such as deterministic finite accepters, which however suffer from poor evolvability: the necessity to re-compute the recognizer whenever the alphabet of the underlying model changes. In this paper, we present the visual language *Vibes* that both is naturally suited for specifying behavioral requirements of algorithms, and enables the creation of highly evolvable specifications. Based on our observations from controlled experiments with 23 professional software engineers and 21 M.Sc. computer science students, we evaluate the usability of Vibes in terms of its understandability, learnability, and operability. This evaluation suggests that Vibes is an easy-to-use language.

1. Introduction

New generations of large-scale and complex embedded systems such as wafer scanners [4], medical MRI[1] scanners, and electron microscopes are rarely developed from scratch [23]. Instead, engineers continuously modify older generations to develop new ones. Therefore, evolvability is one of the key quality factors that determine the commercial success or failure of large-scale and complex embedded systems.

During the evolution of such systems, manually verifying the software behavior with respect to a set of requirements is a time-consuming and error-prone task [10]. If the verification is automatically performed by a model checker [6] however, time can be saved, and errors can be prevented [10].

To be able to use a model checker, requirements need to be specified using a formal language. Although temporal logic languages such as LTL [6] are suitable for specifying behavioral requirements, these languages are commonly considered to have insufficient *usability*.[2] For example, Hatcliff and Dwyer [13] state the following: "Although model-checker property specification languages are built on the theoretically elegant temporal logics, practitioners

☆ This work has been carried out as a part of the Ideals and Darwin projects, under the management of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the Senter and Bsik programs.

☆☆ This paper has been recommended for acceptance by Shi Kho Chang.

* Corresponding author. Tel.: +49 89 382 13810; fax: +49 89 382 52691.

E-mail addresses: gulesirg@cs.utwente.nl, ggulesir@hotmail.com, guercan.guelesir@bmw.de (G. Güleşir), bergmans@cs.utwente.nl (L. Bergmans), aksit@cs.utwente.nl (M. Akşit), k.g.vandenberg@cs.utwente.nl (K. van den Berg).

[1] Magnetic resonance imaging.

[2] A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users [2].

1045-926X/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.jvlc.2013.08.005

and even researchers find it difficult to use them to accurately express complex event-sequencing properties. Once written, the specifications are often hard to read and debug".

Temporal logic languages are designed for specifying behavioral requirements of non-terminating systems, which have infinite executions. Therefore, these languages do not have a built-in construct that can identify the last step of a finite execution. Consequently, temporal logic languages are not always *naturally suited* for specifying behavioral requirements of algorithmic systems, which have finite executions. Nevertheless, model checker tool developers can overcome this limitation by implementing an artificial termination construct that can be used in a transparent way.

Behavioral requirements of algorithmic systems can be naturally specified using regular language recognizers, such as deterministic finite accepters or regular expressions [18]. The downside of using such a recognizer is however, poor *evolvability*: the necessity to recompute the recognizer whenever the alphabet of the model of the underlying system changes.[3] This necessity arises from the fact that the recognizer shares the same alphabet with the model of the underlying system.

Unlike regular language recognizers, temporal logic formulas do not suffer from the evolvability problem explained above; because such a formula does not depend on the alphabet of the model of the underlying system. As a result, the formula does not need to be updated upon a modification of the alphabet.

In the context of this paper, we say that a specification is *highly evolvable*, if and only if the specification does not need to be changed as long as the behavioral requirement it represents does not change. This definition of highly evolvable specifications also entails that such a specification does not need to be changed whenever the underlying model changes. In other words, a highly evolvable specification does not depend on the underlying model but only on the requirement it represents.

The goal of this research is to come up with an *easy-to-use* language that both is *naturally suited* for specifying behavioral requirements of algorithms, and enables the creation of *highly evolvable* specifications. To reach this goal, we introduce a visual language *Vibes (VIsual BEhavior Specifications)* that combines the strengths and eliminates the weaknesses mentioned above of temporal logic languages and regular language recognizers. Since such a language has not been presented in the literature to our best knowledge, we consider Vibes as the contribution of this paper.

At a first glance, a Vibes diagram is similar to a traditional state-transition diagram. However, Vibes diagrams are distinguished by a key feature that we call *Context-Sensitive Wildcard* (*CSW*). Intuitively, a CSW is a transition that stands for an infinite set of transitions, such that the elements of this set is determined by the 'context' of the CSW. In this paper, we formally define CSW as the key feature of Vibes. We present the syntax, formal semantics, expressive power, and an empirical evaluation of Vibes, which reveals the theoretical and practical implications of using CSWs in visual specifications of software behavior.

The remainder of this paper is structured as follows: In Section 2, we present an example application, survey some of the related work, and define the goal of this research. In Section 3, we informally present Vibes, and explain why it brings us to the goal of this research. In Sections 4, 5, 6, and 7 we present the syntax, formal semantics, and expressive power of Vibes. Section 8 contains empirical results that provide insights into the usability of Vibes. The remaining sections contain the related work, conclusions, and future work.

2. An example: the authentication algorithm of an ATM

An automated teller machine (ATM) is an embedded system that can authenticate a user based on a bank card and password. Upon successful authentication, the user can access his or her bank account(s) to perform various actions such as withdrawing money. In this section, we present two realistic requirements for the authentication algorithm of a generic ATM, and create a model that fulfills these requirements. In addition, we discuss some of the related work, and define the goal of this research.

Two realistic requirements for the authentication algorithm of a generic ATM can be as follows:

R1 If a user cannot enter the correct password within three attempts during an authentication session, then the ATM must block the card (i.e., retain the card and prevent its further use).

R2 If the ATM blocks a card during an authentication session, then the ATM must immediately terminate the session.

To create a model of the authentication algorithm that fulfills these requirements, we need to choose a suitable modeling language. Statecharts [1,12] is suitable in our case; because the authentication algorithm can be seen as a reactive system [11] where the user can perform an input action such as entering a password, and depending on the current state, the algorithm may react by performing an output action such as authenticating the user. Accordingly, we can model the authentication algorithm using statecharts, as shown in Fig. 1.

This model can be informally explained as follows: If a user inserts a card into the card slot of the ATM, then the ATM reads the card. If the card is invalidated, then the ATM ejects the card, and terminates the session. If the card is validated, then the ATM sets the number of authentication attempts to zero, and then requests the password of the user. After the password is entered, the ATM increments the number of authentication attempts, and then tries to authenticate the user. If the user is authenticated, then the algorithm terminates as normal. If the authentication attempt fails (i.e., the password is wrong), then the ATM allows the user to re-enter the password at most two

[3] An update is not necessary upon removing a symbol from the alphabet.

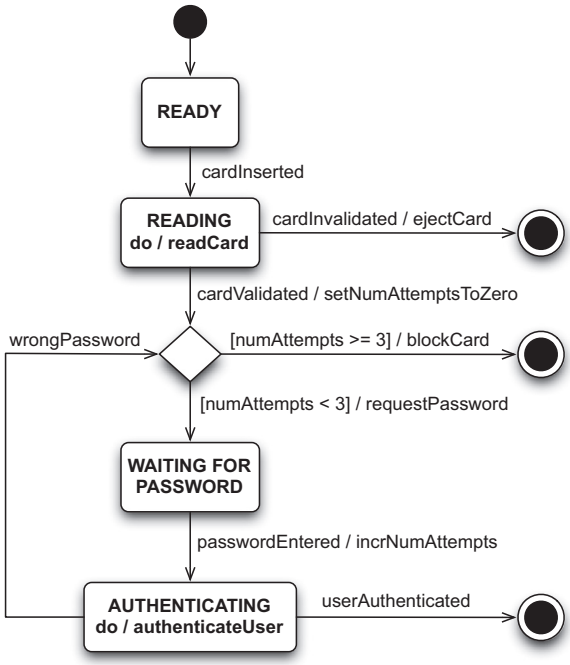

Fig. 1. The authentication algorithm of a generic ATM, modeled as a UML statechart.

additional times. If the user cannot enter the right password upon two additional trials, then the ATM blocks the card.

For the sake of simplicity in this paper, the model shown in Fig. 1 does not consider the possibility that the user requests to cancel the process and eject the card after it has been validated. ATMs usually allow this. In such a case, the reset of the number of attempts to zero might provide a vulnerability, where a malicious user cancels after one failed attempt and then retries the card with the same number of attempts available.

After a model of a system is created, the model typically needs to be checked against its requirements. A manual check is both time consuming and error prone [10]. If a model checker tool is used instead, people's time can be saved and errors can be prevented [10]. To be able to use a model checker however, requirements need to be formalized. This can be done in two steps: In the first step, the requirements can be rewritten in terms of the action[4] names that appear in the model. In the second step, the rewritten requirements can be specified using a formal language. Below, we perform the first step by rewriting the requirements R1 and R2, in terms of the action names that appear in Fig. 1.

R1′ In each possible execution of the authentication algorithm, immediately after the third occurrence of the wrongPassword action, there must be an occurrence of the blockCard action.

R2′ In each possible execution of the authentication algorithm, if there is an occurrence of the blockCard action, then this must be the last occurrence of the execution.

In Sections 2.1 and 2.2, we perform the second step of the formalization, by specifying R1′ and R2′ using alternative formalisms. In these sections, we also evaluate these formalisms for the purpose of specifying the behavioral requirements of algorithms. Based on the outcome of this evaluation, we define the goal of this research, in Section 2.3.

2.1. Temporal logic languages

Temporal logic languages such as LTL [6], CTL [6], CTL* [6], and FLTL [7,17] are suitable for specifying the temporal or logical properties of reactive systems. Therefore, we can formalize the requirements of the authentication algorithm using these languages. For example, R1′ can be formalized as the following LTL specification: *spec* = (((*eventually* wrongPassword) ⇒ (*eventually* wrongPassword)) ⇒ (*eventually* wrongPassword)) ⇒ (*next* blockCard). Based on this example specification, below we explain a key benefit of using temporal logic languages for specifying behavioral requirements.

Note that the set of action names that appear in *spec* is equal to the set of action names that appear in R1′. Therefore, *spec* and R1′ are at the same level of abstraction, which has the following benefit: As long as R1′ does not change, one does not need to update *spec* during the evolution of the authentication algorithm modeled in Fig. 1. Thus, *spec* is highly evolvable, which is also true in general: temporal logic languages enable us to create highly evolvable specifications.

In the remainder of this section, we discuss the disadvantages of using temporal logic languages for specifying the behavioral requirements of algorithms.

In temporal logic languages, there is no built-in notion of 'last action occurrence', because these languages are designed for specifying the requirements of infinite executions. Hence, temporal logic languages are not always natural for specifying the requirements of algorithms, which have finite executions. For example, R2′ is a desired property of the last action occurrence in the possible executions of the authentication algorithm. To specify R2′ using a temporal logic language, possible executions of the authentication algorithm, which are finite, need to be transformed to infinite executions. For example, the finite sequence ⟨cardInserted, readCard, cardInvalidated, ejectCard⟩ of action occurrences represents a possible execution of the authentication algorithm that we modeled as the statechart shown in Fig. 1. This sequence can be transformed to the infinite sequence ⟨cardInserted, readCard, cardInvalidated, ejectCard, ⊙, ⊙, ⊙, …⟩, where a ⊙ represents an occurrence of a pseudo action that is reserved for marking the end of a finite sequence of action occurrences. Assuming that all possible executions of the authentication algorithm are transformed to infinite executions as explained above, R2′ can be specified using LTL as follows: (*eventually* blockCard) ⇒ (*next* ⊙). Nevertheless, this specification is less natural and concise,

[4] For the sake of simplicity in this paper, we use the term *action* to uniformly refer to the terms *action*, *event*, and *activity*, which actually represent distinct concepts in the terminology of statecharts. Such a distinction is not necessary in the context of this paper.

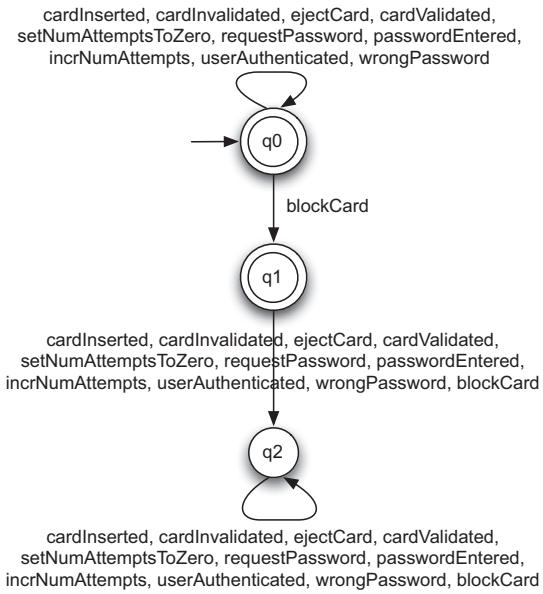

Fig. 2. The state-transition diagram of a DFA that is a formal specification of the requirement R2′ defined in Section 2.

due to the usage of $\Rightarrow$ (*next* $\odot$). Thus, temporal logic languages are not always natural for specifying properties of algorithms, since these languages are designed for specifying properties of infinite executions.

According to Hatcliff and Dwyer [13], one of the major problems that are currently preventing the successful application of model checking technology to software is "the requirement specification problem: the difficulty of expressing software requirements in the temporal specification languages of the existing model-checking tools. Although model-checker property specification languages are built on the theoretically elegant temporal logics, practitioners and even researchers find it difficult to use them to accurately express complex event-sequencing properties. Once written, the specifications are often hard to read and debug" [13].

Based on the discussion in this section, we can conclude that temporal logic languages enable us to create highly evolvable specifications; but these languages are neither always naturally suited for specifying the behavioral requirements of algorithms, nor are they commonly considered to have sufficient usability.

2.2. Regular language recognizers

In the theory of computation [18], a regular language is a set of finite sequences of symbols from a finite alphabet, such that the elements of this set are determined by a regular language recognizer: deterministic finite accepter (DFA), non-deterministic finite accepter (NFA), regular expression, or regular grammar. Since regular language recognizers accept or reject finite sequences of symbols, they are naturally suited for specifying requirements on the executions of algorithms, including the requirements on the last action occurrences. For example, using the action names shown in Fig. 1, R2′ can be specified as the DFA whose state-transition diagram is shown in Fig. 2.

A disadvantage of using DFAs however, is as follows: as visible in Fig. 2, the alphabet of the DFA consists of all the action names that appear in the model shown in Fig. 1. That is, the DFA shares the same alphabet with the model. This is a drawback, because each time the set of action names that appear in the underlying model changes, e.g., a new action name is added or an existing action name is renamed, the transitions of the DFA need to be recomputed, e.g., new transitions need to be added or existing transitions need to be renamed, although the actual requirement represented by the DFA may remain intact. Other types of regular language recognizers also have this drawback, because they need to share the same alphabet with the underlying model. Thus, regular language recognizers do not allow us to create highly evolvable specifications.

Based on the discussion in this section, we can conclude that regular language recognizers are naturally suited for specifying behavioral requirements of algorithms, but these languages do not allow us to create highly evolvable specifications.

2.3. The goal of this research

The goal of this research is to come up with a language that (a) is naturally suited for specifying behavioral requirements of algorithms, (b) enables the creation of highly evolvable specifications, and (c) is easy to use. In Section 3, we intuitively explain this new language, which we call Vibes. In addition, we discuss why Vibes brings us to the goal stated above.

3. Vibes: VIsual BEhavior Specifications

The requirement R2′ (see Section 2) can be formally specified using Vibes, as shown in Fig. 3. Informally speaking, Vibes diagrams can be seen as state-transition diagrams augmented with wildcard transitions (e.g., the $\star$−transition shown in Fig. 3). In this section, we explain Vibes using Fig. 3 as an illustrative example.

A Vibes diagram is a pattern of nodes and edges. For a given finite sequence of action occurrences, the edges match the occurrences, whereas the nodes determine if the sequence is matched by the pattern.

Regarding the pattern shown in Fig. 3, the matching of a given sequence starts at the node q0; since its stereotype contains initial. Such a node is called *initial node*. There is exactly one initial node in each pattern.

The $\star$−labelled edge originating from q0 matches each occurrence from the beginning of a sequence, until an occurrence of blockCard is reached. This "until" condition

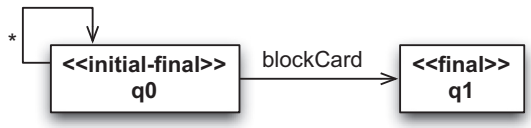

Fig. 3. A Vibes diagram that represents the requirement R2′, which is defined in Section 2.

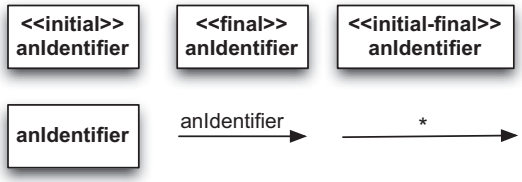

Fig. 4. The elements of the notation of Vibes.

is due to the existence of the blockCard-labelled edge originating from the same node (i.e., q0). In a pattern, no two edges originating from the same node have the same label.

In general, a $\star-$labelled edge matches an occurrence, if and only if this occurrence cannot be matched by the other edges *originating from the same node*. That is, the matching of a $\star-$labelled edge is 'sensitive' to the other edges originating from the same node. Therefore, a $\star-$labelled edge e is a *Context-Sensitive Wildcard (CSW)*, where the context is the set of labels of the other edges whose source node is the same as the source node of e.

During the matching of a given sequence of action occurrences, if the first occurrence of blockCard is reached, then this occurrence is matched by the edge labelled with blockCard (see Fig. 3). If there are no additional occurrences in the sequence, then the sequence *terminates* at q1. If the sequence terminates at q1, then the sequence is matched by the pattern, because q1 contains final in its stereotype.

In general, a given sequence is *matched by a pattern*, if and only if the sequence terminates at a node that has final in its stereotype. We call such a node *final node*. There can be zero or more final nodes in a pattern.

If a given sequence has additional occurrences after the first occurrence of blockCard, these additional occurrences cannot be matched, because there is no edge originating from q1 (see Fig. 3). Consequently, such a sequence is not matched by the pattern shown in Fig. 3.

Based on the explanation so far in this section, one can conclude that the pattern shown in Fig. 3 matches a given sequence, if and only if either the sequence does not have any occurrence of blockCard, or the first occurrence of blockCard is the final element of the sequence. Hence, the Vibes diagram shown in Fig. 3 is a formal specification of R2′, which is defined in Section 2.

Vibes brings us to the goal stated in Section 2.3, due the following reasons:

- Vibes diagrams 'work with' *finite* sequences of actions. Therefore, Vibes is naturally suited for specifying behavioral properties of algorithms. In this respect, Vibes is similar to regular language recognizers.
- While drawing a Vibes diagram to specify a requirement, one can use CSWs to abstract from the action names that do not appear in the requirement but in the underlying system. Consequently, one does not need to maintain a Vibes diagram during the evolution of the underlying system, as long as the requirement does not change. Therefore, Vibes diagrams are highly evolvable. In this respect, Vibes is similar to temporal logic languages.
- Vibes is a visual language that is similar to state-transition diagrams, which are widely used by practitioners. This fact and the empirical evidence presented in Section 8 suggest that Vibes is an easy-to-use language. In Section 8.3, we provide a detailed evaluation of Vibes, from a usability perspective.

Having intuitively explained Vibes in this section, we now precisely define the notation, syntax, and formal semantics of Vibes, in the upcoming sections.

4. The notation and syntax of Vibes

In Fig. 4, the notational elements of Vibes are depicted.

The rectangles are *nodes*, and the arrows are *edges*. To explain these elements, we use the terms "alphabet" and "string", which are defined in [18] as follows: a finite and non-empty set of symbols is called *alphabet*. A finite sequence of symbols from an alphabet is called *string*. A *Vibes identifier* is a string consisting of alphanumeric symbols.

Nodes In Fig. 4, the node with the stereotype initial is called an *initial node*. The node with the stereotype final is called a *final node*. The node with the stereotype initial-final is called an *initial-final node*, which is both an initial and a final node. The node without any stereotype is called a *plain node*. The anIdentifier labels on the nodes are placeholders of Vibes identifiers that are the names of the nodes.

Edges The arrow with the label anIdentifier is an edge where anIdentifier is the placeholder of an action name. The edge with the label $\star$ is a *context-sensitive wildcard (CSW)* .

Reserved words initial, initial-final, final, and $\star$ are the reserved words [21] of Vibes. Each of these reserved words has a mathematical meaning defined in Section 5.

Syntax A Vibes diagram has (a) either one initial node or one initial-final node, (b) zero or a finite number of final nodes, (c) zero or a finite number of plain nodes, and (d) zero or a finite number of edges. Each edge has a source node, target node, and label. No two edges have both the same source node and the same label.

5. Formal semantics of Vibes

An arguably obvious way to provide formal semantics for Vibes would be to define an algorithm that takes a Vibes diagram together with a model of the corresponding system as inputs, and then constructs a DFA that (a) denotes the Vibes diagram, and (b) shares the same alphabet with the model. For example, if we would provide the Vibes diagram shown in Fig. 3 together with

the model shown in Fig. 1 as the inputs, then the algorithm would construct the DFA whose state-transition diagram is shown in Fig. 2.

A DFA that is constructed as explained above would typically have a larger number of transitions compared to the number of edges of the corresponding Vibes diagram because for each action name that appears in the model but not in the Vibes diagram, the DFA would have $|Q|$ additional transitions, where Q denotes the set of states of the DFA. For example, the DFA whose state-transition graph is shown in Fig. 2 has 33 transitions, whereas the corresponding Vibes diagram (Fig. 3) has two edges. The arguably disproportionate difference between these numbers is due to the 10 action names that appear in the model (Fig. 1) but not in the Vibes diagram (Fig. 3).

Having to deal with such a disproportionate number of transitions would create storage and performance bottlenecks for the algorithms that would need to exhaustively visit each transition of a DFA. An example of such an algorithm is presented in [9].[5] This algorithm performs an intersection emptiness check for multiple Vibes diagrams, to determine the feasibility of a model that fulfills all of the requirements represented by the diagrams. To be able to do the emptiness check, the algorithm constructs a product automaton of the input automata that denote the diagrams. This construction requires the algorithm to visit each transition of each input automaton at least once.

To avoid the excessive number of transitions and the consequent storage and performance bottlenecks explained above, we define the semantics of Vibes using a formalism that is a variant of DFA. In Section 5.1, we present this alternative formalism.

5.1. Deterministic Abstract Recognizer (DAR)

In this section, we introduce a formalism called *deterministic abstract recognizer* (*DAR*), which we use for defining the semantics of Vibes, in Section 5.2. A *DAR* is a variant of a DFA. The key difference between a DFA and a *DAR* is as follows: a DFA with an alphabet Σ either accepts or rejects a finite sequence of symbols, provided that the sequence is an element of Σ^*; whereas a *DAR* either accepts or rejects *any* finite sequence of symbols. To precisely explain this difference, we first need to formally define *DAR*:

A *DAR* M is a quintuple $\langle Q, \Sigma_a, \delta, q_0, F\rangle$, where

- $Q = \Omega \cup \{q_t\}$ is a finite set of *states*, where Ω is the set of *user-defined states*, q_t is the *default trap state*, and $q_t \notin \Omega$.
- $\Sigma_a = \Sigma_b \cup \{\#\}$ is the *abstract input alphabet*, where Σ_b is a finite set of symbols such that $\# \notin \Sigma_b$. Σ_b is called the *base input alphabet*. # is a reserved symbol that will be explained in this section.
- $\delta : Q \times \Sigma_a \to Q$ is a total function called *transition function*. $\forall a \in \Sigma_a(\delta(q_t, a) = q_t)$.
- $q_0 \in \Omega$ is the *initial state*.
- $F \subseteq \Omega$ is a set of *final states*.

To explain how M accepts or rejects a given sequence of symbols, we use the following terms: the set of all possible symbols is called the *universal set of symbols*, and this set is denoted by Υ. A finite sequence of symbols from Υ is called *strand*.[6] ϵ denotes the *empty strand* (i.e., the strand that contains no symbol). If w and x are strands, then wx denotes the strand obtained by concatenating w and x. Υ^* denotes the set of strands obtained by concatenating zero or more symbols from Υ. Note that any alphabet (see Section 4) is a proper subset of Υ, and Υ^* is the set of all possible finite sequence of symbols.

Let $q \in Q$, $a \in \Upsilon$, and $w \in \Upsilon^*$. The function $\delta^* : Q \times \Upsilon^* \to Q$ is called *extended transition function*, which is recursively defined as follows: $\delta^*(q, \epsilon) = q$, and

$$\delta^*(q, wa) = \begin{cases} \delta(\delta^*(q, w), a) & \text{if } a \in \Sigma_b \\ \delta(\delta^*(q, w), \#) & \text{if } a \notin \Sigma_b \end{cases}$$

M accepts w if and only if $\delta^*(q_0, w) \in F$. *M rejects w* if and only if $\delta^*(q_0, w) \notin F$. The asymptotic time complexity of the extended transition function is $O(|w|)$, where $|w|$ denotes the number of symbols in w. Note that the definition of the extended transition function provides the semantics of the # symbol. We call the # symbol *wildcard*, because it matches any symbol in $\Upsilon \backslash \Sigma_b$.

$L(M) = \{w \in \Upsilon^* | \delta^*(q_0, w) \in F\}$ is the *language* of M. A set L of strands is an *Open Regular Language (ORL)*, if and only if there is a DAR M such that $L(M) = L$.

If L is an ORL, then the strands in L consist of symbols from Υ; i.e. not from an alphabet, which is a non-empty and *finite* set of symbols. Since Υ is an 'open-ended' set, we chose the name "open regular language". In Section 7, we show that the set of regular languages is a proper subset of the set of ORLs, and the set of ORLs is not a subset of the set of context-free languages [18].

"Strand" and "string" are different but related terms: First of all, both a string and a strand are finite sequences of symbols. A string consists of symbols from an alphabet. Since any alphabet is a subset of Υ, a string is a strand. Since a strand w consists of finite number of symbols, the set Σ of symbols in w is also finite. Therefore, Σ is an alphabet, and w can be interpreted as a string that consists of symbols from Σ. The empty string can be interpreted as the empty strand, and vice versa.

We conclude this section by revisiting the key difference between a DFA and a DAR: a DFA with an alphabet Σ either accepts or rejects a finite sequence of symbols, provided that the sequence is an element of Σ^*; whereas a DAR either accepts or rejects a finite sequence of symbols, provided that the sequence is an element of Υ^*. Since Υ^* is the set of all possible finite sequences of symbols, a DAR either accepts or rejects *any* finite sequence of symbols.

5.2. Translating Vibes diagrams to DARs

In this section, we provide the semantics of Vibes, by presenting a function that takes a Vibes diagram as the

[5] In [9,10] Vibes is referred to as "VisuaL", and ⋆ is referred to as "$".

[6] Note that "strand" and "string" (see Section 4) are different terms. Despite being different terms, "strand" and "string" are still related. We will explain the relation in this section.

Fig. 5. The states of M^e, after step 2.

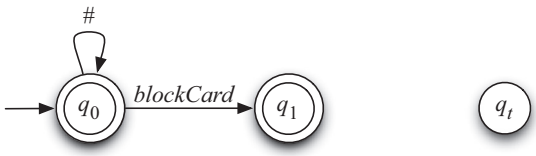

Fig. 6. The states and some of the transitions of M^e, after step 4.

input, and outputs the DAR corresponding to the Vibes diagram.

We formally define the semantics of Vibes by a total function $getDARof : V \rightarrow D$, where V is the set of Vibes diagrams, and D is the set of DARs. Let S be a Vibes diagram, and M be a DAR, such that $getDARof(S) = M$. In the remainder of this section, we step-by-step explain how *getDARof* constructs M, based on S. The explanation of each step is structured as follows: first, we formally explain the step in general terms using S and M; next we provide an example execution of the step using an example Vibes diagram S^e and the corresponding DAR M^e.

5.2.1. Step 1: the initialization of M

At this step, *getDARof* initializes $M = \langle Q, \Sigma_a, \delta, q_0, F \rangle$, such that

- $Q = \Omega \cup \{q_t\}$ and $\Omega = \{q_0\}$,
- $\Sigma_a = \Sigma_b \cup \{\#\}$ and $\Sigma_b = \varnothing$,
- δ is not defined yet, and
- $F = \varnothing$.

Now, let us see an example. Let S^e denote the Vibes diagram shown in Fig. 3.[7] If S^e is given to *getDARof* as the input, then *getDARof* initializes a DAR $M^e = \langle Q^e, \Sigma_a^e, \delta^e, q_0, F^e \rangle$, such that

- $Q^e = \Omega^e \cup \{q_t\}$ and $\Omega^e = \{q_0\}$,
- $\Sigma_a^e = \Sigma_b^e \cup \{\#\}$ and $\Sigma_b^e = \varnothing$,
- δ^e is not defined yet, and
- $F^e = \varnothing$.

5.2.2. Step 2: adding the user-defined states

At this step, *getDARof* adds new states to Ω and F, as follows: Let $n_0, n_1, \ldots, n_m$ be the nodes of S, such that n_0 is either the initial or the initial-final node. Given $n_0, n_1, \ldots, n_m$, *getDARof* performs the following steps:

1. Define new states $q_1, q_2, \ldots, q_m$, and add them to Ω.
2. For each n_i where $0 \le i \le m$, map n_i to q_i. We denote this mapping with the total function $getStateOf : N \rightarrow \Omega$, where N is the set of nodes of S.
3. If n_0 is the initial-final node, then add q_0 to F.
4. For each final node n_f of S, add $getStateOf(n_f)$ to F.

Now, let us revisit the example. The nodes of S^e (see Fig. 3) are q0 and q1, where q0 is the initial-final node.[8] Accordingly, *getDARof*

1. Defines a new state q_1, and adds it to Ω^e.
2. Maps q0 to q_0, and q1 to q_1.
3. Adds q_0 to F^e.
4. Adds q_1 to F^e.

Consequently, Ω^e becomes $\{q_0, q_1\}$, F^e becomes $\{q_0, q_1\}$, and Q^e becomes $\{q_0, q_1, q_t\}$. In Fig. 5, the states of M^e are depicted.

The initial state is depicted as the circle that is the target of the only arrow without any source, each non-final state is depicted as a single circle, and each final state is depicted as a double circle.

5.2.3. Step 3: adding the symbols

At this step, *getDARof* adds new symbols to Σ_b, as follows: Let LBL be the set of the labels of the edges of S. For each $lbl \in (LBL \backslash \{\star\})$, *getDARof* defines a new symbol a, maps lbl to a, and adds a to Σ_b. The mapping between the labels and the symbols is denoted by the total function $getSymbolOf : (LBL \backslash \{\star\}) \rightarrow \Sigma_b$.

Now, let us revisit the example. The set of the labels of the edges of S^e (see Fig. 3) is {blockCard, $\star$}. Accordingly, *getDARof* defines a new symbol, say, *blockCard*. Subsequently, *getDARof* maps blockCard to *blockCard*. Finally, *getDARof* adds *blockCard* to Σ_b^e. Thus, Σ_b^e becomes $\{blockCard\}$, and Σ_a^e becomes $\{blockCard, \#\}$.

5.2.4. Step 4: partially defining the transition function

At this step, *getDARof* partially defines δ, as follows: for each edge e (of S) with the source node sn, target node tn, and label lbl, *getDARof* does the following: If $lbl = \star$, then *getDARof* defines that $\delta(getStateOf(sn), \#) = getStateOf\ (tn)$. If $lbl \neq \star$, then *getDARof* defines that $\delta(getStateOf(sn), getSymbolOf(lbl)) = getStateOf(tn)$.

Now, let us revisit the example. In S^e (see Fig. 3), there is a $\star$−labelled edge from q0 to q0; thus, *getDARof* defines that $\delta^e(q_0, \#) = q_0$. There is a blockCard-labelled edge from q0 to q1; thus *getDARof* defines that $\delta^e(q_0, blockCard) = q_1$. These transitions are depicted as labelled arrows in Fig. 6.

5.2.5. Step 5: defining the remaining transitions with

At this step, *getDARof* defines the remaining transitions performed by M with the symbol #, as follows: for each state $q \in Q$, if $\delta(q, \#)$ is not defined yet, then *getDARof* defines that $\delta(q, \#) = q_t$.

Now let us revisit the example. The transitions $\delta^e(q_1, \#)$ and $\delta^e(q_t, \#)$ are not defined yet (see Fig. 6). Therefore,

[7] We use the superscript ^{e} for distinguishing the example diagram from the general diagram S, which is introduced earlier in Section 5.2.

[8] Note that we intentionally format the nodes of the Vibes diagram S^e as they appear in Fig. 3 (e.g., q0), which is a formatting different from the formatting of the states of the DAR M^e (e.g., q_0). In this way, we distinguish between a node of S^e and the corresponding state of M^e in this paper.

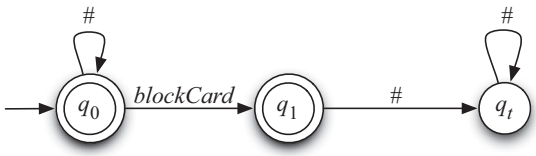

Fig. 7. The states and some of the transitions of M^e, after step 5.

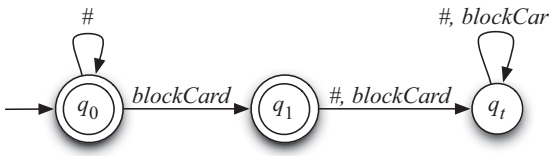

Fig. 8. The transition graph of M^e, after step 6.

getDARof defines that $\delta^e(q_1, \#) = q_t$ and $\delta^e(q_t, \#) = q_t$, as shown in Fig. 7.

5.2.6. Step 6: defining the remaining transitions

At this step, *getDARof* defines the remaining transitions of M, as follows: for each state $q \in Q$, and for each symbol $a \in \Sigma_b$, if $\delta(q, a)$ is not defined yet, then *getDARof* defines that $\delta(q, a) = \delta(q, \#)$.

Now, let us revisit the example. The transitions $\delta^e(q_1, blockCard)$ and $\delta^e(q_t, blockCard)$ are not defined yet (see Fig. 7). Therefore, *getDARof* defines that $\delta^e(q_1, blockCard) = \delta^e(q_1, \#) = q_t$ and $\delta^e(q_t, blockCard) = \delta^e(q_t, \#) = q_t$, as visible in Fig. 8, which is the transition graph[9] of M^e.

Upon the completion of this step, the construction of M and M^e are also completed. This six-step construction (i.e. $getDARof : V \to D$) defines the semantics of Vibes. The asymptotic time complexity of *getDARof* is $O(|Q| \times |\Sigma_b|)$, which is determined by the sixth step.

5.3. The benefit of using DARs instead of DFA

Note that the function *getDARof* does not depend on a model of the underlying system, while translating a Vibes diagram to a DAR. Therefore, the resulting DAR does not contain any additional transitions due to the action names that appear in the model of the underlying system, but not in the Vibes diagram. Consequently, the algorithms that need to exhaustively visit each transition do not suffer from the storage and performance bottlenecks mentioned at the beginning of Section 5. This is the benefit of using DARs instead of DFA to provide the formal semantics of Vibes.

For example, the DFA whose state-transition diagram is shown in Fig. 2, and the DAR whose transition graph is shown in Fig. 8 can be seen as alternative formal representations of the Vibes diagram shown in Fig. 3. However, the DAR has six transitions as visible in Fig. 8, whereas the DFA has 33 transitions as visible in Fig. 2.

[9] The *transition graph* of a DAR is a graph where nodes represent the states, and edges represent the transitions of the DAR.

6. The expressive power of Vibes

In Section 5, we have seen that each Vibes diagram represents a DAR; hence expresses an ORL, which is defined in Section 5.1. This entails that the set of ORLs is the 'upper bound' on the expressive power of Vibes. In this section, our goal is to find the 'lower bound', so that we can determine the exact expressive power of Vibes. In particular, we intend to answer the following question in this section: for any ORL L, is there a Vibes diagram that can express L? To answer this question, we state the following theorem:

Theorem 1. *For any DAR M, there is a Vibes diagram S such that $L(getDARof(S)) = L(M)$.*

Proof. For any DAR $M = \langle Q = \Omega \cup \{q_t\}, \Sigma_a = \Sigma_b \cup \{\#\}, \delta, q_0, F\rangle$, it is possible to construct a Vibes diagram S such that $L(getDARof(S)) = L(M)$. This construction is denoted by the total function $getVibesOf : D \to V$, where D is the set of DARs, and V is the set of Vibes diagrams. If M is given to *getVibesOf* as the input, then *getVibesOf* performs the following steps to construct S:

6.1. Step 1: creating the nodes

At this step, *getVibesOf* creates the nodes of S, as follows: for each state $q \in Q$, *getVibesOf* creates a distinct node n, such that

- q is mapped to n.
- If $q = q_0$ and $q \in F$, then n is the initial-final node.
- If $q = q_0$ and $q \notin F$, then n is the initial node.
- If $q \neq q_0$ and $q \in F$, then n is a final node.
- If $q \neq q_0$ and $q \notin F$, then n is a plain node.

We denote the mapping from the states of M to the nodes of S using the total function $getNodeOf : Q \to N$, where N is the set of nodes of S.

Now, let us see an example of this step. In Section 5.2, we constructed the DAR M^e, whose transition graph is shown in Fig. 8. As visible in this figure, the states of M^e are q_0, q_1, and q_t. q_0 is the initial and a final state, and q_1 is a final state. Accordingly, *getVibesOf* creates the nodes, say, q0, q1, and qt of S^e, such that

- $getNodeOf(q_0) =$ q0, where q0 is the initial-final node.
- $getNodeOf(q_1) =$ q1, where q1 is a final node.
- $getNodeOf(q_t) =$ qt, where qt is a plain node.

Fig. 9 shows the Vibes diagram S^e, upon the creation of its nodes.

6.2. Step 2: creating the edges

At this step, for each transition $\delta(q_i, a_j) = q_k$ of M, *getVibesOf* creates an edge e of S, such that the source of e is $getNodeOf(q_i)$, and the target of e is $getNodeOf(q_k)$. If $a_j =$, then the label of e is $\star$, else the label of e is the symbol denoted by a_j.

Fig. 9. The Vibes diagram S^e, after step 1.

Now, let us revisit the example. Fig. 10 shows the Vibes diagram S^e, upon the creation of the edges according to the transitions of M^e (see Fig. 8).

Upon the completion of this step, the construction of S and S^e are also completed. $L(getDARof(S)) = L(M)$, because the only difference between $getDARof(S)$ and M is as follows: $getDARof(S)$ has one extra state, say, q_t^{new}, which is the *unreachable* default trap state of $getDARof(S)$. Since this is the only difference between $getDARof(S)$ and M, we can conclude that $L(getDARof(S)) = L(M)$.

Now, let us revisit the example. Fig. 11 shows the transition graph of $getDARof(S^e)$, where S^e is shown in Fig. 10.

By comparing Figs. 8 and 11, one can notice that $L(getDARof(S^e)) = L(M^e)$. □

Based on Theorem 1 and the semantics of Vibes (Section 5.2), we can conclude as follows: Using Vibes, one can express any ORL and nothing else. Hence, the Vibes language and the DAR formalism have the same expressive power.

In [9], interested readers can find additional theoretical results, such as the closure properties of ORLs, and an algorithm that minimizes the number of nodes and edges of a Vibes diagram without altering the ORL expressed by the diagram. For example, if the Vibes diagram shown in Fig. 10 is given as the input to this algorithm, then the algorithm outputs the Vibes diagram as shown in Fig. 3.

7. Open regular languages vs. other language families

In Section 5.1, we have already defined the Open Regular Languages (ORLs). In this section, we compare ORLs with the Regular Languages (RLs) [18] and Context-Free Languages (CFLs) [18].

7.1. Open regular languages versus regular languages

In this section, we compare ORLs with RLs. In particular, we answer the following two questions: Is any RL also an ORL? Is any ORL also an RL? To answer the first question, we use the following theorem:

Theorem 2. *For any given RL L, there is a DAR M^{dar}, such that $L(M^{dar}) = L$.*

Proof. Let L be an arbitrary RL. By the definition of RL [18], there is a DFA $M^{dfa} = \langle Q^{dfa}, \Sigma^{dfa}, \delta^{dfa}, q_0^{dfa}, F^{dfa}\rangle$, such that $L(M^{dfa}) = L$. Based on M^{dfa}, we can step-by-step construct a DAR $M^{dar} = \langle Q^{dar} = Q^{dfa} \cup \{q_t^{dar}\}, \Sigma_a = \Sigma^{dfa} \cup \{\#\ dar\}, \delta^{dar}, q_0^{dfa}, F^{dfa}, \Xi, \eta\rangle$, such that $L(M^{dar}) = L(M^{dfa}) = L$. The steps of this construction are as follows:

1. For each state $q \in Q^{dfa}$ and for each symbol $a \in \Sigma^{dfa}$, define $\delta^{dar}(q,a) = \delta^{dfa}(q,a)$.
2. For each symbol $a \in \Sigma^{dfa}$, define $\delta^{dar}(q_t^{dar}, a) = q_t^{dar}$.

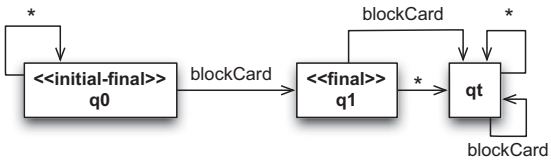

Fig. 10. The Vibes diagram S^e, after step 2.

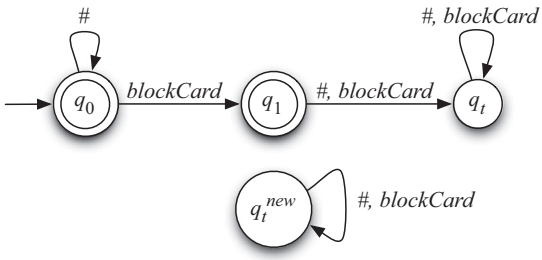

Fig. 11. The transition graph of $getDARof(S^e)$, after step 2.

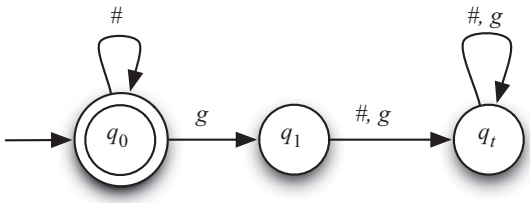

Fig. 12. The transition graph of a DAR.

3. For each $q \in Q^{dar}$, define $\delta^{dar}(q, \#^{dar}) = q_t^{dar}$.
4. Define Ξ and η arbitrarily. □

Based on Theorem 2 and the definition of ORL (Section 5.1), we conclude that any RL is also an ORL. Consequently, the answer to the first question stated at the beginning of this section is "yes". This also entails that DFAs are not more expressive than DARs. To answer the second question, we use the following theorem:

Theorem 3. *For any given ORL L, it is not guaranteed that there is a DFA M^{dfa}, such that $L(M^{dfa}) = L$.*

Proof. For any given ORL L, assume that there is a DFA M^{dfa}, such that $L(M^{dfa}) = L$ (i.e. assume that Theorem 3 is false). Let M^{dar} denote a DAR whose transition graph is shown in Fig. 12. Observe that $L(M^{dar}) = (\Upsilon \setminus \{g\})^*$, which is the set of all possible strands that do not contain the symbol g. By the definition of ORL (Section 5.1), $(\Upsilon \setminus \{g\})^*$ is an ORL. Based on the assumption above, there is a DFA M^{dfa} such that $L(M^{dfa}) = (\Upsilon \setminus \{g\})^*$. Hence, the input alphabet of M^{dfa} is $\Upsilon \setminus \{g\}$. Since the set $\Upsilon \setminus \{g\}$ is infinite, this set is not an alphabet [18]. Note that the previous two sentences contradict with each other. □

Based on Theorem 3 and the definition of RL (Section 5.1), we can conclude that a given ORL is not necessarily an RL. Consequently, the answer to the second question stated at the beginning of this section is "no". This also

entails that DARs have different expressive power than DFAs.

Since (a) any RL is also an ORL, and (b) a given ORL is not necessarily an RL, we conclude that the set *RLs* of regular languages is a proper subset of the set *ORLs* of open regular languages. Since (a) DFAs are not more expressive than DARs, and (b) DARs have different expressive power than DFAs, we conclude that DARs are more expressive than DFAs.

7.2. Open regular languages versus context-free languages

A given set L of strings is a Context-Free Language (CFL), if and only if there is a Non-deterministic Pushdown Automaton (NPDA) that exclusively accepts the strings in L [18]. The set of RLs is a proper subset of the set of CFLs [18]. Since the set of RLs is a proper subset of both the set of CFLs and the set of ORLs, the relation between ORLs and CFLs is an interesting topic to investigate. In this section, we study this relation. In particular, we answer the following two questions: Is any ORL also a CFL? Is any CFL also an ORL?.

In the Venn diagram depicted in Fig. 13, *CFLs* denotes the set of context-free languages, and each number denotes the distinct set represented by the closed region where the number is placed.

By discovering whether each of these four sets is empty or not, we can understand the relation between CFLs and ORLs.

Set 3 contains RLs, so it is non-empty. Since an RL is both a CFL and an ORL, $CFLs \cap ORLs \neq \varnothing$.

Let $a^n b^n$ denote the set of strings, where a string consists of n number of a's followed by n number of b's, such that $n \geq 0$. For example, *aaabbb* is in $a^n b^n$, but *abb* not. $a^n b^n$ is known to be a CFL that is not an RL [18]. Now, the question is, whether $a^n b^n$ is in Set 2 or 4 (see Fig. 13).

$a^n b^n$ is not an RL, due to the following facts: any DFA has a finite memory (i.e. a finite set of states), thus for a sufficiently large value of n, a DFA cannot 'remember' how many a's it encountered. Therefore, the DFA cannot 'know' how many b's the string should have. This means that it is impossible to construct a DFA that exclusively accepts the strings in $a^n b^n$. Since a set of strings is an RL if an only if there is a DFA that exclusively accepts these strings, $a^n b^n$ is not an RL.

Similar to a DFA, a DAR also has a finite memory (i.e. a finite set of states). Hence, it is not possible to construct a DAR that exclusively accepts the strings in $a^n b^n$. As a result, $a^n b^n$ is in Set 4 (Fig. 13), so Set 4 is non-empty.

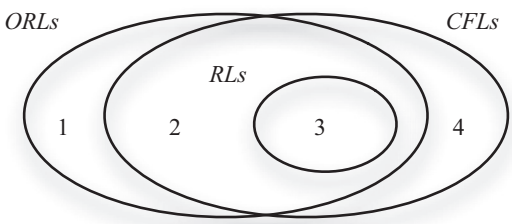

Fig. 13. Each number in this Venn diagram denotes a distinct set represented by the region where the number is placed.

In Section 7.1, we have shown that $(\Upsilon\backslash\{g\})^*$ is an ORL but not an RL. $(\Upsilon\backslash\{g\})^*$ is not an RL, because the sequences in an RL (i.e. the sequences accepted by a DFA) consist of symbols from a *finite* set of symbols, whereas the sequences in $(\Upsilon\backslash\{g\})^*$ consist of symbols from $\Upsilon\backslash\{g\}$, which is *infinite*. Since, the sequences in a CFL, i.e., the sequences accepted by a non-deterministic push-down automata (NPDA) [18], also consist of symbols from a *finite* set of symbols, we can conclude that $(\Upsilon\backslash\{g\})^*$ is not a CFL, either. Consequently, $(\Upsilon\backslash\{g\})^*$ is an element of Set 1, that is, Set 1 is non-empty.

Let us assume that Set 2 is non-empty. Hence there is at least one language L that is both an ORL and CFL but not an RL. Accordingly, there is a DAR M_{dar} and a NPDA M_{npda} such that $L(M_{dar}) = L(M_{npda}) = L$, and there is no DFA M_{dfa} such that $L(M_{dfa}) = L$. This means M_{dar} has infinite number of states and M_{npda} has an infinite alphabet, which are contradictions. Therefore, we can conclude that Set 2 is empty.

Based on the discussion so far in this section, we conclude the following:

- Since Set 1 is non-empty, a given ORL is not necessarily a CFL. Hence, the answer to the first question stated at the beginning of this section is "no".
- Since Set 4 is non-empty, a given CFL is not necessarily an ORL. Hence, the answer to the second question stated at the beginning of this section is "no".
- Based on (a) the definition of ORL (Section 5.1), (b) the definition of CFL (see the beginning of this section), and (c) the fact that Sets 1 and 4 are non-empty, we conclude that DARs and NPDAs have different expressive power; i.e. NPDAs are not more expressive than DARs, and vice versa.
- Since Set 2 is empty, $ORLs \cap CFLs = RLs$.

Using the theoretical results presented in this section as a starting point, we think that one can construct a hierarchy of open formal languages (open regular languages, open context-free languages, open context-sensitive languages, etc.) analogous to the Chomsky hierarchy, where the intersection of these two hierarchies is the set of regular languages.

8. Empirical results

In Section 2.3, we mentioned that a part of the goal of this research is to define a language that is easy to use. In this section, we provide empirical results suggesting that Vibes is indeed an easy-to-use language. In Section 8.1, we discuss the industrial Vibes diagrams created by a professional software engineer. In Section 8.2, we present the results of the controlled experiments that we conducted using the industrial Vibes diagrams. Based on these empirical results, we discuss why Vibes can be considered as an easy-to-use language, in Section 8.3.

8.1. Industrial Vibes diagrams

After we developed the Vibes language, we wanted to gain an initial understanding of whether a professional software engineer can efficiently and effectively use Vibes in an industrial setting. Therefore, we conducted a preliminary case study. In this section, we first explain the industrial setting in which this case study was conducted, and then present the results of the case study.

We conducted the case study at ASML (www.asml.com) which is a supplier of lithography systems for the semiconductor industry. In a nutshell, ASML develops wafer scanners, which are mechatronic systems that print integrated circuit designs onto silicon chips. Software of an ASML wafer scanner contains approximately 15 million lines of source code written in the C programming language [16] and is decomposed into approximately 200 components that are continuously maintained and extended by more than 600 software engineers on a daily basis.

We conducted the case study within the context of a mid-sized component that has 55,000 lines of source code. The domain expert of this component is a professional software engineer who has 15 years of industrial experience.

Before the case study, we trained the domain expert for 1 h, so that he can create Vibes diagrams. Subsequently, the expert created three Vibes diagrams each of which expresses a behavioral requirement on a distinct function of the component. The expert selected the requirements and the functions based on the existing daily work that he had to carry out at that time.

In total, the expert worked 8 h under daily conditions: he was interrupted by colleagues, phone calls, lunch and coffee breaks, etc. Using two video cameras, we captured the expert, his screen, and his desk while he was working. The resulting footage enabled us to accurately calculate the net amount of time he spent for creating the diagrams, which was 155 min.

The first diagram created by the expert contains 11 nodes and 19 edges. To create this diagram, the expert spent 80 min in total. In Table 1, the data for each of the three diagrams are listed.

Using the data presented in Table 1, one can calculate that the expert spent on the average 160, 83, and 56 s per node or edge while creating D1, D2, and D3, respectively. This calculation indicates that the expert quickly gained speed in creating diagrams. To be able to generalize this conclusion to other engineers however, we need to repeat this study with more software developers.

To create the specifications, the expert had to rigorously analyze the relationship between the implementation, the detailed design, the architecture, and the requirements of the software component. This rigorous analysis enabled him to find one defect, which had to be repaired in the next release, four design anomalies that required restructuring and maintenance, and one undocumented feature. Two weeks earlier, the component in which the expert found these problems had been maintained by himself, and reviewed by two of his colleagues.

Table 1
The size and cyclomatic complexity of the Vibes diagrams, and the time that the domain expert spent for creating the diagrams.

Diag.	*# Nodes*	*# Edges*	*Complexity*	*Time (min.)*
D1	11	19	10	80
D2	11	23	14	47
D3	10	20	12	28

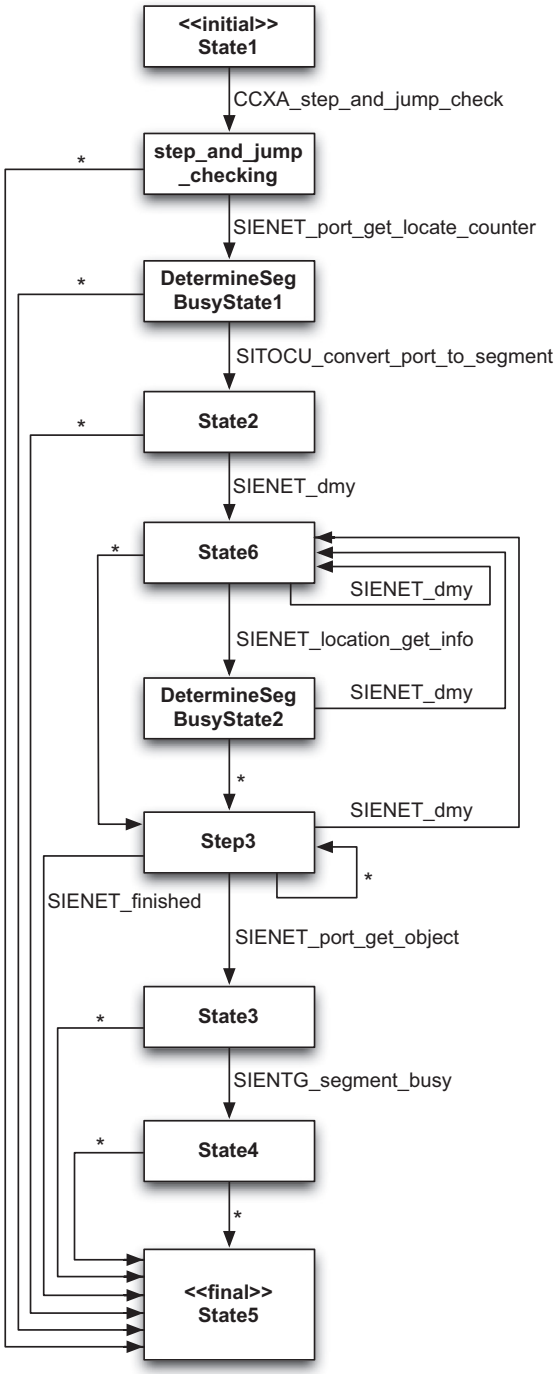

Fig. 14. The industrial Vibes diagram D3.

To provide an understanding of how an industrial Vibes diagram looks like, we present D3 in Fig. 14.[10]

This Vibes diagram represents a requirement on a function whose state model has 27 distinct action names. The diagram however contains eight action names, which are shown as edge labels in Fig. 14.

8.2. Experimental results

In [10], we report on two controlled experiments that we conducted with 44 participants (21 M.Sc. computer science students and 23 professional software engineers). In these experiments, the participants were asked to use the industrial Vibes diagrams, which we discussed in Section 8.1, to repair realistic defects that we injected into the corresponding functions from ASML's software. The goal of this experiment was to evaluate our tools, which can automatically find and report such defects. Accordingly, we formulated the following hypotheses:

- H_0^1: The tools do not have any effect on the amount of time spent by the participants.
- H_0^2: The tools do not have any effect on the number of defects that the participants leave not-repaired.

Based on the results of the statistical tests we performed, we successfully rejected both H_0^1 and H_0^2, in both the student and developer experiments.

In the student experiment, the tools reduced the time spent by an average student by 50%. In addition, the tools prevented approximately one defect per 100 lines of source code in this experiment.

In the developer experiment, the tools reduced the time spent by an average developer by 75%. In addition, the tools prevented approximately one defect per 140 lines of source code in this experiment.

8.3. An evaluation of Vibes from a usability perspective

According to an ISO standard [2], the usability of a software product can be investigated in terms of three sub-characteristics:

Understandability Attributes of software that bear in the users' effort for recognizing the logical concept and its applicability [2].

Learnability Attributes of software that bear on the users' effort for learning its application [2].

Operability Attributes of software that bear on the users' effort for operation and operation control [2].

In this section, we evaluate the usability of Vibes by discussing its understandability, learnability, and operability, based on the empirical results presented in Sections 8.1 and 8.2.

[10] The identifiers in Fig. 14 are obfuscated, due to our non-disclosure agreement with ASML.

8.3.1. Understandability of Vibes

Vibes has been developed as a part of a solution to an industrial problem presented in [10]. This problem was reported to us and experienced by some of the professional software engineers at ASML. The logical concept and application of Vibes was easy to recognize for the engineers at ASML, as confirmed by the empirical results presented in Sections 8.1 and 8.2. Thus, we can conclude that Vibes is an easy-to-understand language for the software engineers at ASML. Nonetheless, additional research is needed to find out whether this conclusion can be generalized to other people.

8.3.2. Learnability of Vibes

The professional software engineer mentioned in Section 8.1 could create Vibes diagrams after a 1 h training. In addition, the 44 participants of the experiments mentioned in Section 8.2 were able to work with the Vibes diagrams, upon a 15-min training. Accordingly, we can conclude that Vibes is an easy-to-learn language for the professional software engineers and M.Sc. computer science students whose profiles are explained in [10]. To find out whether Vibes is easy to learn for a different group of people requires additional research.

8.3.3. Operability of Vibes

The results of the controlled experiments mentioned in Section 8.2 indicate that tool-support plays a key role in the operability of Vibes diagrams. If there is tool support that can automatically verify various artifacts such as design models or source code, then Vibes can be considered as an operable language for professional software engineers and M.Sc. computer science students whose profiles are explained in [10].

Based on the discussion so far in Section 8.3, we can conclude that Vibes is an easy-to-use language, at least for the specific group of people and within the specific context explained throughout Section 8.

9. Related work

9.1. Temporal logic versus Vibes

Temporal logic languages such as LTL [6], CTL [6], CTL* [6], FLTL [7,17] are textual languages that are intended for expressing the temporal or logical properties of non-terminating systems; whereas Vibes is a visual language that is intended for expressing temporal or logical properties of terminating systems. Given a dedicated termination construct built into a temporal-logic-based model checker tool, one can still use temporal logic languages to express all possible properties that can be expressed by Vibes. However, Vibes would arguably be easier to use while specifying such properties.

The semantics of a temporal logic formula is typically defined with respect to a model of the underlying system [6], whereas the semantics of a Vibes diagram is defined independent of any underlying system, as explained in Section 5.2.

9.2. Wildcards in automata-based testing and verification

The idea of using wildcard transitions in automata is not entirely new. For example, the * transitions [7] facilitate partial order reduction for the properties that are closed under stuttering. The "*"-transitions [14], and the *other*-transitions [5] seem to be similar to context-sensitive wildcards. However, the semantics of these transitions are not formally defined in the respective literature.

The classical book by Sippu and Soisalon-Soininen [22] also presents wildcard transitions for the definition of lexical analyzers, and any reasonable formalization of the notion of pattern would of course provide a similar construct to deal with variables. Building further upon the familiar notion of wildcard however, this paper reveals not only theoretical but also practical implications of using context-sensitive wildcards in visual specifications of software behavior, and introduces a promising visual language that can easily be used by people in order to specify behavioral requirements of algorithms.

9.3. Closure properties of open regular languages

In [9], we introduced open-regular languages (ORLs) as a new family of formal languages, and investigated some of their closure properties. Consequently, we showed the following:

- For any given ORL L, $\overline{L}$ is also an ORL.
- For any two ORLs L_1 and L_2, $L_1 \cup L_2$ is also an ORL.
- For any two ORLs L_1 and L_2, $L_1 \cap L_2$ is also an ORL.
- For any two ORLs L_1 and L_2, the set of strands obtained by concatenating a strand from L_1 with a strand from L_2 is also an ORL.
- For any ORL L, the set of strands obtained by concatenating zero or more strands from L is also an ORL.

Based on these closure properties, we defined operators for composing new Vibes diagrams from existing ones [9].

9.4. Automata for strings over infinite sets of symbols

In Section 5.1, we have explained the key difference of DARs from DFAs: DARs can accept or reject finite sequences of symbols from the universal set of symbols, which is an infinite set. Kaminski and Francez [15], Globerman and Harel [8], Milo et al. [19], and Neven et al. [20] have also studied the recognition of finite sequences of symbols from an infinite set of symbols, as we discuss in the remainder of this section.

9.4.1. Register automata

Kaminski and Francez [15] introduced *register automata*: a register automaton is a finite-state machine equipped with a finite number of registers. Each register can store a symbol. While processing the input tape, a register automaton compares the current symbol of the tape with the symbols in the registers. Based on the current state and the result of the comparison, the automaton decides (a) the next state, (b) whether to store the current symbol (of the tape) in a register (by overwriting the existing symbol in the register), and (c) whether to move to the left or the right position on the tape, or to stay at the current position of the tape. The symbols on the input tape of a register automaton can come from an infinite set of symbols.

A DAR has a built-in wildcard symbol (i.e., the # symbol), which matches an infinite number of symbols. Whereas a register automaton has finite number of registers each of which contains exactly one 'regular' (i.e., non-wildcard) symbol. Therefore, the symbol in a register of a register automaton cannot match infinite number of symbols. As a result, a register automaton cannot 'simulate' the #−transitions of a DAR.

A register automaton can modify the contents of the registers, while processing the input tape. Whereas, a DAR cannot modify the contents of the abstract input alphabet. These differences between a DAR and a register automaton suggest that DARs and register automata possibly have different expressive power.

9.4.2. Pebble automata

According to Milo et al. [19], a *pebble automaton* is a finite-state machine equipped with a finite number of consecutively numbered pebbles. A pebble can be placed on a position of the input tape, so that the symbol at that position is marked by the automaton. While processing the input tape, a pebble automaton can drop down the pebbles or pick them up. This is regulated according to the stack convention: the ith pebble can be dropped down only if the $(i-1)$th pebble is already on the tape, and the ith pebble can be picked up only if the $(i+1)$th pebble is already picked up. While processing the input tape, a pebble automaton compares the current symbol of the input tape with the symbol marked by the most recently dropped pebble. Based on the current state and the result of the comparison, the automaton decides (a) the next state, (b) whether to drop down or pick up a pebble, and (c) whether to move to the left or to the right position on the tape, or to stay at the current position of the tape. The symbols on the input tape of a pebble automaton can be from an infinite set of symbols.

A pebble automaton has a finite number of pebbles each of which can be used for marking exactly one 'regular' (i.e. non-wildcard) symbol. Therefore, the symbols marked by the pebbles of a pebble automaton cannot match infinite number of symbols. As a result, a pebble automaton cannot 'simulate' the #−transitions of a DAR.

By dropping down and picking up the pebbles, a pebble automaton can modify the set of marked symbols. Whereas, a DAR cannot modify the contents of the abstract input alphabet. These differences between a DAR and a pebble automaton suggest that DARs and pebble automata possibly have different expressive power.

Neven et al. [20] have investigated the expressive power of different types of register and pebble automaton, and compared them with first order logic and monadic second-order logic. They investigated variations of register and pebble automaton: one way or two way tape readers;

and deterministic, non-deterministic, or alternating (i.e. hybrid) versions.

10. Conclusions

The commonly used graphical languages such as statecharts support hierarchies (i.e. nested structures), so that one can define different levels of abstraction in behavioral specifications. In this paper, we presented an additional mechanism for abstraction, which we call Context-Sensitive Wildcard (CSW). We defined CSW as the key feature of Vibes, which is a simple visual language for expressing the logical and temporal properties of possible executions of an algorithm. We presented the syntax, formal semantics, and expressive power of Vibes, which reveals the theoretical and practical implications of using CSWs, in the visual specifications of software behavior.

A Vibes diagram represents an automaton called Deterministic Abstract Recognizer (DAR), which is a variant of a Deterministic Finite Accepter (DFA) [18]. The key difference between DFA and DAR is as follows: a DFA with an alphabet Σ either accepts or rejects a finite sequence of symbols, provided that the symbols are from Σ, whereas a DAR either accepts or rejects *any* finite sequence of symbols. The usage of DAR instead of DFA avoids the storage and performance bottlenecks for the algorithms that need to visit each transition.

DARs express a new family of formal languages called Open Regular Languages (ORLs) [9]. Using Vibes, one can express any ORL and nothing else; thus the Vibes language and the DAR formalism have the same expressive power. Based on these findings it is possible to extend the existing visual languages with CSW, which is an abstraction mechanism to create highly evolvable specifications of software behavior.

We conducted an initial pilot study where a professional software engineer created three real-life Vibes diagrams. Using these diagrams and the corresponding source code, we conducted formal experiments with 23 professional software engineers, and 21 M.Sc. computer science students. Based on our observations from these empirical studies, we evaluated the usability of Vibes in terms of its understandability, learnability, and operability. This evaluation suggests that vibes is an easy-to-use language.

10.1. Future work

10.1.1. A formal comparison of DARs with register and pebble automata

In Section 9.4, we discussed the similarities and differences between DARs and register and pebble automata. In the future we intend to conduct a rigorous and formal comparison, so that we can find out the relative expressive power of Vibes with respect to the expressive power of register and pebble automata.

10.1.2. Extending the Vibes language for expressing the logical or temporal properties of non-terminating systems

The current version of Vibes, as it is defined in this paper, is intended for specifying behavioral requirements of terminating systems. Considering the current trend towards always-on connectivity and ubiquitous interaction however, an interesting direction for future research would be to extend Vibes for specifying behavioral requirements of non-terminating systems, which typically interact with their environment over a prolonged period of time, in a collaborative context, and across different users. In this section, we outline what we think is a solid basis for such an extension to Vibes.

In Section 9.1, we mentioned some of the existing formalisms for expressing the logical or temporal properties of non-terminating systems. Yet another formalism for expressing such properties is Büchi automata [6,3]. There are three differences between a Büchi automaton and a DFA:

1. A Büchi automaton with an input alphabet Σ either accepts or rejects any *infinite* sequence of symbols from Σ, whereas a DFA with an input alphabet Σ either accepts or rejects any *finite* sequence of symbols from Σ.
2. A Büchi automaton can perform transitions non-deterministically, whereas a DFA performs transitions deterministically.
3. A Büchi automaton may have multiple initial states, whereas a DFA has exactly one initial state.

The set of infinite sequences of symbols accepted by a Büchi automaton is called ω-regular language [6].

Using the contents of Section 5 as a basis, one can define a new type of automata, say *Non-deterministic Abstract Infinite-sequence Recognizer* (*NAIR*), such that the differences between a NAIR and a DAR is the same as the differences between a Büchi automaton and a DFA:

1. A NAIR either accepts or rejects any *infinite* sequence of symbols, whereas a DAR either accepts or rejects any *finite* sequence of symbols.
2. A NAIR can perform transitions non-deterministically, whereas a DAR performs transitions deterministically.
3. A NAIR has multiple initial states, whereas a DAR has exactly one initial state.

Since a NAIR would accept or reject any infinite sequence of symbols, it could be used for expressing logical or temporal properties of non-terminating systems. Accordingly, Vibes could be extended, such that each Vibes diagram represents a NAIR, and each NAIR can be represented by a Vibes diagram. Hence, the extended Vibes would be naturally suited not only for reasoning about non-determinism, but also for expressing the logical and temporal properties of non-terminating systems.

10.1.3. Augmenting the existing visual languages with context-sensitive wildcards

There are several visual languages for expressing the behavioral designs of software systems: statecharts [12], activity diagrams [1], sequence diagrams [1], collaboration diagrams [1], etc. Wieringa [24] surveys such visual languages. Augmenting these languages with context-sensitive

wildcards may be an interesting research direction for the future. For example, if concurrency were a key issue in modeling behavior, one could augment activity diagrams with context-sensitive wildcards in order to benefit from the advantages of Vibes, and at the same time to have explicit constructs for representing concurrency and synchronization. Based on a few examples of such diagrams, one can conduct empirical studies to find out if the combination of context-sensitive wildcards with concurrency and synchronization concepts has any impact on the usability of augmented activity diagrams.

References

[1] UML, ⟨http://www.uml.org/ ⟩.
[2] ISO/IEC 9126-1:1991, Software Engineering – Product Quality.
[3] Bowen Alpern, Fred B. Schneider, Verifying temporal properties without temporal logic, ACM Transactions on Programming Languages and Systems 11 (1) (1989) 147–167.
[4] ASML, ⟨http://www.asml.com ⟩.
[5] Hao Chen, David Wagner, Mops: an infrastructure for examining security properties of software, in: CCS '02: Proceedings of the 9th ACM Conference on Computer and Communications Security, New York, NY, USA, 2002, ACM, pp. 235–244.
[6] Edmund M. Clarke, Orna Grumberg, Doron A. Peled, Model Checking, The MIT Press, 1999.
[7] Dimitra Giannakopoulou, Jeff Magee, Fluent model checking for event-based systems, SIGSOFT Software Engineering Notes 28 (5) (2003) 257–266.
[8] Noa Globerman, David Harel, Complexity results for two-way and multi-pebble automata and their logics, Theoretical Computer Science 169 (2) (1996) 161–184.
[9] Gürcan Güleşir, Evolvable Behavior Specifications Using Context-Sensitive Wildcards, Ph.D. Thesis, University of Twente, Enschede, March 2008.
[10] Gürcan Güleşir, Klaas van den Berg, Lodewijk Bergmans, Mehmet Akşit, Experimental evaluation of a tool for the verification and transformation of source code in event-driven systems, Empirical Software Engineering 14 (6) (2009) 720–777.
[11] D. Harel, A. Pnueli, On the development of reactive systems, in: Logics and Models of Concurrent Systems, Springer-Verlag, New York, Inc., USA, 1985, pp. 477–498.
[12] David Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8 (June (3)) (1987) 231–274.
[13] John Hatcliff, Matthew B. Dwyer, Using the bandera tool set to model-check properties of concurrent java software, in: CONCUR '01: proceedings of the 12th International Conference on Concurrency Theory, Springer-Verlag, London, UK, 2001, pp. 39–58.
[14] Claude Jard, Thierry Jéron, TGV: theory, principles and algorithms: a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems, International Journal on Software Tools for Technology Transfer 7 (4) (2005) 297–315.
[15] Michael Kaminski, Nissim Francez, Finite-memory automata, Theoretical Computer Science 134 (2) (1994) 329–363.
[16] B.W. Kernighan, D.M. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
[17] Emmanuel Letier, Jeff Kramer, Jeff Magee, Sebastian Uchitel, Fluent temporal logic for discrete-time event-based models, in: ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM Press, New York, NY, USA, 2005, pp. 70–79.
[18] Peter Linz, An Introduction to Formal Languages and Automata, Jones and Bartlett Publishers, Inc., USA, 2001.
[19] Tova Milo, Dan Suciu, Victor Vianu, Typechecking for XML transformers, in: Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, ACM, 2000, pp. 11–22.
[20] Frank Neven, Thomas Schwentick, Victor Vianu, Finite state machines for strings over infinite alphabets, ACM Transactions on Computational Logic 5 (3) (2004) 403–435.
[21] Robert W. Sebesta, Concepts of Programming Languages, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
[22] Seppo Sippu, Eljas Soisalon-Soininen, Parsing Theory, vol. 1: Languages and Parsing, Springer-Verlag, New York, Inc., New York, NY, USA, 1988.
[23] Remco van Engelen, Jeroen Voeten (Eds.), Ideals: Evolvability of Software-Intensive High-tech Systems, Embedded Systems Institute, Eindhoven, The Netherlands, 2007.
[24] Roel Wieringa, A survey of structured and object-oriented software specification methods and techniques, ACM Computing Surveys 30 (4) (1998) 459–527.