Science of Computer Programming 183 (2019) 102300

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Soft component automata: Composition, compilation, logic, and verification ☆

Tobias Kappé [a,*], Benjamin Lion [b], Farhad Arbab [b,c], Carolyn Talcott [d]

[a] *University College London, London, United Kingdom*
[b] *Centrum Wiskunde & Informatica, Amsterdam, the Netherlands*
[c] *LIACS, Leiden University, Leiden, the Netherlands*
[d] *SRI International, Menlo Park, USA*

ARTICLE INFO

Article history:
Received 1 April 2018
Received in revised form 12 June 2019
Accepted 12 August 2019
Available online 16 August 2019

Keywords:
Compositionality
Diagnosis
Soft preference
Compilation
Verification

ABSTRACT

The design of a complex system warrants a compositional methodology, i.e., composing simple components to obtain a system that meaningfully exhibits their collective behavior. We propose an automaton-based paradigm for compositional design of such systems where an *action* is accompanied by one or more *preferences*. At run-time, these preferences provide a natural fallback mechanism for the component, while at design-time they can be used to reason about the behavior of the component in an uncertain physical world. Using algebraic structures on preferences and actions, we can compose formal representations of individual components or agents to obtain a representation of the composed system, exhibiting intuitively meaningful behavior.

We extend linear temporal logic with two unary connectives that reflect the compositional structure of actions, and show that it is decidable whether all behaviors of a given automaton satisfy a formula of this extended logic. We then show how this logic can be used to diagnose undesired behavior by tracing the falsification of a specification back to one or more culpable components. Lastly, we implement a toolchain that compiles our automata to Maude, allowing us to apply the rich model checking capability of Maude to verify agent behavior.

1. Introduction

Consider the design of a software package that steers a crop surveillance drone. Such a drone should survey a field and relay the locations of possible signs of disease to its owner. There are a number of concerns at play, including but not limited to maintaining altitude, battery levels and distance from birds of prey. In such a situation, it is best practice to isolate these separate concerns into different modules — allowing for code reuse, and requiring the use of well-defined protocols in case coordination between modules is necessary. One would also like to verify that the designed system satisfies desired properties, such as “even with little energy left, the drone can always reach the charging station”.

In the event that the designed system violates its verification requirements or exhibits behavior that does not conform to the specification, it is often useful to have an example of such behavior. For instance, if the surveillance drone fails to

☆ This paper is an extended version of a paper published at FACS’17 [33].
[*] Corresponding author.
E-mail address: tkappe@cs.ucl.ac.uk (T. Kappé).

https://doi.org/10.1016/j.scico.2019.08.001
0167-6423/© 2019 Elsevier B.V. All rights reserved.

maintain altitude, an example of behavior where this happens could tell us that the drone attempted to reach the far side of the field and ran out of energy.

Additionally, failure to verify an LTL-like formula typically comes with a counterexample — indeed, a counterexample arises from the automata-theoretic verification approach quite naturally [51]. Taking this idea of *diagnostics* one step further in the context of a compositional design, it would also be useful to be able to identify the components responsible for allowing a behavior that deviates from the specification, whether this behavior comes from a run-time observation or a design-time counterexample to a desired property. The designer then knows which components should be adjusted (in our example, this may turn out to be the route planning component), or, at the very least, rule out components that are not directly responsible (such as the wildlife evasion component).

In this paper, we propose an automata-based paradigm based on soft constraint automata [2,32], called soft component automata (SCAs[1]). An SCA is a state-transition system where transitions are labeled with actions and preferences. Higher-preference transitions typically contribute more towards the goal of the component; if a component is in a state where it wants the system to move north, a transition with action north has a higher preference than a transition with action south. At run-time, preferences provide a natural fallback mechanism for an agent: in ideal circumstances, the agent would perform only actions with the highest preferences, but if the most-preferred actions fail, the agent may be permitted to choose a transition of lower preference. At design-time, preferences can be used to reason about the behavior of the SCA in suboptimal conditions, by allowing all actions whose preference is bounded from below by a threshold. In particular, this is useful if the designer wants to determine the circumstances where a property is no longer verified by the system.

Because the actions and preferences of an SCA reside in algebraic structures, we can define a composition operator on SCAs that takes into account the composition of actions as well as preferences. The result of composition of two SCAs is another SCA where actions and preferences reflect those of the operands. As we shall see, SCAs are amenable to verification against formulas in linear temporal logic (LTL), i.e., one can check whether the behavior of an SCA is contained in the behavior allowed by a formula of LTL. Moreover, SCAs can be compiled to Maude, which allows us to use the model checking tools present there to verify their behavior.

Soft component automata are a generalization of constraint automata [5]. The latter can be used to coordinate interaction between components in a verifiable fashion [3]. Just like constraint automata, the framework we present blurs the line between *computation* and *coordination* — both are captured by the same type of automata. Consequently, this approach allows us to reason about these concepts in a uniform fashion: coordination is not separate in the model, it is effected by components which are inherently part of the model.

The contributions of this paper are as follows. First, we propose an automata-based and compositional design paradigm for autonomous agents in which robustness is a first-class concept, by allowing the designer to specify alternative actions that are available in less-than-ideal circumstances. Using these alternatives, an agent can still achieve some subset of its goals or its original goals to a lesser degree. Second, we put forth a dialect of LTL that accounts for the compositional structure of actions and can be used to verify guarantees about the behavior of components, as well as their behavior in composition. We also show that whether the behavior of an automaton satisfies the specification in an LTL formula is decidable in our model. Third, exploiting the algebraic structure of preferences, we propose a method to trace errant behavior back to one or more components. This method can be used with both run-time and design-time failures: in the former case, the behavior arises from the action history of the automaton, in the latter case it is a counterexample obtained from verification. Fourth, we describe a compiler that first translates a specification of our automata in an intermediate representation based on the Reo coordination language, and then outputs a Maude program that implements the automata. This Maude program can then be used to ask verification questions (matching the proposed dialect of LTL), and its output can be used in diagnosis.

The remainder of this paper is organized as follows. In Section 2, we mention some related work; in Section 3 we discuss the necessary notation and mathematical structures. In Section 4, we introduce soft component automata, along with a toy model. We discuss the syntax and semantics of the LTL-like logic used to verify properties of SCAs in Section 5. In Section 6, we propose a method to extract which components bear direct responsibility for a failure. In Section 7, we describe the implementation of an SCA-to-Maude compiler, and show how its output can be leveraged for verification and diagnosis. Our conclusions comprise Section 8, and some directions for further work appear in Section 9.

2. Related work

The algebraic structure for preferences called the *constraint semiring* was proposed by Bistarelli et al. [8,7], to reason about a type of constraint satisfaction problem where constraints are tagged with preference values that signal the need for a constraint to be satisfied. Further exploration of such structures appears in [9,22]; composition of preference structures is discussed in [25,32].

The structure we propose for modeling actions and their compositions can be thought of as an algebraic reconsideration of *static constructs* [26]. Both our formalism and static constructs are intended as a handle on how the actions of individual subsystems combine into the actions of the system at large. In contrast with static constructs, however, our method is

[1] Here, we use the abbreviation *SCA* exclusively to refer to soft *component* automata.

more hierarchical, because actions are composed in a pairwise manner. Consequently, our approach is more compositional: actions of subsystems compose into actions of a larger subsystem, which may yet compose into actions of an even larger subsystem; the static constructs encountered in [26] apply to only one such step.

The automata formalism used in this paper generalizes *soft constraint automata* [2], which were originally proposed to give descriptions of Web Services [2]; these are, in turn, based on *constraint automata* [5], an automata model used to give an operational semantics to the graphical coordination language Reo [1]. In [32], soft constraint automata were first used to model fault-tolerant, compositional autonomous agents. Soft component automata come with an abstract view on actions and their composition, whereas for soft constraint automata this notion is embedded in the concept of *ports* and constraints imposed on those ports for each transition. Our perspective on actions allows us to treat them more abstractly, for instance by reasoning about composability of actions, a notion that finds its way into the logic that we use for verification.

Weighted automata were initially introduced by Schützenberger [48], and are used to qualitatively reason about the acceptance of a word. Schützenberger characterized the behavior of such automata as rational formal power series, where weights are algebraically defined as elements of a semiring. Several formalisms were shown to be equivalent to describe the behavior of weighted automata, such as rational series, linear presentations, or quantitative logics [18]. A version of Büchi's theorem is presented for weighted automata over finite words in [17]: the formal power series definable by certain weighted sentences in monadic second-order logic (MSO) coincide with the series recognizable by weighted automata. In [19], this result is extended to weighted Büchi automata over infinite words. Our use of weights is slightly different, in the sense that we map a word to a sequence of weights, instead of their product. We use the sequence of weights to define a partial order on words sharing the same prefix. The "diagnostic preference" introduced in Section 6, is the closest related work with weighted automata: each word is mapped to a weight, and its acceptance condition depends on whether the weight is below the threshold.

Using preference values to specify the behavior of autonomous agents is also explored from the perspective of rewriting logic in the *soft agent framework* [49,50]. Experiments with the soft agent framework show that behavior based on soft constraints can indeed contribute to robustness [40]. The soft agent framework has been extended with a family of generic fault models, protocols to specify desired behavior, constraint solving techniques for checking that a trace is compliant with a protocol, and algorithms to detect local deviations from expected effects of actions. A notion of *gedankenexperiment* (cf. [45]) is proposed to test if a given fault can be blamed for failure to follow a protocol [34]. In the extended soft agent work the emphasis is put on finding the kinds and number of faults for which the agents ability to adapt, using preferences, is insufficient, assuming that in the absence of faults the agents will succeed in carrying out the specified protocol.

Sampath et al. [47] discuss methods to detect unobservable errors based on a model of the system and a trace of observable events; others extended this approach [14,43] to a multi-component setting. Casanova et al. [11] wrote about fault localization in a system where some components are unobservable, based on which computations (tasks involving multiple components) fail. In these paradigms, one tries to find out where a *runtime fault* occurs; in contrast, we try to find out which component is responsible for *undesired behavior*, i.e., behavior that is allowed by the system but not desired by the specification.

A general framework for fault ascription in concurrent systems based on *counterfactuals* is presented in [23,24]. Formal definitions are given for necessary and/or sufficient conditions under which failures in a given set of components cause a system to violate a given property. Components are specified by sets of sets of events (analogous to actions) representing possible correct behaviors. A parallel (asynchronous) composition operation is defined on components, but there is no notion of composition of events or explicit interaction between components. A system is given by a global behavior (a set of event sets) together with a set of system component specifications. The global behavior, which must be provided separately, includes component events, but may also have other events, and may violate component specifications (hence the faulty components). In our approach, global behavior is obtained by component composition. Undesired behavior may be local to a component or emerge as the result of interactions among components.

Fontana et al. [37] address the problem of finding causal relations among events in traces generated by stochastic simulation of rule based systems (specifically cell signaling systems modeled in the Kappa language). An event is the application of a rule instance in a particular context at particular time. They are interested in questions such as *if event e_0 had not occurred would event e_1 have occurred?*. To answer this question, the authors introduce a notion of counterfactual trace, (τ, i, τ'), where τ is the 'factual trace', i represents the intervention corresponding to the blocking of a given event, and τ' is a counterfactual to τ where the given event did not happen. The challenge is to formalize the property that τ' agrees with τ as much as possible, while avoiding the blocked event, which is done in this paper by adapting the stochastic simulation structure. The authors give probabilistic semantics to statements of the form $\tau \models [i]\psi$ (had the intervention i happened in τ then ψ would have been true), by sampling counterfactual traces and evaluating the probability of τ' satisfying ψ. They refine the resulting causal relations using notions of enabling and prevention. The authors point out that systematic methods to identify events to test for causality are missing although they mention some application specific heuristics. We regard preferences and probabilities as different dimensions in defining the space of possible actions/events. Our models focus on preferences (which action is better in some sense, controlled by the agent) while the Kappa models work with probabilities (which event is more likely, controlled by the environment). Considering the two dimensions, it is interesting to consider: (1) how the notion of counterfactual trace can be adapted to traces generated by exploring different preferences; and (2) whether there is an extension of our model with probabilistic environments (for example fault/failure models) where the above notion of counterfactual trace applies fairly directly.

In LTL, a counterexample to a negative result arises naturally if one employs automata-based verification techniques [42, 51]. In this paper, we further exploit counterexamples to gain information about the component or components involved in violating the specification. The application of LTL to constraint automata is inspired by an earlier use of LTL for constraint automata [3].

Some material in this paper appeared in the first author's master's thesis [30].

3. Preliminaries

If S is a set, then 2^S denotes the set of subsets of S, i.e., the *powerset* of S. We write S^* for the set of *words* over S, and if $w \in S^*$ we write $|w|$ for the *length* of w. We write $w(n)$ for the n-th letter of w (starting at 0). Furthermore, let S^ω denote the set of functions from $\mathbb{N}$ to S, also known as *streams* over S [46]. We define for $w \in S^\omega$ that $|w| = \omega$ (the smallest infinite ordinal). Concatenation of a stream or word to a word is defined as expected.

We use the superscript ω to denote infinite repetition — for instance, writing $w = (01)^\omega$ for the infinite stream of alternating zeroes and ones. We write S^π for the set of *eventually periodic* streams in S^ω, i.e., $w \in S^\omega$ such that there exist $w_h, w_t \in \Sigma^*$ with $w = w_h \cdot w_t^\omega$. We write $w^{(k)}$ with $k \in \mathbb{N}$ for the *k-th derivative* of w, which is given by $w^{(k)}(n) = w(k+n)$.

A *tree* over a set S is a pair $T = \langle N, \lambda \rangle$, where $N \subseteq \mathbb{N}^*$ is non-empty and prefix-closed (i.e., if $wx \in N$ then $w \in N$), and $\lambda : T \to S$ is the *labeling* function. We write T_0 for the *head* of T, which is given by $\lambda(\epsilon)$. A *branch* of T is a prefix-closed set $N' \subseteq N$ such that if $w \in N'$, then there exists exactly one $n \in \mathbb{N}$ such that $wn \in N'$; note that a branch may be infinite if the tree has infinitely many nodes. We write $\mathcal{T}(S)$ for the set of trees over S.

We write $\mathcal{B}(S)$ for the set of all *positive Boolean formulas* over S, i.e., the expressions built from elements of S, the constants **false** and **true**, and the operators $\vee$ and $\wedge$. Also, $\mathcal{S}(S)$ is the set of all *semilattice formulas* over S, i.e., the subset of $\mathcal{B}(S)$ built from elements of s, the constant **false** and the operator $\vee$. When $S' \subseteq S$ and $B \in \mathcal{B}(S)$, we write $S' \models B$ if B is true when all $s \in S'$ are evaluated as **true**, and all $s \in S \setminus S'$ are evaluated as **false**, where $\vee$ and $\wedge$ are assigned their obvious meaning.

If S is a set and $\odot : S \times S \to S$ a function, we refer to $\odot$ as an *operator on* S and write $p \odot q$ instead of $\odot(p,q)$. We always use parentheses to disambiguate expressions if necessary. To model composition of actions, we need a slight generalization. If $R \subseteq S \times S$ is a relation and $\odot : R \to S$ is a function, we refer to $\odot$ as an R-*operator on* S; we also use infix notation by writing $p \odot q$ instead of $\odot(p,q)$ whenever pRq.

If $\odot : R \to S$ is an R-operator on S, we refer to $\odot$ as *idempotent* if $p \odot p = p$ for all $p \in S$ such that pRp, and *commutative* if $p \odot q = q \odot p$ whenever $p, q \in S$, pRq and qRp. Lastly, $\odot$ is *associative* when for all $p, q, r \in S$, pRq and $(p \odot q)Rr$ hold if and only if qRr and $pR(q \odot r)$; moreover, either of these implies that $(p \odot q) \odot r = p \odot (q \odot r)$. When $R = S \times S$, we recover the canonical definitions of idempotency, commutativity and associativity.

A *constraint semiring*, or *c-semiring*, provides a structure on preference values that allows us to *compare* the preferences of two actions to see if one is preferred over the other as well as *compose* preference values of component actions to find out the preference of their composed action. Formally, a c-semiring [8,7] is a tuple $\langle \mathbb{E}, \leq, \bigoplus, \otimes \rangle$ such that all of the following hold[2]:

(i) $\langle \mathbb{E}, \leq, \bigoplus \rangle$ is a complete join-semilattice, i.e., $\langle \mathbb{E}, \leq \rangle$ is a partially ordered set and $\bigoplus : 2^{\mathbb{E}} \to \mathbb{E}$ is the least upper bound operator. Concretely, this means that if $E \subseteq \mathbb{E}$, then $\bigoplus E$ is the least element of $\mathbb{E}$ such that for all $e \in E$ it holds that $e \leq \bigoplus E$.

(ii) $\otimes : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ is a commutative and associative operator, such that for $e \in \mathbb{E}$ and $E \subseteq \mathbb{E}$, it holds that

$$e \otimes \bigoplus E = \bigoplus \{e \otimes e' : e' \in E\}$$

We often denote a c-semiring by its carrier; if we refer to $\mathbb{E}$ as a c-semiring, the constituent elements of that c-semiring are denoted as $\leq_{\mathbb{E}}$, $\bigoplus_{\mathbb{E}}$, et cetera. We drop the subscript when only one c-semiring is in context.

One possible model of a c-semiring is $\mathbb{W} = \langle \mathbb{R} \cup \{\infty\}, \geq, \inf, \hat{+} \rangle$, called the *weighted c-semiring*, where inf is the infimum and $\geq$ (resp. $\hat{+}$) is comparison (resp. addition) of reals extended to $\mathbb{R} \cup \{\infty\}$ in the obvious way. In this model, preferences are expressed in real numbers (or ∞) which signify their "weight". We note that if $e, e' \in \mathbb{W}$ with $e' \leq_{\mathbb{W}} e$, i.e., e is at least as preferable as e' in $\mathbb{W}$, then in this case $e' \geq e$ — values of *lower weight* express a *higher preference*. We will use the weighted c-semiring in examples throughout this paper.

Note that a c-semiring need not be totally ordered. For instance, consider $\mathbb{U} = \langle 2^{\{\mathsf{R},\mathsf{W},\mathsf{X}\}}, \supseteq, \bigcap, \cup \rangle$, called the *UNIX c-semiring* [6], where preferences are subsets of $\{\mathsf{R},\mathsf{W},\mathsf{X}\}$ that represent access permissions (i.e., *read*, *write*, *execute* respectively) required to implement a plan. In this model, $e, e' \in \mathbb{U}$ with $e' \leq_{\mathbb{U}} e$, i.e., e is at least as preferable as e' in $\mathbb{U}$, when $e' \supseteq e$, i.e., the permissions in e are contained in e'. Thus, $\mathbb{U}$ encodes the "principle of least privilege", where alternatives that require fewer privileges are preferred over those that require more. In particular, $\{\mathsf{R},\mathsf{W}\}$ is incomparable with $\{\mathsf{X}\}$

[2] We deviate from the notation used in earlier work for the sake of brevity, but any c-semiring in the sense of [8,7] is a c-semiring in this sense, and vice versa.

as far as $\mathbb{U}$ is concerned; indeed, these privilege levels are incomparable: there are contexts where read- and write-access to a file might give a user more privileges than execute access, and vice versa.

We write $\mathbf{0}$ for $\bigoplus \emptyset$ and $\mathbf{1}$ for $\bigoplus \mathbb{E}$ (the unique least and greatest elements of $\mathbb{E}$, respectively). The operator $\bigoplus$ induces an idempotent, commutative and associative binary operator $\oplus : \mathbb{E} \times \mathbb{E} \to \mathbb{E}$ by defining $e \oplus e' = \bigoplus(\{e, e'\})$. Note also that $e \oplus \mathbf{0} = e$ and $e \oplus \mathbf{1} = \mathbf{1}$, as well as $e \otimes \mathbf{0} = \mathbf{0}$ and $e \otimes \mathbf{1} = e$ [7].

The least upper bound operator $\bigoplus$ uniquely induces a greatest lower bound operator $\bigwedge_{\mathbb{E}}$, which we will use later on.[3] Lastly, $\otimes$ is *intensive*, meaning that for any $e, e' \in \mathbb{E}$, we have $e \otimes e' \leq e$ [7].

4. Component model

We now discuss the component model that we propose for construction of autonomous agents.

4.1. Component action systems

Observable behavior of agents is the result of the actions put forth by their individual components; we thus need a way to talk about how actions compose. For example, in our crop surveillance drone, the following may occur:

- The communication component wants to synchronize the pictures taken with the base, while the routing component wants to move north. Since “synchronize” and “move north” are not mutually exclusive, they are said to compose *concurrently* into a single action “synchronize while moving north”, and we say that this action *captures* the former two actions.
- The drone has a single antenna that can be used for GPS and communications, but not both at the same time. The component responsible for relaying pictures has finished its transmission and wants to release the antenna, while the navigation component wants to get a fix on the location and requests use of the antenna. In this case, the actions “release privilege” and “obtain privilege” compose *jointly*, into a “transfer privilege” action.
- The routing component proposes to move north, while the wildlife avoidance component notices a hawk approaching from that same direction, and thus wants to move south. In this case, these actions available to the components are contradictory in composition: they cannot be composed, jointly or concurrently. Some other composition of actions from both components that can be composed needs to be selected for the drone to do anything.

All of these possibilities are captured in the definition below.

Definition 1. A *component action system (CAS)* is a tuple $\langle \Sigma, \circledcirc, \boxbox \rangle$, such that Σ is a finite set of *actions*, $\circledcirc \subseteq \Sigma \times \Sigma$ is a reflexive and symmetric relation and $\boxbox : \circledcirc \to \Sigma$ is an idempotent, commutative and associative $\circledcirc$-operator on Σ. We call $\circledcirc$ the *composability relation*, and $\boxbox$ the *composition operator*.

Every CAS $\langle \Sigma, \circledcirc, \boxbox \rangle$ induces a relation $\sqsubseteq$ on Σ, where for $a, b \in \Sigma$, $a \sqsubseteq b$ if and only if there exists a $c \in \Sigma$ such that a and c are composable ($a \circledcirc c$) and they compose into b ($a \boxbox c = b$). One can easily verify that $\sqsubseteq$ is a preorder; accordingly, we call $\sqsubseteq$ the *capture preorder* of the CAS.

As with c-semirings, we may refer to a set Σ as a CAS. When we do, we denote its composability relation, composition operator and preorder by $\circledcirc_\Sigma$, $\boxbox_\Sigma$ and $\sqsubseteq_\Sigma$. We drop the subscript when there is only one CAS in context.

We model incomposability of actions by omitting them from the composability relation; i.e., if south is an action that compels the agent to move south, while north drives the agent north, we set $\mathsf{south} \not\circledcirc \mathsf{north}$. Note that $\circledcirc$ is not necessarily transitive. This makes sense in the scenarios above, where $\mathsf{snapshot}$ is composable with south as well as north, but north is incomposable with south.

As a consequence of the definition of a CAS, we obtain a property akin to *conflict inheritance* (as studied for event structures, cf. [53]). More specifically, we find that incomposability of actions carries over to their compositions: if $\mathsf{south} \circledcirc \mathsf{snapshot}$ and $\mathsf{south} \not\circledcirc \mathsf{north}$, also $(\mathsf{south} \boxbox \mathsf{snapshot}) \not\circledcirc \mathsf{north}$. This is formalized in the following lemma.

Lemma 1. *Let $\langle \Sigma, \circledcirc, \boxbox \rangle$ be a CAS and let $a, b, c \in \Sigma$. If $a \circledcirc b$ but $a \not\circledcirc c$, then $(a \boxbox b) \not\circledcirc c$. Moreover, if $a \not\circledcirc c$ and $a \sqsubseteq b$, then $b \not\circledcirc c$.*

Proof. For the first claim, suppose that $(a \boxbox b) \circledcirc c$. Then, since $\boxbox$ is a commutative $\circledcirc$-operator, we know that $(b \boxbox a) \circledcirc c$; moreover, since $\boxbox$ is an associative $\circledcirc$-operator, it follows that $a \circledcirc c$ and $b \circledcirc (a \boxbox c)$, the former of which contradicts the premise that $a \not\circledcirc c$. We thus conclude that $(a \boxbox b) \not\circledcirc c$.

For the second claim, suppose that $a \not\circledcirc c$ and $a \sqsubseteq b$. Then there exists a $d \in \Sigma$ such that $a \circledcirc d$ and $a \boxbox d = b$. By the above, $b = (a \boxbox d) \not\circledcirc c$. □

[3] In general, for c-semirings with an infinite carrier, having a *complete* join-semilattice structure, i.e., allowing the computation of arbitrary joins, is necessary for this greatest lower bound operator to exist. For our purposes, one can also define a c-semiring based on a lattice (with binary meet and join), but we choose the notation in [7] for the sake of continuity.

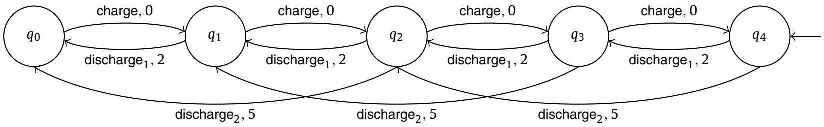


Fig. 1. A component modeling energy management, A_e.

The composition operator facilitates orthogonal as well as logical composition. Given actions obtain, release and transfer, with their interpretation as in the second scenario, we can encode that obtain and release are composable by stipulating that obtain $\circledcirc$ release, and say that their (logical) composition involves an exchange of privileges by choosing obtain $\boxdot$ release $=$ transfer. Furthermore, the capture preorder describes our intuition of capturing: if snapshot and move match the first scenario, with snapshot $\circledcirc$ north, then snapshot, north $\sqsubseteq$ snapshot $\boxdot$ north.

Port automata [35] contain a model of a CAS. Here, actions are sets of symbols called *ports*, i.e., elements of 2^P for some finite set P. Actions $\alpha, \beta \in 2^P$ are compatible when they agree on a fixed set $\gamma \subseteq P$, i.e., if $\alpha \cap \gamma = \beta \cap \gamma$, and their composition is $\alpha \cup \beta$. Similarly, we also find an instance of a CAS in *(soft) constraint automata* [5,2]; see [30] for a full discussion of this correspondence.

4.2. Soft component automata

Having introduced the structure we impose on actions, we are now ready to discuss the automaton formalism that specifies the sequences of actions that are allowed, along with the preferences attached to such actions.

Definition 2. A *soft component automaton (SCA)* is a tuple $\langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t \rangle$ where Q is a finite set of *states*, with $q^0 \in Q$ the *initial state*, Σ is a CAS and $\mathbb{E}$ is a c-semiring, with $t \in \mathbb{E}$ the *threshold*. Lastly, $\rightarrow \subseteq Q \times \Sigma \times \mathbb{E} \times Q$ is a finite relation called the *transition relation*. We write $q \xrightarrow{a, e} q'$ when $\langle q, a, e, q' \rangle \in \rightarrow$.

An SCA models the actions available in each state of the component, how much these actions contribute towards the goal and the way actions transform the state. The threshold value restricts the available actions to those with a preference bounded from below by the threshold, either at run-time, or at design-time when one wants to reason about behaviors satisfying some minimum preference.

We stress here that the threshold value is purposefully defined as part of an SCA, rather than as a parameter to the semantics in Section 4.4. This allows us to speak of the preferences of an individual component, rather than a threshold imposed on the whole system; instead, the threshold of the system arises from the thresholds of its components, which is especially useful in Section 6.

We depict SCAs in a fashion similar to the graphical representation of finite state automata: as a labeled graph, where vertices represent states and edges represent transitions, labeled with elements of its CAS and c-semiring. The initial state is indicated by an arrow without origin. The CAS, c-semiring and threshold value will always be made clear where they are germane to the discussion.

An example of an SCA is A_e, drawn in Fig. 1; its CAS contains the incomposable actions charge, discharge$_1$ and discharge$_2$, and its c-semiring is the weighted c-semiring $\mathbb{W}$. This particular SCA can model the component of the crop surveillance drone responsible for keeping track of the amount of remaining energy in the system; in state q_n (for $n \in \{0, 1, \ldots, 4\}$), the drone has n units of energy left, meaning that in states q_1 to q_4, the component can spend one unit of energy through discharge$_1$, and in states q_2 to q_4, the drone can consume two units of energy through discharge$_2$. In states q_0 to q_3, the drone can try to recharge through charge.[4] Recall that, in $\mathbb{W}$, higher values reflect a lower preference (a higher *weight* or *cost*); thus, charge is preferred over discharge$_1$.

Here, A_e is meant to describe the possible behavior of the energy management component only. Availability of the actions within the *total model* of the drone (i.e., the composition of all components) is subject to how actions compose with those of other components; for example, the availability of charge may depend on the state of the component modeling position. Similarly, preferences attached to actions concern energy management only. In states q_0 to q_3, the component prefers to top up its energy level through charge, but the preferences of this component under composition with some other component may cause the composed preferences of actions composed with charge to be different. For instance, the total model may prefer executing an action that captures discharge$_2$ over one that captures charge when the former entails movement and the latter does not, especially when survival necessitates movement.

Nevertheless, the preferences of A_e affect the total behavior. For instance, the weight of spending one unit of energy (through discharge$_1$) is lower than the weight of spending two units (through discharge$_2$). This means that the energy component prefers to spend a small amount of energy in a single step. This reflects a level of care: by preferring small steps, the component hopes to avoid situations where too little energy is left to avoid disaster.

[4] A more detailed model is possible by extending SCAs with *memory cells* [29] and using a memory cell to store the energy level. In such a setup, a state would represent a *range* of energy values that determines the component's disposition regarding resources.

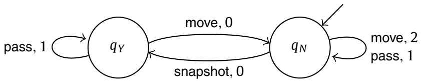


Fig. 2. A component modeling the desire to take a snapshot at every location, A_s.

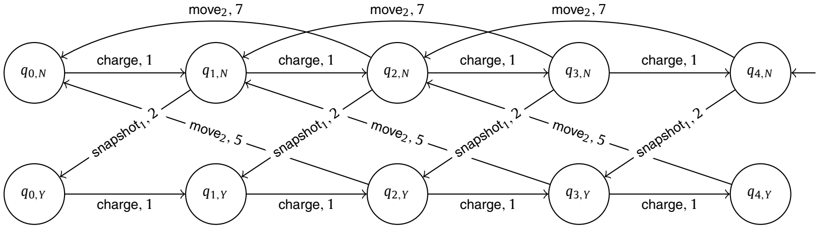


Fig. 3. The composition of the SCAs A_e and A_s, dubbed $A_{\mathrm{e,s}}$: a component modeling energy and snapshot management. We abbreviate pairs of states $\langle q_i, q_j\rangle$ by writing $q_{i,j}$.

4.3. Composition

Composition of two SCAs arises naturally, as follows.

Definition 3. Let $A_i = \langle Q_i, \Sigma, \mathbb{E}, \to_i, q_i^0, t_i\rangle$ be an SCA for $i \in \{0,1\}$. The *(parallel) composition* of A_0 and A_1 is the SCA $\langle Q, \Sigma, \mathbb{E}, \to, q^0, t_0 \otimes t_1\rangle$, denoted $A_0 \bowtie A_1$, where $Q = Q_0 \times Q_1$, $q^0 = \langle q_0^0, q_1^0\rangle$, $\otimes$ is the composition operator of $\mathbb{E}$, and $\to$ is the smallest relation satisfying

$$\frac{q_0 \xrightarrow{a_0,\, e_0}_0 q_0' \qquad q_1 \xrightarrow{a_1,\, e_1}_1 q_1' \qquad a_0 \circledcirc a_1}{\langle q_0, q_1\rangle \xrightarrow{a_0 \boxdot a_1,\; e_0 \otimes e_1} \langle q_0', q_1'\rangle}$$

In a sense, composition is a generalized product of automata, where composition of actions is mediated by the CAS: transitions with composable actions manifest in the composed automaton, as transitions with composed action and preference.

Composition is defined for SCAs that share CAS and c-semiring. Absent a common CAS, we do not know which actions compose, and what their compositions are. However, composition of SCAs with different c-semirings does make sense when the components model different concerns (e.g., for our crop surveillance drone, "minimize energy consumed" and "maximize covering of snapshots"), both contributing towards the overall goal. Earlier work on soft constraint automata [32] explored this possibility. The additional composition operators proposed there can easily be applied to soft component automata.

A state q of a component may become unreachable after composition, in the sense that no state composed of q is reachable from the composed initial state. For example, in the total model of our drone, it may occur that any state representing the drone at the far side of the field is unreachable, because the energy management component prevents some transition for lack of energy. Contrarily, since the composed automaton has a lower threshold than that of the components (by intensivity), it may also be the case that there are states in the composed automaton whose constituent states were previously unreachable in the components; this happens when the (composed) preference of a transition along the path outweighs the composed threshold preference.

To discuss an example of SCA composition, we introduce the SCA A_s in Fig. 2, which models the crop surveillance drone's objective to take a snapshot of every location before moving to the next. The CAS of A_s includes the pairwise incomposable actions pass, move and snapshot, and its c-semiring is the weighted c-semiring $\mathbb{W}$. We leave the threshold value t_s undefined for now. The purpose of A_s is reflected in its states: q_Y (resp. q_N) represents that a snapshot of the current location was (resp. was not) taken since moving there. If the drone moves to a new location, the component moves to q_N, while q_Y is reached by taking a snapshot. If the drone has not yet taken a snapshot, it prefers to do so over moving to the next spot (missing the opportunity).[5]

We grow the CAS of A_e and A_s to include the actions move, move$_2$, snapshot and snapshot$_1$ (here, the action α_i is interpreted as "execute action α and account for i units of energy spent"), and $\circledcirc$ is the smallest reflexive, commutative and transitive relation such that the following hold: move $\circledcirc$ discharge$_2$ (moving costs two units of energy), snapshot $\circledcirc$ discharge$_1$ (taking a snapshot costs one unit of energy) and pass $\circledcirc$ charge (the snapshot state is unaffected by charging). We also choose move $\boxdot$ discharge$_2$ = move$_2$, snapshot $\boxdot$ discharge$_1$ = snapshot$_1$ and pass $\boxdot$ charge = charge. The composition of A_e and A_e is depicted in Fig. 3.

[5] A more detailed description of such a component may count the number of times the drone has moved without taking a snapshot first, and assign a preference accordingly.

The structure of $A_{\mathsf{e,s}}$ reflects that of A_{e} and A_{s}; for instance, in state $q_{2,Y}$ two units of energy remain, and we have a snapshot of the current location. The same holds for the transitions of $A_{\mathsf{e,s}}$; for example, $q_{2,N} \xrightarrow{\mathsf{snapshot}_1,\ 2} q_{1,Y}$ is the result of composing $q_2 \xrightarrow{\mathsf{discharge}_1,\ 2} q_1$ and $q_N \xrightarrow{\mathsf{snapshot},\ 0} q_Y$.

Also, note that in $A_{\mathsf{e,s}}$ the preference of the move_2-transitions at the top of the figure is lower than the preference of the diagonally-drawn move_2-transitions. This difference arises because the component transition in A_{s} of the former is $q_N \xrightarrow{\mathsf{move},\ 2} q_N$, while that of the latter is $q_Y \xrightarrow{\mathsf{move},\ 0} q_N$. As such, the preferences of the component SCAs manifest in the preferences of the composed SCA.

The action $\mathsf{snapshot}_1$ is not available in states of the form $q_{i,Y}$, because the only action available in q_Y is pass, which does not compose into $\mathsf{snapshot}_1$.

4.4. Behavioral semantics

The final part of our component model is a description of the behavior of SCAs. Here, the threshold determines which actions have sufficient preference for inclusion in the behavior. Intuitively, the threshold is an indication of the amount of flexibility allowed. In the context of composition, lowering the threshold of a component is a form of compromise: the component potentially gains behavior available for composition. Setting a lower threshold makes a component more permissive, but may also make it harder (or impossible) to achieve its goal.

The question of where to set the threshold is one that the designer of the system should answer based on the properties and level of flexibility expected from the component; Section 5 addresses the formulation of these properties, while Section 6 discusses adjusting the threshold.

Definition 4. Let $A = \langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t \rangle$ be an SCA. We say that a stream $\sigma \in \Sigma^\omega$ is a *behavior* of A if there exist streams $\mu \in Q^\omega$ and $\nu \in \mathbb{E}^\omega$ such that $\mu(0) = q^0$, and for all $n \in \mathbb{N}$, we have $t \leq \nu(n)$ and $\mu(n) \xrightarrow{\sigma(n),\ \nu(n)} \mu(n+1)$. The set of behaviors of A, denoted by $L(A)$, is called the *language* of A.

Note the similarity between the behavior of an SCA and that of Büchi-automata [10]; we elaborate on this similarity in [31].

Remark 1. Similar to the semantics of Büchi-automata [10], the semantics of SCAs does not include finite traces of actions. In particular, this means that states without outgoing transitions as well as states where all outgoing transitions are below the threshold (i.e., “deadlock states”), do not play a part in the semantics. Consequently, the absence (or presence) of traces that lead to these states cannot be subject to verification. Since the use case for our model is systems that should operate indefinitely, we (tacitly) exclude SCAs with states where no outgoing transition is possible, and moreover assume that the threshold is never so high as to prevent all outgoing transitions of a state. This assumption, however, comes at the cost of two important caveats.

The first caveat is that deadlock states may arise as a result of composition, either as a result of the composed threshold, or because the outgoing transitions of two (reachable) states all have actions that are incomposable. It is therefore important to ascertain that a composite system still satisfies the side-condition on deadlock states — if it does not, then this could indicate that the systems being composed do not cooperate well, for example because the CAS does not provide enough composable actions, or because the threshold of one system is too strict. In the latter case, the designer may elect to lower the threshold of one or more components, enabling transitions that prevent the deadlock.

The second caveat is that when the threshold of an SCA is increased to eliminate behavior that violates some desired property, this should not be achieved by raising the threshold to the point where undesired behavior is excluded by having the system deadlock at an earlier point. If such an increase in threshold is necessary to accomplish the desired goal, one could say that the system prevents bad behavior by crashing, which itself is not desirable.

The example SCAs that we have considered so far avoid deadlocks caused by a lack of outgoing transitions (even in composition). We briefly return to the second caveat when we discuss our procedure for eliminating undesired behavior.

Consider $\sigma = \langle \mathsf{snapshot}, \mathsf{move}, \mathsf{move} \rangle^\omega$ and $\tau = \langle \mathsf{snapshot}, \mathsf{move}, \mathsf{pass} \rangle^\omega$. We can see that when $t_{\mathsf{s}} = 2$, both are behaviors of A_{s}; when $t_{\mathsf{s}} = 1$, τ is a behavior of A_{s}, while σ is not, since every second move-action in σ has preference 2. More generally, if A and A' are SCAs over c-semiring $\mathbb{E}$ that differ only in their threshold values $t, t' \in \mathbb{E}$, and $t \leq t'$, then $L(A') \subseteq L(A)$. In the case of A_{e}, the threshold can be interpreted as a bound on the amount of energy to be spent in a single action; if $t_{\mathsf{e}} < 5$, then behaviors with $\mathsf{discharge}_2$ do not occur in $L(A_{\mathsf{e}})$.

Interestingly, if A_1 and A_2 are SCAs, then $L(A_1 \bowtie A_2)$ is not uniquely determined by $L(A_1)$ and $L(A_2)$. For example, suppose that $t_{\mathsf{e}} = 4$ and $t_{\mathsf{s}} = 1$, and consider $L(A_{\mathsf{e,s}})$, which contains $\langle \mathsf{snapshot} \rangle \cdot \langle \mathsf{move}, \mathsf{snapshot}, \mathsf{charge}, \mathsf{charge}, \mathsf{charge} \rangle^\omega$ even though the corresponding stream of component actions in A_{e}, i.e., the stream $\left\langle \mathsf{discharge}_1 \right\rangle \cdot \left\langle \mathsf{discharge}_2, \mathsf{discharge}_1, \mathsf{charge}, \mathsf{charge}, \mathsf{charge} \right\rangle^\omega$ is not contained in $L(A_{\mathsf{e}})$. This is a consequence of a more general observation for c-semirings, namely that $t \leq e$ and $t' \leq e'$ is sufficient but not necessary to derive $t \otimes t' \leq e \otimes e'$.

$$\frac{\sigma \in \Sigma^\omega}{\sigma \models_\Sigma \top} \qquad \frac{\sigma \in \Sigma^\omega}{\sigma \models_\Sigma \sigma(0)} \qquad \frac{\sigma \models_\Sigma \phi \qquad \sigma \models_\Sigma \psi}{\sigma \models_\Sigma \phi \wedge \psi} \qquad \frac{\sigma \not\models_\Sigma \phi}{\sigma \models_\Sigma \neg\phi}$$

$$\frac{\forall k < n.\ \sigma^{(k)} \models_\Sigma \phi \qquad \sigma^{(n)} \models_\Sigma \psi}{\sigma \models_\Sigma \phi\, U\, \psi} \qquad \frac{\sigma^{(1)} \models_\Sigma \phi}{\sigma \models_\Sigma X\phi}$$

$$\frac{\tau \models_\Sigma \phi \qquad \tau \sqsubseteq^\omega \sigma}{\sigma \models_\Sigma \succ\phi} \qquad \frac{\tau \models_\Sigma \phi \qquad \tau \circledcirc^\omega \sigma}{\sigma \models_\Sigma \circledcirc\phi}$$

Fig. 4. Semantics of our LTL dialect. In these rules, the free variables in the premise (i.e., n in the fifth rule and τ in the last two rules) are implicitly quantified existentially; their types should be apparent. We also define $\sqsubseteq^\omega$ and $\circledcirc^\omega$ as pointwise extensions of $\sqsubseteq$ and $\circledcirc$, i.e., $\sigma \sqsubseteq^\omega \tau$ when, for all $n \in \mathbb{N}$, it holds that $\sigma(n) \sqsubseteq \tau(n)$, and similarly for $\circledcirc^\omega$.

5. Linear temporal logic

We now turn our attention to verifying the behavior of an agent, by means of a simple dialect of linear temporal logic (LTL). We extend LTL in order to reflect the compositional nature of actions in component action systems. This extension has two aspects, which correspond roughly to $\sqsubseteq$ and $\circledcirc$: reasoning about behaviors that *capture* (i.e., are composed of) other behaviors, and about behaviors that are *composable* with other behaviors. For instance, consider the following scenarios:

(i) We want to verify that in certain circumstances, the drone performs a series of actions where it goes north before taking a snapshot. This is useful when, for this particular property, we do not care about other actions that may also be performed while or as part of going north, for instance, whether or not the drone engages in communication while moving.
(ii) We want to verify that every behavior of the snapshot-component is composable with some behavior that eventually recharges. This is useful when we want to abstract away from the action that allows recharging, i.e., it is not important which particular action composes with charge.

Our logic accommodates both scenarios, by providing two new connectives: $\succ\phi$ describes every behavior that captures a behavior satisfying ϕ, while $\circledcirc\phi$ holds for every behavior composable with a behavior that satisfies ϕ.

5.1. Syntax and semantics

The syntax of the LTL dialect we propose for SCAs contains ports conjunction, negation, the U (*until*) and X (*next*) connectives, as well as two new unary connectives $\succ$ (*captures*) and $\circledcirc$ (*composable*), which work as follows:

- The *captures*-connective $\succ$ is intended to extend the semantics of a formula to the behavior that it captures. More precisely, if $\tau \in \Sigma^\omega$ is described by ϕ and $\sigma \in \Sigma^\omega$ captures this τ at every action — i.e., for all $n \in \mathbb{N}$, we have $\tau(n) \sqsubseteq \sigma(n)$ — then σ is a behavior described by $\succ\phi$.
- The *composable*-connective $\circledcirc$ is aimed at translating the semantics of a formula to describe the behavior composable with it. If $\tau \in \Sigma^\omega$ is described by ϕ, and this τ is composable with $\sigma \in \Sigma^\omega$ at every action — i.e., for all $n \in \mathbb{N}$, we have $\tau(n) \circledcirc \sigma(n)$ — then σ is described by $\circledcirc\phi$.

Formally, given a CAS Σ, the language $\mathcal{L}_\Sigma$ is generated by the grammar

$$\phi, \psi ::= \top \mid a \in \Sigma \mid \phi \wedge \psi \mid \phi\, U\, \psi \mid X\phi \mid \neg\phi \mid \succ\phi \mid \circledcirc\phi$$

As a convention, unary connectives take precedence over binary connectives. For example, $\succ\phi\, U \neg\psi$ should be read as $(\succ\phi)\, U (\neg\psi)$. We use parentheses to disambiguate when necessary.

The semantics is given as a relation $\models_\Sigma$ between Σ^ω and $\mathcal{L}_\Sigma$. More precisely, $\models_\Sigma$ is the smallest such relation that satisfies the inference rules in Fig. 4. Although the atoms are formulas of the form $\phi = a \in \Sigma$ that have an exact matching semantics, in general one can use predicates over Σ. We chose not to use predicates here to simplify the presentation of examples.

As usual, we obtain disjunction ($\phi \vee \psi$), implication ($\phi \to \psi$), "always" ($\Box\phi$) and "eventually" ($\Diamond\phi$) from these connectives. For example, $\Diamond\phi$ is defined as $\top\, U\, \phi$, meaning that, if $\sigma \models_\Sigma \Diamond\phi$, there exists an $n \in \mathbb{N}$ such that $\sigma^{(n)} \models_\Sigma \phi$. The operator $\circledcirc$ has an interesting dual that we shall consider momentarily.

We can extend $\models_\Sigma$ to a relation between SCAs (with underlying c-semiring $\mathbb{E}$ and CAS Σ) and formulas in $\mathcal{L}_\Sigma$, by defining $A \models_\Sigma \phi$ to hold precisely when $\sigma \models_\Sigma \phi$ for all $\sigma \in L(A)$. In general, we can see that fewer properties hold as the threshold t approaches the lowest preference in its c-semiring, as a consequence of the fact that decreasing the threshold can only introduce new (possibly undesired) behavior. Limiting the behavior of an SCA to some desired behavior described by a formula thus becomes harder as the threshold goes down, since the set of behaviors exhibited by that SCA is typically larger for lower thresholds.

We view the tradeoff between available behavior and verified properties as essential and desirable in the design of robust autonomous systems, because it represents two options available to the designer. On the one hand, she can make a component more accommodating in composition (by lowering the threshold, allowing more behavior) at the cost of possibly losing safety properties. On the other hand, she can restrict behavior such that a desired property is guaranteed, at the cost of possibly making the component less flexible in composition.

Example 1 *(No wasted moves).* Suppose we want to verify that the agent never misses an opportunity to take a snapshot of a new location. This is expressed by

$$\phi_{\mathsf{w}} = \succ \Box(\mathsf{move} \to X(\neg\mathsf{move}\; U\; \mathsf{snapshot}))$$

This formula reads as "every behavior captures that, at any point, if the current action is a move, then it is followed by a sequence where we do not move until we take a snapshot". Indeed, if $t_e \otimes t_s = 5$, then $A_{\mathsf{e,s}} \models_\Sigma \phi_{\mathsf{w}}$, since in this case every behavior of $A_{\mathsf{e,s}}$ captures that between move-actions we find a snapshot-action. However, if $t_e \otimes t_s = 7$, then $A_{\mathsf{e,s}} \not\models_\Sigma \phi_{\mathsf{w}}$, since $\langle \mathsf{move}_2, \mathsf{move}_2, \mathsf{charge}, \mathsf{charge}, \mathsf{charge}, \mathsf{charge}\rangle^\omega$ would be a behavior of $A_{\mathsf{e,s}}$ that does not satisfy ϕ_{w}, as it contains two successive actions that capture move.[6] This shows the primary use of $\succ$, which is to verify the behavior of a component in terms of the behavior contributed by its subcomponents.

Example 2 *(Room for charging).* We can describe the behavior of an SCA in terms of the behavior it is composable with, by means of the connective $\circledcirc$. For instance, if σ is a behavior of A_{e}, then the structure of A_{e} tells us that the number of non-charge actions between instances of charge is at most four. Now let τ be σ with every instance of charge replaced by pass. Then $\sigma \circledcirc^\omega \tau$; hence, any behavior of A_{e} is composable with some behavior where there are at most four non-pass actions between instances of pass. This allows us to verify (for any t_{e})

$$A_{\mathsf{e}} \models_\Sigma \circledcirc \Box \left(\neg\mathsf{pass} \to \left(X\,\mathsf{pass} \vee X^2\,\mathsf{pass} \vee X^3\,\mathsf{pass} \vee X^4\,\mathsf{pass}\right)\right)$$

where X^n denotes n applications of X.

In words, the formula on the right-hand side means that every behavior is composable with ($\circledcirc$) some behavior where, at any point ($\Box$), if the action is not ($\neg$) a pass, then ($\to$) one of the next four actions ($X \cdots \vee X^2 \cdots$) is a pass.

Example 3 *(Verifying a component interface).* In the previous example, we verified that the behavior of a component A is compatible with behavior captured by some formula. Dually, we can verify that the behavior compatible with A satisfies a formula. Such a property is useful, because it tells us that, in composition, A filters out the behaviors of the other operand that do not satisfy ϕ. In other words, this tells us something about the behavior *imposed* by A in composition. This can be expressed using the $\circledcirc$-connective, by checking whether $A \models_\Sigma \neg \circledcirc \neg\phi$ holds: if this is the case, then for all $\sigma, \tau \in \Sigma^\omega$ with σ a behavior of A and $\sigma \circledcirc^\omega \tau$, we have $\sigma \not\models_\Sigma \circledcirc \neg\phi$, thus in particular $\tau \not\models_\Sigma \neg\phi$ and therefore $\tau \models_\Sigma \phi$.

More concretely, consider the component A_{e}. From its structure, we can tell that the action charge must be executed at least once every five moves. Thus, if τ is composable with a behavior of A_{e}, then τ must also execute some action composable with charge once every five moves. This claim can be encoded by

$$\phi_{\mathsf{c}} = \neg \circledcirc \neg \Box \left(X \circledcirc \mathsf{charge} \vee X^2 \circledcirc \mathsf{charge} \vee \cdots \vee X^5 \circledcirc \mathsf{charge}\right)$$

If $A_{\mathsf{e}} \models_\Sigma \phi_{\mathsf{c}}$, then every behavior of A_{e} is incomposable with a behavior where, at some point, one of its next five actions is not composable with charge. Accordingly, if $\sigma \in \Sigma^\omega$ is composable with some behavior of A_{e}, then, at every point in σ, one of the next five actions must be composable with charge. Behaviors that fail to meet this requirement are excluded from composition with A_{e}.

5.2. Decision procedure

Throughout this section we fix a CAS Σ. We describe a procedure to decide whether $A \models_\Sigma \phi$ holds for a given SCA A and $\phi \in \mathcal{L}_\Sigma$. This procedure leverages an existing technique used for deciding LTL formulas [42,52,51], which relies on a type of automata called Büchi-automata.

Definition 5. A *Büchi-automaton (BA)* is defined as a tuple $A = \langle Q, \delta, q^0, F\rangle$ where Q is a finite set of *states*, with $q^0 \in Q$ the *initial state* and $F \subseteq Q$ the *accepting states*. Lastly, $\delta : Q \times \Sigma \to \mathcal{S}(Q)$ is called the *transition function* of A.

We say that a stream $\sigma \in \Sigma^\omega$ is a *behavior* of A if there exists a stream $\mu \in Q^\omega$ such that $\mu(0) = q^0$, and for $n \in \mathbb{N}$ we have that $\{\mu(n+1)\} \models \delta(\mu(n), \sigma(n))$; furthermore, there exist infinitely many $n \in \mathbb{N}$ such that $\mu(n) \in F$. The set of behaviors of A, denoted $L(A)$, is called the *language* of A.

[6] Recall that move_2 is the composition of move and $\mathsf{discharge}_2$, i.e., $\mathsf{move} \sqsubseteq \mathsf{move}_2$.

The main pattern of decision procedures for LTL-like logics based on Büchi-automata is to translate both the model under verification, M, and the property to verify, ϕ, into BAs A_M and A_ϕ respectively with equivalent semantics, and then compare the languages of A_M and A_ϕ — specifically, if $L(A_M) \subseteq L(A_\phi)$, then any behavior of M is a behavior accepted by A_ϕ, and hence M should satisfy ϕ. As it turns out, the latter question is, in general, decidable for BAs [51,20,21].

Lemma 2. *Let A and A' be BAs. We can decide whether $L(A) \subseteq L(A')$. As a matter of fact, in case of a negative answer, we can obtain* periodic *stream σ that witnesses this — i.e., such that $\sigma \in L(A)$ but $\sigma \notin L(A')$.*

The challenge, then, is to translate both SCAs and formulas in $\mathcal{L}_\Sigma$ into equivalent Büchi-automata. For SCAs, this translation is fairly straightforward.

Lemma 3. *Let A be an SCA. We can construct a BA A' s.t. $L(A) = L(A')$.*

Proof. Let $A = \langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t\rangle$; we choose $A' = \langle Q, \delta, q^0, F\rangle$, where δ is given by choosing for $q \in Q$ and $a \in \Sigma$:

$$\delta(q, a) = \bigvee \{q' \in Q : q \xrightarrow{a,e} q',\ t \leq e\}$$

where $\bigvee$ is the obvious generalization of $\vee$ to subsets of $\mathcal{S}(Q)$; it should be clear that ordering and bracketing of the terms does not matter with regard to $L(A)$.

To see that $L(A) = L(A')$, it suffices to note that if $\mu \in Q^\omega$ witnesses that $\sigma \in L(A)$, then μ also witnesses that $\sigma \in L(A')$, and vice versa. □

We say that a BA A *implements* $\phi \in \mathcal{L}_\Sigma$ if $\sigma \models_\Sigma \phi$ precisely when $\sigma \in L(A)$. It remains to show that for $\phi \in \mathcal{L}_\Sigma$ we can find a BA A_ϕ that implements ϕ. Although this translation is strictly possible using only BAs [42], it is more efficient to first translate ϕ into a more general type of Büchi-automaton known as an *alternating Büchi-automaton* [51].

Definition 6. An *alternating Büchi-automaton (ABA)* is defined as a tuple $A = \langle Q, \delta, q^0, F\rangle$, where Q, q^0 and F are defined as in BAs, and $\delta : Q \times \Sigma \rightarrow \mathcal{B}(Q)$ is referred to as the *(alternating) transition function* of A.

A stream $\sigma \in \Sigma^\omega$ is a *behavior* of A if there exists a $T = \langle N, \lambda\rangle \in \mathcal{T}(Q)$ with $T_0 = q^0$, such that (i) if $N' \subseteq N$ is a branch of T, then there exist infinitely many $w \in N'$ with $\lambda(w) \in F$, and (ii) if $w \in N$, then

$$\{\lambda(wn) : n \in \mathbb{N},\ wn \in N\} \models \delta(\lambda(w), \sigma(|w|))$$

As before, the set of behaviors of A, denoted $L(A)$, is called the *language* of A.

As it turns out, ABAs are exactly as expressive as BAs, as witnessed by the following lemma due to Miyano and Hayashi [41].

Lemma 4. *Let A be an ABA. We can construct a BA A' s.t. $L(A) = L(A')$.*

Thus, to implement $\phi \in \mathcal{L}_\Sigma$ using a BA, all we need to do is implement ϕ using an ABA. To this end, we first observe that the effects of the connectives $\succ$ and $\circledcirc$ on a formula can be mirrored by manipulating ABAs that implement their underlying formulas, as follows.

Lemma 5. *Let $\phi \in \mathcal{L}_\Sigma$, and suppose that we can construct an ABA that implements ϕ. Then we can construct BAs that implement $\succ\phi$ and $\circledcirc\phi$.*

Proof. We argue the claim for $\succ\phi$; the case for $\circledcirc\phi$ can be argued analogously. Suppose A_ϕ is an ABA implementing ϕ; by Lemma 4, we obtain a BA $A'_\phi = \langle Q, \delta, q^0, F\rangle$ implementing ϕ. We choose the BA $A_{\succ\phi} = \langle Q, \delta', q^0, F\rangle$, where

$$\delta'(q, a) = \bigvee \{\delta(q, a') : a' \in \Sigma,\ a' \sqsubseteq a\}$$

We claim that $A_{\succ\phi}$ implements $\succ\phi$. To see this, suppose $\sigma \in L(A_{\succ\phi})$; in that case there exists a $\mu \in Q^\omega$ such that $\mu(0) = q^0$, and for $n \in \mathbb{N}$ it holds that

$$\{\mu(n+1)\} \models \delta'(\mu(n), \sigma(n))$$

This tells us that if $n \in \mathbb{N}$, then there exists an $a_n \in \Sigma$ with $\sigma(n) \sqsubseteq a_n$, and

$$\{\mu(n+1)\} \models \delta(\mu(n), a_n)\}$$

If we now choose $\tau \in \Sigma^\omega$ by setting $\tau(n) = a_n$, we find that $\sigma \sqsubseteq^\omega \tau$ and $\tau \in L(A'_\phi)$. Since A'_ϕ implements ϕ, it follows that $\tau \models \phi$, and thus that $\sigma \models_\Sigma \succ\phi$; the implication in the other direction goes through analogously. □

It should be emphasized that the construction above yields a (non-alternating) BA. This is a consequence of the fact that, to implement $\succ\phi$, we need to translate the ABA that implements ϕ into a BA; if we skip this step, and manipulate the transition structure of (the ABA) A_ϕ along the same lines as above, the resulting ABA does not necessarily implement $\succ\phi$.

Example 4. Suppose that Σ is the CAS used in earlier examples, and consider the ABA with three states $\{q^0, q^1, q^2\}$, with q^1 and q^2 accepting, and δ given by

$$\delta(q,a) = \begin{cases} q^1 \wedge q^2 & q = q^0 \\ q^1 & q = q^1,\ a = \mathsf{move} \\ q^2 & q = q^2,\ a = \mathsf{discharge}_2 \\ \textbf{false} & \text{otherwise} \end{cases}$$

This ABA implements $\phi = X(\Box\,\mathsf{move} \wedge \Box\,\mathsf{discharge}_2)$, and, accordingly, its language is empty. If we apply the transformation used in Lemma 5, we obtain a modified transition function δ', given by

$$\delta'(q,a) = \begin{cases} q^1 \wedge q^2 & q = q^0 \\ q^1 & q = q^1,\ \mathsf{move} \sqsubseteq a \\ q^2 & q = q^2,\ \mathsf{discharge}_2 \sqsubseteq a \\ \textbf{false} & \text{otherwise} \end{cases}$$

This ABA accepts the stream $\sigma = \langle \mathsf{move}_2 \rangle^\omega$, even though $\sigma \not\models_\Sigma \succ\phi$.

We can also implement the negation of a formula ϕ that appears below either $\succ$ or $\circledcirc$, provided that we can implement ϕ itself.

Lemma 6. *Let $\phi \in \mathcal{L}_\Sigma$, and suppose that we can construct an ABA that implements ϕ. Then we can construct ABAs that implement $\neg\succ\phi$ and $\neg\circledcirc\phi$.*

Proof. Like the previous lemma, we argue the case for the formula involving $\succ$. By Lemma 5, we find a BA $A_{\succ\phi}$ that implements $\succ\phi$. Using the techniques pioneered by Kupferman and Vardi [36], we can construct an ABA $A_{\neg\succ\phi}$ with $L(A_{\neg\succ\phi}) = \Sigma^\omega \setminus L(A_{\succ\phi})$; it is easy to see that $A_{\neg\succ\phi}$ implements $\neg\succ\phi$. □

Lemma 7. *Let $\phi \in \mathcal{L}_\Sigma$, and suppose we can implement every subformula of ϕ of the form $\succ\psi$ or $\circledcirc\psi$. We can then construct an ABA A_ϕ that implements ϕ.*

Proof. The main idea is to define a transition structure on the subformulas of ϕ that do not appear below $\succ$ or $\circledcirc$ (and their negations), in accordance with the construction from [51], modified to defer the transitions on formulas of the form $\succ\psi$ or $\circledcirc\psi$ (or their negations) to the automata that implement them.

Let R_ϕ be the set of subformulas of ϕ of the form $\succ\psi$ or $\circledcirc\psi$, and their negations. Let Q_ϕ be set of subformulas of ϕ that do not appear below $\succ$ or $\circledcirc$, and their negations. If $\chi \in R_\phi$, we write $A'_\chi = \langle Q'_\chi, \delta'_\chi, {q^0_\chi}', F'_\chi \rangle$ for the automaton implementing χ (if χ is negated, then A'_χ exists by Lemma 6). We assume (w.l.o.g.) that the Q'_χ are pairwise disjoint, and do not overlap with Q_ϕ.

We choose $A_\phi = \langle Q, \delta, \phi, F \rangle$, where Q and F are given by

$$Q = Q_\phi \cup \bigcup \{ Q'_\chi : \chi \in R_\phi \}$$

$$F = \{ \chi : \chi \in Q_\phi,\ \exists \psi, \rho \text{ s.t. } \chi = \neg(\psi\ U\ \rho) \} \cup \bigcup \{ F'_\chi : \chi \in R_\phi \}$$

We then choose δ by specifying for elements of Q that:

$$\delta(\top, a) = \textbf{true}$$

$$\delta(b,a) = \begin{cases} \textbf{true} & a = b \\ \textbf{false} & \text{otherwise} \end{cases}$$

$$\delta(\psi \wedge \chi, a) = \delta(\psi, a) \wedge \delta(\chi, a)$$

$$\begin{aligned}
\delta(\neg\psi, a) &= \overline{\delta(\psi, a)} && (\text{when } \neg\psi \notin R_\phi) \\
\delta(X\,\psi, a) &= \psi \\
\delta(\psi\, U\, \chi, a) &= \delta(\chi, a) \vee (\delta(\psi, a) \wedge \psi\, U\, \chi) \\
\delta(\chi, a) &= \delta'_\chi(q^{0\,\prime}_\chi, a) && (\text{when } \chi \in R_\phi) \\
\delta(q, a) &= \delta'_\chi(q, a) && (\text{when } q \in Q'_\chi)
\end{aligned}$$

where for $B \in \mathcal{B}(Q_\phi)$ we define $\overline{B}$ inductively, setting

$$\begin{aligned}
\overline{\textbf{true}} &= \textbf{false} \\
\overline{\textbf{false}} &= \textbf{true} \\
\overline{B_0 \vee B_1} &= \overline{B_0} \wedge \overline{B_1} \\
\overline{B_0 \wedge B_1} &= \overline{B_0} \vee \overline{B_1}
\end{aligned}$$

as well as $\overline{\psi} = \chi$ if ψ is of the form $\neg\chi$, and $\overline{\psi} = \neg\psi$ otherwise. The argument from [51, Theorem 22] can now be applied to show that for all $\psi \in Q_\phi$, it holds that σ is accepted starting in ψ if and only if $\sigma \models_\Sigma \psi$; consequently, σ is accepted by A_ϕ (i.e., starting in ϕ) if and only if $\sigma \models_\Sigma \phi$. □

We can now use the lemmas above to arrive at the claimed decidability result.

Theorem 1. *If A is an SCA and $\phi \in \mathcal{L}_\Sigma$, then we can decide whether $A \models_\Sigma \phi$. Furthermore, in case of a negative answer, we obtain a periodic stream $\sigma \in \Sigma^\pi$ such that $\sigma \in L(A)$ but $\sigma \not\models_\Sigma \phi$.*

Proof. By Lemma 3, we can construct a BA A' such that $L(A) = L(A')$. We can then use induction on the nesting depth of $\succ$ and $\circledcirc$ as well as Lemma 5 and Lemma 7 to find an ABA implementing ϕ, which we can convert into a BA A_ϕ implementing ϕ by means of Lemma 4. Deciding whether $A \models_\Sigma \phi$ is then equivalent to deciding whether $L(A') \subseteq L(A_\phi)$, which is possible by Lemma 2; this procedure also provides the periodic counterexample stream. □

Remark 2. Although the complexity of the decision procedure above is hard to analyze in general, it is dominated by the step where a BA implementing ϕ is constructed. More precisely, the conversion of an ABA into a BA may result in a BA that has exponentially more states than the original ABA. In addition to converting the ABA that implements ϕ into a BA, we also need to apply this procedure in Lemma 5. On the other hand, as shown in previous examples, it is possible to verify non-trivial properties of an SCA using formulas that do not nest $\succ$ or $\circledcirc$ more than two times.

We consider an alternative method for verification of SCAs in Section 7.

6. Diagnostics

Having developed a logic for SCAs as well as its decision procedure, we investigate how a designer can cope with undesirable behavior σ exhibited by the agent, either as a run-time behavior, or as a counterexample to a formula found at design-time (obtained through Theorem 1). The tools outlined here can be used by the designer to determine the right threshold value for a component given the properties that the component (or the system at large) should satisfy.

6.1. Eliminating undesired behavior

A simple way to counteract undesired behavior is to see if the threshold can be raised to eliminate it — possibly at the cost of eliminating other behavior. For instance, in Section 5.1, we saw that $A_{\mathsf{e},\mathsf{s}} \not\models_\Sigma \phi_{\mathsf{w}}$, with counterexample $\sigma = \langle \mathsf{move}_2, \mathsf{move}_2, \mathsf{charge}, \mathsf{charge}, \mathsf{charge}, \mathsf{charge}\rangle^\omega$, when $t_\mathsf{e} \otimes t_\mathsf{s} = 7$. Since all move_2-labeled transitions of $A_{\mathsf{e},\mathsf{s}}$ have preference 7, raising[7] $t_\mathsf{e} \otimes t_\mathsf{s}$ to 5 ensures that σ is not present in $L(A_{\mathsf{e},\mathsf{s}})$; indeed, if $t_\mathsf{e} \otimes t_\mathsf{s} = 5$, then $A_{\mathsf{e},\mathsf{s}} \models_\Sigma \phi_\mathsf{w}$. This additional property, however, comes at the cost of eliminating the transitions labeled move_2 with cost 7, i.e., exactly the transitions where the agent makes a move without having taken a snapshot of the current location.

Remark 3. As noted earlier, we should also be careful not to raise the threshold too much, thereby eliminating behavior (in the formal sense of Definition 4) by introducing deadlock states, where no transition is available because all outgoing transitions exceed the threshold. For example, in the extreme case where we choose $t_\mathsf{e} \otimes t_\mathsf{s} = 0$, we find that $L(A_{\mathsf{e},\mathsf{s}}) = \emptyset$,

[7] Recall that $7 \leq_{\mathbb{W}} 5$, so 5 is a “higher” threshold in this context.

since some state of $A_{\mathsf{e},\mathsf{s}}$ is now a deadlock state; consequently, $A_{\mathsf{e},\mathsf{s}} \models_\Sigma \psi$ holds for *any* ψ. In the case where we choose $t_\mathsf{e} \otimes t_\mathsf{s} = 5$, this situation is prevented: all states of $A_{\mathsf{e},\mathsf{s}}$ continue to have available outgoing transitions.

In general, since raising the threshold does not add new behavior, this does not risk adding additional undesired behavior. The only downside to raising the threshold is that it possibly eliminates desirable behavior, and if taken too far, it may introduce deadlock states to the system.

We define the *diagnostic preference* of a behavior as a tool for determining a threshold that rules out a given behavior, as follows.

Definition 7. Let $A = \langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t\rangle$ be an SCA, and let $\sigma \in \Sigma^\pi \cup \Sigma^*$. The *diagnostic preference* of σ in A, denoted $d_A(\sigma)$, is calculated as follows:

1. Let Q_0 be $\{q^0\}$, and for $n < |\sigma|$ set $Q_{n+1} = \{q' : q \in Q_n,\ q \xrightarrow{\sigma(n),\ e} q'\}$.
2. Let $\xi \in \mathbb{E}^\pi \cup \mathbb{E}^*$ be the stream s.t. $\xi(n) = \bigoplus\{e : q \in Q_n,\ q \xrightarrow{\sigma(n),\ e} q'\}$.
3. $d_A(\sigma) = \bigwedge\{\xi(n) : n \leq |\sigma|\}$ (recall that $\bigwedge$ is the greatest lower bound).

Since σ is finite or eventually periodic, and Q is finite, ξ is also finite or eventually periodic. Consequently, $d_A(\sigma)$ is computable.

Lemma 8. *Let $A = \langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t\rangle$ be an SCA, and let $\sigma \in \Sigma^\pi \cup \Sigma^*$. If $\sigma \in L(A)$, or σ is a finite prefix of some $\tau \in L(A)$, then $t \leq_\mathbb{E} d_A(\sigma)$.*

Proof. If $\sigma \in L(A)$, there exist streams $\mu \in Q^\omega$ and $\nu \in \mathbb{E}^\omega$ such that $\mu(n) = q^0$, and for all $n \in \mathbb{N}$, $t \leq \nu(n)$ and $\mu(n) \xrightarrow{\sigma(n),\ \nu(n)} \mu(n+1)$. It is not hard to see that $\mu(n) \in Q_n$ for $n \in \mathbb{N}$. Then also $t \leq_\mathbb{E} \nu(n) \leq_\mathbb{E} \xi(n)$ for all $n \in \mathbb{N}$. Thus, $t \leq_\mathbb{E} d_A(\sigma)$. Likewise, if σ is a finite prefix of some $\tau \in L(A)$, then $t \leq_\mathbb{E} d_A(\tau)$ by the above, and $d_A(\tau) \leq_\mathbb{E} d_A(\sigma)$ by definition of d_A, thus $t \leq_\mathbb{E} d_A(\sigma)$. □

Since $d_A(\sigma)$ is a necessary upper bound on t when σ is a behavior of A, it follows that we can exclude σ from $L(A)$ if we choose t such that $t \not\leq_\mathbb{E} d_A(\sigma)$. In particular, if we choose t such that $d_A(\sigma) <_\mathbb{E} t$, then $\sigma \notin L(A)$. Note that this may not always be possible: if $d_A(\sigma)$ is $\mathbf{1}$ then such a t does not exist.

Note that there may be another threshold (i.e., not obtained by Lemma 8), which may also eliminate fewer desirable behaviors. Thus, while this lemma gives helps to choose a threshold to exclude some behaviors, it is not a definitive guide. We refer to [31] for a concrete example.

6.2. Localizing undesired behavior

One can also use the diagnostic preference to identify the components that are involved in allowing undesired behavior. Let us revisit the first example from Section 5.1, where we verified that every pair of move-actions was separated by at least one snapshot action, as described in ϕ_w. Suppose we choose $t_\mathsf{e} = 10$ and $t_\mathsf{s} = 1$; then $t_\mathsf{e} \otimes t_\mathsf{s} = 11$, thus $\sigma = \langle \mathsf{move}_2, \mathsf{charge}, \mathsf{charge}\rangle^\omega \in L(A_{\mathsf{e},\mathsf{s}})$, meaning $A_{\mathsf{e},\mathsf{s}} \not\models_\Sigma \phi_\mathsf{w}$. By Lemma 8, we find that $11 = t_{\mathsf{e},\mathsf{s}} = t_\mathsf{e} \otimes t_\mathsf{s} \leq_\mathbb{W} d_{A_{\mathsf{e},\mathsf{s}}}(\sigma) = 7$. Even if A_s's threshold were as strict as possible (i.e., $t_\mathsf{s} = 0 = \mathbf{1}_\mathbb{W}$), we would find that $t_\mathsf{e} \otimes t_\mathsf{s} \leq_\mathbb{W} d_{A_{\mathsf{e},\mathsf{s}}}(\sigma)$, meaning that we cannot eliminate σ by changing t_s only. This finding tells us that the current setting of $t_\mathsf{s} = 1$ is already high enough to prevent all behaviors of A_s that can be filtered out, from contributing to the undesired behavior σ in $A_{\mathsf{e},\mathsf{s}}$. On the other hand, raising t_e does eliminate σ, and in this sense, we may say that A_s is responsible for σ.[8]

More generally, let $(A_i)_{i\in I}$ be a finite family of automata over the c-semiring $\mathbb{E}$ with thresholds $(t_i)_{i\in I}$. Furthermore, let $A = \bowtie_{i\in I} A_i$ and let ψ be such that $A \not\models_\Sigma \psi$, with counterexample behavior σ. Suppose now that for some $J \subseteq I$, we have $\bigotimes_{i\in J} t_i \leq_\mathbb{E} d_A(\sigma)$. Since $\otimes$ is intensive, we furthermore know that $\bigotimes_{i\in I} t_i \leq_\mathbb{E} \bigotimes_{i\in J} t_i$. Therefore, at least one of t_i for $i \in J$ must be adjusted to exclude the behavior σ from the language of $\bowtie_{i\in I} A_i$.

We call $(t_i)_{i\in J}$ *suspect* thresholds: *some* t_i for $i \in I$ must be adjusted to eliminate σ; by extension, we refer to J as a *suspect subset* of I. Note that I may have distinct and disjoint suspect subsets. If $J \subseteq I$ is disjoint from every suspect subset of I, then J is called *innocuous*. If J is innocuous, changing t_j for some $j \in J$ (or even t_j for all $j \in J$) alone does not exclude σ. Finding suspect and innocuous subsets of I thus helps in finding out which thresholds need to change in order to exclude a specific undesired behavior.

Algorithm 1 gives pseudocode to find minimal suspect subsets of a suspect set I; we argue correctness of this algorithm in Theorem 2.

[8] We could argue that A_s alone is not responsible for σ in $A_{\mathsf{e},\mathsf{s}}$ either, because modifying the preference of the move-loop on q_N in A_s can help to exclude the undesired behavior as well. In our framework, however, the threshold is a generic property of any SCA, and so we use it as a handle for talking about localizing undesired behaviors to component SCAs.

```
Function FindSuspect($I$):
    $M := \emptyset$;
    foreach $i \in I$ do
        if $I \setminus \{i\}$ is suspect then
            $M := M \cup$ FindSuspect($I \setminus \{i\}$);
        end
    end
    if $M = \emptyset$ then
        return $\{I\}$;
    else
        return $M$;
    end
end
```

Algorithm 1: Algorithm to find minimal suspect subsets.

Theorem 2. *If I is suspect and $d_A(\sigma) < \mathbf{1}$, then* `FindSuspect`(I) *contains exactly the minimal suspect subsets of I.*

Proof. First, note that it is easy to see that `FindSuspect` never returns $\emptyset$.

The proof proceeds by induction on I. In the base, where $I = \{i\}$, we can see that $\bigotimes \emptyset = \mathbf{1}$, thus, since $d_A(\sigma) < \mathbf{1}$, it follows that $I \setminus \{i\} = \emptyset$ is not suspect. The first branch of the subsequent **if** is selected, which returns $\{I\}$ itself. This matches the fact that I is the only suspect subset of I.

In the inductive step, we assume the claim holds for all strict subsets of I. We consider two cases. On the one hand, if there exists an $i \in I$ such that $I \setminus \{i\}$ is suspect, then we know that the **foreach**-loop will modify M (since `FindSuspect` never returns an empty set). Moreover, I itself is not minimally suspect. The algorithm then returns

$$\bigcup \{\texttt{FindSuspect}(I \setminus \{i\}) : i \in I,\ I \setminus \{i\} \text{ suspect}\}$$

By induction, `FindSuspect`$(I \setminus \{i\})$ returns all minimal suspect subsets of $I \setminus \{i\}$. Since each of these is also a minimal suspect subset of I, and since very minimal suspect subset of I that is not equal to I is contained in one of these, the claim follows by the fact that we ruled out I as a minimal suspect subset. □

In the case where $d_A(\sigma) = \mathbf{1}$, it is easy to see that $\{\{i\} : i \in I\}$ is the set of minimal suspect subsets of I.

In the worst case, every subset of I is suspect, and therefore the only minimal suspect subsets are the singletons; in this scenario, there are $O(|I|!)$ calculations of a composed threshold value. Using memoization to store the minimal suspect subsets of every $J \subseteq I$, the complexity can be reduced to $O(2^{|I|})$.

While this complexity makes the algorithm seem impractical (I need not be a small set), we note that the case where all components individually fail to filter out a contribution to a certain undesired behavior should be exceedingly rare in a system that was designed with the violated concern in mind: it would mean that *every component* contains behavior that ultimately composes into the undesired behavior — in a sense, every component facilitates a behavior that counteracts their collective interest.

7. Implementation

In this section, we propose an implementation of our soft component automata model in order to perform design and verification experiments. For this purpose, we use Reo [1], a language to describe coordination of systems of components. One reason for using Reo is its graphical syntax, which gives an intuitive encoding of soft component automata in terms of graphical components and interaction primitives. Moreover, Reo reflects the modular and compositional aspects that make SCAs suitable for specifying complex behaviors: connectors compose into more complex connectors, just like how SCAs compose into more complex SCAs. We take advantage of this feature and, after defining an encoding of SCAs into Reo connectors, we represent the composition of SCAs as the composition of their corresponding connectors. Another reason is the existence of a compilation chain that makes it possible to compile the same Reo model to an execution language (such as Java or C) or to a language that supports verification (such as the rewriting logic language Maude [13]). Effective optimizations implemented in the current Reo compiler help to keep the size of resulting composed models manageable, yielding similarly manageable models in Maude, Java, etc.

We begin with a brief introduction to Reo in Section 7.1, before presenting our encoding of SCA as a Reo connector in Section 7.2. The encoding is manual, but should be straightforward from the close resemblance with the structure of an automaton. We then describe the internal representation of Reo connectors used in the Reo compiler, and show in Section 7.3 how this internal representation can naturally be translated to rewrite rules, e.g., in Maude. The generation of a Maude rewrite system from a Reo connector is completely automated. The code to reproduce the experiments is publicly accessible at [38]. Finally, we illustrate in Section 7.4 the utility of the Maude representation produced by our Reo compiler for verification.

7.1. Reo as candidate

Reo is a language used for the design of coordination protocols. On the one hand, Reo is characterized by a graphical representation of connectors, which gives an intuitive understanding of the data flowing in a Reo circuit. On the other hand, interaction primitives in Reo have a formal compositional semantics, which enable verification of connectors, built out of the composition of primitives. We first introduce the graphical representation of Reo connectors, and then provide a compositional semantics for interaction primitives as a fragment of first order logic.

Reo connectors are built using two main constructs: *components* and *ports*. A *port* supports synchronous transfer of data. Conceptually, each port has two sides, input and output, each of which accommodates a single type of operation: *put* or *get*. Performing a put operation on a port involves the output side of that port: the port is used as an output port in this context. Symmetrically, performing a get operation on a port involves the input side of that port: the port is used as an input port in this context. A port *fires* whenever a pair of put and get operations on that port execute atomically. We define a *component* as a set of ports together with a relation that constrains data flow among those ports. A component is either *atomic* or *composite*. Only the former type of components must have a semantics attached. A composite component inherits its formal semantics from the composition of the semantics of its parties. From the point of view of a component, a port is either a boundary or an internal port. A boundary port of a component is shared by two components: one uses the port as an output port, and the other as an input port. Internal ports are not accessible by other components. We also refer to the set of boundary ports of a component as the *interface* of that component.

Observe that the environment acts as a complement to a component: a boundary port used as an input port by the component is an output port for the environment, and vice versa. Including the environment as a component makes all ports internal and yields a closed system. If some boundary ports are left opened, i.e., not shared with an environment component, then the component is exposed to arbitrary patterns of external put and get operations on those boundary ports.

Graphical syntax. Reo has a graphical syntax that visualizes composition of components. Typically, we call a binary component a *channel* and certain kinds of n-ary components *nodes*. Fig. 5 shows four different channels, one ternary component, and two nodes. Later, the components in Fig. 5 will be used to encode SCAs as Reo connectors. In this section, we describe the behavior of some channels and some nodes only intuitively.

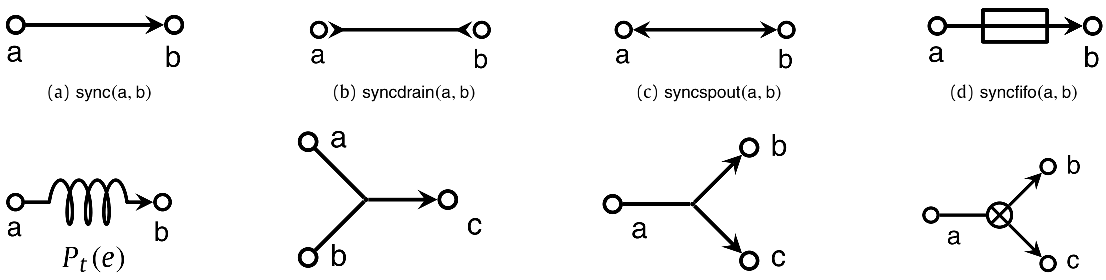
First, we consider three synchronous channels. The sync channel in Fig. 5a represents a synchronous transfer of data from port a to port b. The $\mathsf{syncdrain}$ and $\mathsf{syncspout}$ channels in Fig. 5b and Fig. 5c model a synchronous firing of port a and b without necessarily equating data at those ports; the difference is that in the $\mathsf{syncdrain}$, a and b are input words, whereas in the $\mathsf{syncspout}$ they are output ports.

A $\mathsf{syncfifo}$ channel (depicted in Fig. 5d) is a combination of synchronous and asynchronous behavior. It has an internal buffer with the capacity to hold one data item. This buffer is initially empty. When its buffer is empty, a $\mathsf{syncfifo}$ channel accepts a data item through its input port a, places it in its buffer, which then becomes full. When the buffer of a $\mathsf{syncfifo}$ channel is full, the channel delivers the content of its buffer to a get operation performed by the environment on its output port b, and its buffer becomes empty. A get operation on the output port of a $\mathsf{syncfifo}$ channel with an empty buffer blocks until after its buffer becomes full. In addition to those two operations, and in contrast with a standard fifo channel, a $\mathsf{syncfifo}$ can also synchronously empty its buffer to its output port while taking a datum at its input port. It turns out that a $\mathsf{syncfifo}$ channel can itself be constructed as a Reo circuit, and thus its graphical representation can be considered as a mere abbreviation. The details of the construction of the Reo circuit for a $\mathsf{syncfifo}$ is beyond the scope of this paper, and can be found in [4].

The $\mathsf{bfilter}$ channel in Fig. 5e is a blocking-filter channel, and is context sensitive, in the sense that its behavior depends on its composition. By itself, as pictured on Fig. 5e, the $\mathsf{bfilter}$ synchronously passes the data from a to b if, and only if, the c-semiring value e is above the threshold t. In composition with other $\mathsf{bfilter}$ channels, the channel compares the composite c-semiring value with the composite threshold resulting from the composition. As examples of nodes, the composition of two $\mathsf{bfilter}$ channels $\mathsf{bfilter}\langle \mathsf{e}_1, \mathsf{t}_1 \rangle(\mathsf{a,b})$ and $\mathsf{bfilter}\langle \mathsf{e}_2, \mathsf{t}_2 \rangle(\mathsf{b,c})$ changes the behavior of each $\mathsf{bfilter}$ to respectively $\mathsf{bfilter}\langle \mathsf{e}_1 \otimes \mathsf{e}_2, \mathsf{t}_1 \otimes \mathsf{t}_2 \rangle(\mathsf{a,b})$ and $\mathsf{bfilter}\langle \mathsf{e}_1 \otimes \mathsf{e}_2, \mathsf{t}_1 \otimes \mathsf{t}_2 \rangle(\mathsf{b,c})$, where $\otimes$ is the product in the c-semiring.

As an example of nodes, the merger and the $\mathsf{replicator}$ are represented in Fig. 5f and Fig. 5g respectively. A merger synchronously transfers data from at most one of its input boundary ports to its output boundary port. If data is available at both input ports, a merger non-deterministically chooses one to synchronize with its output port. The $\mathsf{replicator}$ synchronously duplicates the data received at its input port to both of its output ports. The merger and $\mathsf{replicator}$ are called nodes, because they graphically and textually allow an abbreviation for the construction of Reo connectors: a port used multiple times as output port, respectively input port, is substituted by a merger, respectively a replicator, with the appropriate port renaming.

Finally, the ternary component $\mathsf{xrouter}$ depicted in Fig. 5h has the behavior of an *exclusive router*. The $\mathsf{xrouter}$ synchronously transmits the data item that it obtains from its input end through only one of its output ends, selected nondeterministically. The $\mathsf{xrouter}$ component can be generalized to an n-output component. Similarly to the $\mathsf{syncfifo}$, the $\mathsf{xrouter}$ can itself be constructed out of more primitive connectors [5]. For our purposes, it suffices to consider the components in Fig. 5 as yet another set of atomic primitives.



(a) sync(a, b) (b) syncdrain(a, b) (c) syncspout(a, b) (d) syncfifo(a, b)

(e) bfilter<t,e>(a,b) (f) merger(a, b, c) (g) replicator(a, b, c) (h) xrouter(a, b, c)

Fig. 5. Graphical syntax for some primitives.

Besides the graphical syntax used to picture Reo components, we provide a textual definition for the connectors introduced later, using Treo syntax [16]. In Treo, connectors have a unique name, and are defined providing a list of parameters, an interface, and a body. We specify a list of parameters in angular brackets ("$<$" and "$>$"), and an interface as a list of ports in parentheses. Parameters and ports in the signature of a connector are exposed to external components. The body of a connector defines its structure, and is written as the set of its sub-connectors. We use set-builder notation for this purpose. Conditions for sets are written as predicates. As a built-in feature of the textual language, there is no need to specify nodes (mergers, or replicators). The compiler infers and automatically inserts mergers and replicators in the presence of ports used multiple times as input or output. An example of a component expressed in textual Reo is given in Listing 1.

Compilation. Reo connectors are built compositionally out of atomic primitives. The compiler takes as input a Reo connector specified using the Treo syntax, and generates a program in one of the target languages. The generated program, together with a runtime environment, conforms to the behavior of the Reo connectors. We separate the compilation steps into three phases.

The first step consists of enumerating all atomic elements of the Reo connector. The hierarchical structure of connectors is unfolded, and all atomic components are enumerated and correctly instantiated. Nodes are inserted as atomic components. At the end of this step, the connector consists of a set of atomic elements.

The second step defines the behavior of the connector as the composition of the behavior of atomic elements. Since the composition operator is associative and commutative, the compiler can rearrange the composition algebraically, to reduce its complexity. The choice of the semantics representation for the atomics impacts the internal representation of the connector in the compiler. Some semantics for Reo components, such as constraint automata, suffer from a state space explosion, and several optimizations have been studied [28]. In this work, we use a logical semantics (presented in Section 7.3) to define the behavior of atomic components, which, under certain conditions, keeps the composite representation linear [15].

Finally, the compiler uses the logical description of the composite connector together with the input and output information of each port, to produce a set of guarded commands. A backend specific to the target language can subsequently translate these guarded commands into a target language for execution (e.g., Java, C) or prototyping and verification (e.g., Maude).

7.2. Reo representation of SCA

Some existing research has considered the question of synthesizing Reo circuits for constraint automata [4]. In our work, similar channels are used for encoding the structure of the automaton (syncfifo, xrouter, and merger), but a new channel, the bfilter, is introduced to encode the soft part of the action labeling transitions of SCAs. Moreover, we provide, along with the description, the representation of the Reo connector in a textual language, used as input for our compiler.

We propose a general approach to represent SCAs and their composition as Reo circuits. Recall that, by Definition 2, an SCA is formally defined as a tuple $\langle Q, \Sigma, \mathbb{E}, \rightarrow, q^0, t\rangle$ where Q is the set of states, Σ a component action system, $\mathbb{E}$ a c-semiring, $\rightarrow$ a transition relation, $q^0 \in Q$ the initial state, and $t \in \mathbb{E}$ is the threshold value of the SCA. In the sequel, we give a procedure to write an SCA as a Reo circuit. The set of connectors defined hereafter constitutes a domain specific fragment of Reo for building SCA. We conclude this section with an example of composition of two SCAs obtained through composition of their respective Reo representations.

Actions and c-semirings. Given Σ a CAS of an SCA, we map each action $a \in \Sigma$ into a Reo port with the same name. We consider the SCA "doing action a" equivalent to "firing of port a". Given the threshold $t \in \mathbb{E}$, we associate each c-semiring value $e \in \mathbb{E}$ with a predicate $P_t(e)$ whose semantics reflects the truth value of $t \leq e$ in the c-semiring. In order to mirror the semantics defined previously for composition of SCA, the c-semiring value and the threshold value of a predicate may change during composition. We consider the c-semiring to be fixed and shared by all SCAs.

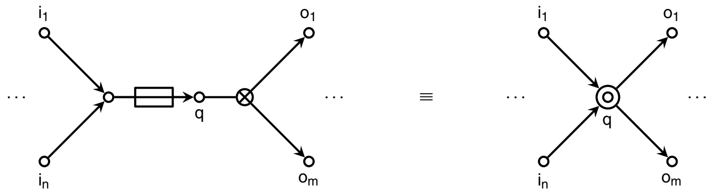


Fig. 6. Graphical abbreviation for a state.

```
State(qi[1..n],qo[1..m]) {
  { sync(qi[k],x)  |  k:<1..n> }
  syncfifo1<"0">(x,y)
  xrouter(y,qo[1..m])
}
```

Listing 1: Component defining a state q in textual Reo.

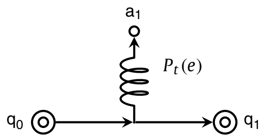


Fig. 7. Reo circuit for a transition of a soft component automaton.

States. We define a state of an SCA as a Reo circuit, which we then graphically abbreviate as a user-defined node (Fig. 6). Essentially, a state is mapped into a syncfifo channel, the empty/full status of whose buffer reflects whether or not the SCA is currently in that state. As depicted in the circuit below, we identify the source end of the syncfifo with the name of the state. Thus, to be in state q of the SCA corresponds to the syncfifo whose source end is q being full. The initial state q^0 starts with a full syncfifo buffer; the syncfifo buffers of all other states start empty. Intuitively, all incoming $(\mathsf{i}_1, \dots, \mathsf{i}_n)$ transitions into a state q, merge at the source end of the syncfifo, and all outgoing transitions $(\mathsf{o}_1, \dots, \mathsf{o}_m)$ out of q synchronize via mutual exclusion with one another on the sink end of the syncfifo. The reason for using the syncfifo instead of the standard fifo primitive is that an outgoing transition can also be an incoming transition into the same state, i.e., allow get and put operations on its ends to synchronously empty and fill its buffer. We use an n-ary *exclusive router* to express that only one outgoing transition is taken from a state with n outgoing transitions. The n-ary xrouter can be constructed out of the ternary xrouter of Fig. 5h.

We call our constructed circuit a *state*, and use ◎ to represent a state of an SCA as a graphical abbreviation and present it as a user-defined node in Reo with n inputs and m outputs. We use ⦿ as graphical abbreviation for the Reo circuit that corresponds to the initial (and current) state of an SCA.

Besides the graphical construct for a state, we introduce a State connector in the textual language of Reo shown in Listing 1. We adopt a convention, and prefix the input and output ports of a state with the name of the state. For instance, the component State(q0i[1..n], q0o[1..m]) represents the state q^0 with n incoming transitions and m outgoing transitions. We refer to the k-th incoming, resp. k-th outgoing, transition to state q0 with the port q0i[k], respectively q0o[k].

Listing 1 shows an example of a component defined using conditional set notation. The number of input ports in the interface of the State component influences how its body is instantiated. The variable k ranges over the list $[1, .., n]$, and thus creates a set of sync channels.

Transitions. A transition in an SCA involves an action, a c-semiring value, a pre-state and a post-state. When the transition is enabled (i.e., its c-semiring value is above the threshold), the transition synchronously fires the action port, and moves the SCA from its pre-state to its post-state. We model this behavior in Reo as the circuit in Fig. 7, which represents the conditional activation of a transition using a blocking-filter channel that compares the c-semiring value of the transition with the threshold of the SCA. Given a c-semiring value e, the predicate $P_t(e)$ of the blocking-filter channel is true if and only if the c-semiring value e is greater than or equal to the threshold value t.

The circuit in Fig. 7 moves the token from node q_0 to node q_1 and fires port a_1, only if $P_t(e)$ is true. If $P_t(e)$ is not satisfied, the circuit in Fig. 7 blocks the transfer of the token from q_0 to q_1, mirroring the fact that its corresponding SCA transition cannot be taken.

The transition primitive in textual Reo is written in Listing 2. The transition component takes three ports in its interface, q_0 and q_1, being respectively the pre-state and post-state, and a_1 being the action. Two values are provided as parameters to a transition component: the c-semiring value e, and the threshold value t. Internally, the transition component connects the pre-state to the post-state through synchronization with the bfilter. The bfilter takes a c-semiring value of a given type as parameter, and performs internal comparison with the threshold value.

```
Transition<e,t>(q0,q1,a1) {
  sync(q0,x)
  bfilter<e,t>(x,a)
  sync(x,q1)
}
```

Listing 2: Component defining a transition in textual Reo.

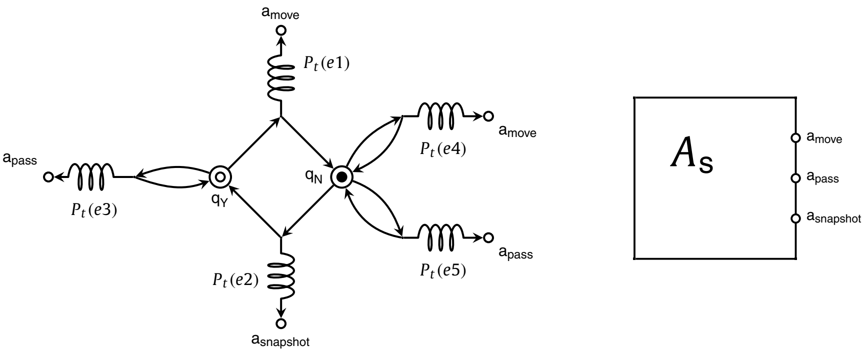


Fig. 8. Reo circuit for the Snapshot SCA.

```
As<t>(move,pass,snap) {
  Transition<1,t>(qYo[1],qYi[1],pass)
  Transition<0,t>(qYo[2],qNi[1],move)
  Transition<0,t>(qNo[1],qYi[2],snap)
  Transition<2,t>(qNo[2],qNi[2],move)
  Transition<1,t>(qNo[3],qNi[3],pass)

  State(qYi[1..2],qYo[1..2])    //State qY
  State(qNi[1..3],qNo[1..3])      //State qN
}
```

Listing 3: Component defining the snapshot SCA in textual Reo.

Soft component automata. Given the constructs for states and transitions, we can build a Reo circuit for every SCA. For instance, the circuit for the automaton in Fig. 2 is shown in Fig. 8. The two states q_Y and q_N are represented as two state-nodes, with q_N initially full (designating it as the initial state).

To avoid visual clutter, we repeat the names of ports in the circuit (e.g., a_{move} appears twice in Fig. 6), but all occurrences of the same port name correspond to a single, unique port. Each of the five transitions of A_s is an instance of the transition component in Reo. For example, the move transition from q_Y to q_N is represented by the transition connector with input from the state q_Y, output from the state q_N, blocking filter with predicate $P_t(e_1)$, and action port a_{move}. The corresponding component view of the automaton is represented by a box that abstracts away the details of its Reo circuit, exposing as its interface the boundary ports on which other components can synchronize.

The snapshot SCA A_e is built out of the State and Transition connectors in Reo defined in Listings 1 and 2. We show the instance of the Snapshot SCA A_s in Listing 3, and adopt the convention defined previously to denote ports of incoming and outgoing transitions.

Component action system. The composition of two SCAs can also be written as a Reo circuit, by encoding the composed SCA. However, such an approach uses the SCA composition and disregards the compositional nature of Reo. Instead, we propose to encode each individual SCA as a Reo circuit, and then compose those encodings on the level of Reo, to obtain a Reo circuit equivalent to their composed automaton. This approach allows for a transparent and incremental translation.

Since composition on the level of SCAs is mediated by their (common) CAS, composition at the level of Reo should also take the CAS into account. To do this, we encode the CAS as a Reo circuit of its own; composition of two automata at the level of Reo is then given by the (Reo) composition of their individual encodings, together with the circuit obtained from their CAS. Furthermore, we hide all ports that are not output ports of the CAS after the composition, so that the only actions observable in the resulting Reo circuit are the actions that are brokered between the operand circuits by the CAS.

There are three “sides” (collections of ports) to a CAS component: one for each of the two operands in the composition, respectively called the *left* and the *right (operand) side*, and a *composite side* for the result of the composition. For each

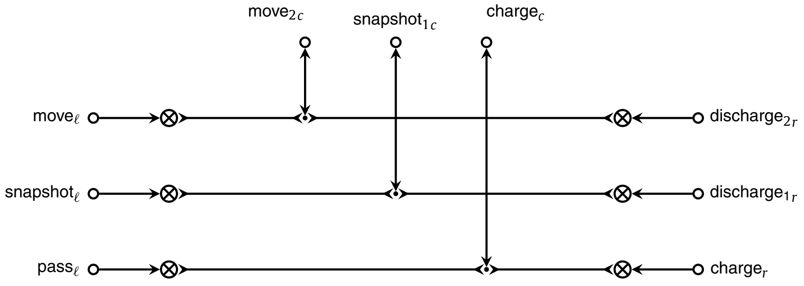


Fig. 9. Partial encoding of a CAS.

```
cas(move,pass,snap,dchge1,dchge2,
    chge,move2,charge,snapshot1){
  syncdrain(move,x)  syncspout(x,move2)
  syncdrain(dchge2,x)
  syncdrain(pass,y) syncspout(y,charge)
  syncdrain(chge,y)
  syncdrain(snap,z) syncspout(z,snapshot1)
  syncdrain(dchge1,z)
}
```

Listing 4: Component defining the CAS for the composition of A_e and A_s in textual Reo.

action α, we add three ports to the circuit, one in each side, labeled α_ℓ, α_r and α_c for the left, right and composite sides respectively. The ports on the operand sides are input ports, and the ports on the composite side are output ports.

The intention of the circuit structure is as follows. If the operand circuits are ready to perform actions α and β respectively, then ports α_ℓ and β_r will be enabled for writing. If $\alpha \circledcirc \beta$, then the CAS circuit brokers their composition, by allowing α_ℓ and β_r to fire simultaneously, synchronously firing the port that represents their composition in the composite side, i.e., $(\alpha \boxdot \beta)_c$, as well. Moreover, the circuit ensures that firing two ports in the left and right sides (when permitted) gives rise to exactly one port firing in the composite side.

More formally, the circuit is built as follows. On the operand sides, each port α_o (where $o \in \{\ell, r\}$) is connected to an exclusive router labeled α_o^R. For each pair of actions in the left and right operand sides that are compatible, i.e., all $\alpha, \beta \in \Sigma$ such that $\alpha \circledcirc \beta$, we draw a synchronous drain from α_ℓ^R and β_r^R to an internal node labeled $\alpha\beta$. Each of these nodes is then connected through a syncspout channel to the composite side node labeled $(\alpha \boxdot \beta)_c$.

The CAS defined for the SCAs A_e and A_s is depicted in Fig. 9. In this example, the exclusive router has a single output, and is not strictly necessary. In general, the CAS could define multiple composite actions out of the same side action. For instance, suppose that the drone in our example is equipped with solar panels, and that the net result of charging using the solar panels while moving is that the energy level does not change. As a result, the energy component's action pass is compatible with the action move, and their composition is the action solar, which means "move with energy from the solar panels". Note how in this scenario, the firing of move_ℓ can occur only in conjunction with firing discharge_{2r} or pass_r, but not both; in the first case, the composite interface port move_{2c} fires, while in the second case the port solar_c fires.

We give in Listing 4 the corresponding Reo component for the CAS described in Fig. 9 for the composition of the snapshot SCA and the energy SCA. We omitted the exclusive routers, since, in this case, they are not necessary.

Composition. The Reo circuit corresponding to a composition of two soft component automata can now be defined as the composition of the Reo circuits for the individual soft component automata, together with the Reo circuit for the relevant component action system (Fig. 10). Following the method above, we translate each of A_s and A_e, respectively representing the snapshot component and the energy management component, into its respective Reo connector.

Based on the steps described above, it is now possible to define a Reo circuit for both A_e and A_s, that we name respectively `Ae` and `As` in textual Reo. The resulting composition, shown in Listing 5, consists of a set containing the connector for each SCA together with the connector for the component action system. The two thresholds values are provided as parameter.

Note that, in this case, the composite component exposes only the composite action of both automata. For verification purposes, we also expose the internal actions of each SCA. We later use internal actions in the encoding of some LTL properties as Maude queries, as in Listing 7.

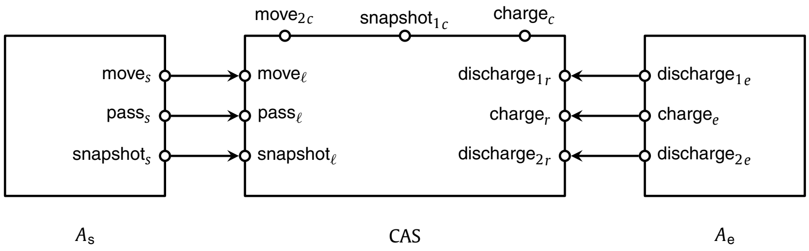


Fig. 10. Composition of two component automata with their component action system.

```
composite(move2,charge,snapshot1) {
  Ae<t1>(move,pass,snap)
  cas(move,pass,snap,dchge1,dchge2,
      chge,move2,charge,snapshot1)
  As<t2>(dchge1,dchge2,chge)
|
  t1 = 5,
  t2 = 3
}
```

Listing 5: Component in textual Reo defining the composition between A_e and A_s.

7.3. Reo to Maude

In Reo, the behavior of a connector is defined in terms of the behavior of the atomics. Several semantics have been defined to represent the behavior of atomic components [27]. We use a logic as semantics for atomics, similar to [12]. We introduce the logic and the operation to compose primitives in the next subsection. The compiler builds internally the logical representation of the connectors. Using the direction of the boundary ports of the connectors, the compiler infers from the logical representation a set of guarded commands, from which a translation to a backend language is defined. We present a translation from the set of guarded commands to a set of rewrite rules written in Maude.

Semantics for atomics. We use a logical semantics to specify the behavior of Reo connectors. In this semantics, connectors are formulas in a fragment of first order logic, and composition of connectors is logically specified as taking the conjunction of their respective formulas.

We first introduce some notation. We use $\mathcal{P}$ to denote the set of variables that represent values exchanged through their homonym ports, M the set of variables that represent the current values stored in their homonym memory cells, and M' the set of variables that represent the next values to be stored in their unprimed homonym memory cells. We use $D = \{1, *\}$ to denote the domain of port and memory variables. The symbol $*$ will be used to encode the non-firing of ports as an equality. Lastly, recall that given a c-semiring value $e \in E$ and a threshold $t \in E$, we denote the inequality $t \leq e$ by $P_t(e)$.

A *term* is either a port variable $p \in \mathcal{P}$, a variable $m \in M$ for current value of a memory cell (as the one involved in the syncfifo, cf. Fig. 5d), a variable $m' \in M'$ for the next value of a memory cell, or a constant: an element of $\{*, 1\}$.

A *formula* is built inductively from terms and the Boolean symbol $\bot$ (false) by the grammar:

$$\phi \quad ::= \quad \bot \quad | \quad t_1 = t_2 \quad | \quad t_1 \neq t_2 \quad | \quad P_t(e) \quad | \quad \phi_1 \wedge \phi_2 \quad | \quad \phi_1 \vee \phi_2$$

Given a port $p \in \mathcal{P}$, the proposition of whether or not p fires is encoded as an equality between p and elements of $\{1, *\}$. If $p = 1$, we say that p fires. On the other hand, $p = *$ means that p does not fire. The inequality $p \neq *$ reduces, in our case, to $p = 1$. Taking $a, b \in \mathcal{P}$, we can now express that a and b fire synchronously with the same datum, as the formula $a = b$.

In the sequel, we assume all formulas for primitive components to be in disjunctive normal form. A *clause* refers to a disjunct in a formula in disjunctive normal form. Fig. 11 shows the formulas that encode the semantics of a few Reo primitives. The formula $a = b$ describes the behavior of a sync channel.

The formula for a syncfifo channel has four clauses. The first one corresponds to filling the buffer with the data observed at port a; the second one empties the buffer through port b; the third one simultaneously empties the buffer through port b, and fills the buffer through port a; and the last one corresponds to the case where no ports fire, in which case the value in the buffer must remain unchanged.

The formula for a merger primitive has three clauses. The first two clauses represent the cases where data flows respectively from port a to port c or port b to port c. The last clause expresses the situation where no port fires. Although the logical disjunction suggests that multiple clauses may be true at the same time, the merger shows an example of exclusive clauses: port a and port b cannot fire at the same time, because $a \neq * \wedge b = *$ and its symmetric term exclude such a possibility.

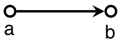
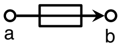
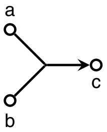
sync(a, b)		$a = b$
syncfifo(a, b)		$(m' = a \wedge a \neq * \wedge b = * \wedge m = b) \vee$ $(m' = a \wedge a = * \wedge b \neq * \wedge m = b) \vee$ $(m' = a \wedge a \neq * \wedge b \neq * \wedge m = b) \vee$ $(m' = m \wedge a = * \wedge b = *)$
merger(a, b, c)		$(a = c \wedge a \neq * \wedge b = *) \vee$ $(b = c \wedge b \neq * \wedge a = *) \vee$ $(a = c \wedge a = * \wedge b = c)$

Fig. 11. From left to right: textual syntax, graphical syntax and logical syntax. From up to bottom: sync channel, syncfifo channel, merger component.

Each component is now described by a formula over its ports and memory variables. The composition of two components is represented by the conjunction of their respective formulas. Internally, the compiler normalizes the composite formula, and gets back a set of synchronous clauses. The notion of synchronous clause is left intuitive here, and can be found with more details in [15]. Essentially, the compiler uses the information present in each clause, such as whether a port must fire or not, to distribute the formula efficiently. In most of cases, the case of quadratic state space explosion present in the constraint automaton distribution is avoided.

To keep the same semantics as soft component automata composition, we introduce a non standard interpretation of conjunction of c-semiring predicates: the conjunction of two c-semiring predicate is a new c-semiring predicate on the composed value and composed threshold. We make this interpretation more precise in the construction of the guarded command from the formula.

Guarded commands. One additional piece of information necessary for implementation is the direction of data flow in the circuit, i.e., whether a port is an input or an output port. The logic intentionally abstracts away the direction of ports to describe only data constraints among port and memory variables. We assume that each port occurring in a formula is orthogonally designated as an input or an output port. Based on the flow direction of each port, we transform the formula describing a connector into a set of guarded commands.

A *guard* is a set of predicates on the variables involved in a clause. A *command* is a series of assignment instructions to the variables involved in a clause. Each clause defines a set of *guards* and *commands*, referred to as a *guarded command*. In some sense, *guarded commands* can be viewed as the implementation of a formula defining a component. An *atomic* formula consists of an equality (or inequality) between terms, or a predicate on a c-semiring value. Given a clause ϕ, the set S_ϕ refers to its atomic formulas. Let p, q be port variables, and m be a memory variable. The construction of a guarded command from a clause ϕ is defined as follows:

- $p = * \in S_\phi$. Formulas of this form are present in the representation of an SCA solely to ensure correct composition. The equality $p = *$ is not added to either the guard or the command.
- $p \neq * \in S$. If p is an input, we add to the guard the predicate $canPut(p)$, where $canPut(p)$ is true when the port is able to provide data. If p is an output, the predicate $canGet(p)$ is added to the guard, where $canGet(p)$ is true when the port is able to get data.
- $m = * \in S$. The predicate $empty(m)$ is added to the guard, where $empty(m)$ is true when no data is present in the memory m. As opposed to a port where data flows from its input through its output, a memory stores data. The equality is thus a guard on the current content of m.
- $m' = * \in S$. The assignment $m \mapsto$ null is added to the command, where null represents no value.
- $m = p \in S$. If p is an output, then the assignment $p \mapsto m$ is added to the command. In order to be well defined, the assignment assumes m to have a value different from null, since the data domain of p does not contain null. Thus, the predicate $nonEmpty(m)$ is added to the guard, where $nonEmpty(m)$ is true when m contains data.[9]
- $m' = p \in S$. If p is an input, the assignment $m' \mapsto p$ is added to the command, and the predicate $canGet(p)$ is added to the guard.[9]
- $m' = m \in S$. The assignment $m' \mapsto m$ is added to the command.
- $p = q \in S$. If p is an input and q is an output, the assignment $q \mapsto p$ is added to the command. Moreover, predicate $canGet(p) \wedge canPut(q)$ is added to the guard. We apply symmetric arguments if p is an output and q an input.[9]
- $P_t(e) \in S$. The predicate $P_t(e)$ is added to the guard.

[9] We omit the cases $m = p$ with p an input, or $m' = p$ with p an output, or $p = q$ with p and q both output or input as they do not occur in the representation of SCA.

Rewrite system. Structurally, guarded commands are analogous to rewrite rules. The guard is similar to the left hand side of a rewrite rule and the command to its right hand side. Additionally, the predicate for comparing a c-semiring value and a threshold has a natural translation as a condition on the rewrite rule: the rewrite rule is available only if the c-semiring value is higher than the threshold. Within this context, we present a translation of guarded commands to rewrite rules in Maude.

Maude uses sorts to type variables. The sort Data^* refers to the set $\mathsf{Data} \cup \{*\}$, where Data in our example is the unary domain $\{1\}$. The sort Fact types the representation of elements of the system state, that corresponds to port and memory values. Elements of the sort Facts are multisets of elements from Fact with multiset union corresponding to conjunction. Sorts such as String or Nat have the expected interpretation. In Maude, ports and memory cells are defined with two constructors

$$\mathsf{p}(\_,\_) : \mathsf{String}\ \mathsf{Data}^* \to \mathsf{Fact}$$

and

$$\mathsf{m}(\_,\_) : \mathsf{Nat}\ \mathsf{Data}^* \to \mathsf{Fact}$$

Each port is identified by a string (e.g., the action name), and each memory cell is identified by a natural number. Data flow is represented by values of sort Data, and data synchronization constraints are encoded using $*$.

Predicates in a guard are terms in the left hand side of its corresponding rewrite rule.

- $\mathit{canGet}(p)$ and $\mathit{nonEmpty}(m)$ are respectively translated to the terms $\mathsf{p}(\text{“p”}, \mathsf{d})$ and $\mathsf{m}(\mathsf{id}, \mathsf{d})$, where id is a natural number corresponding to m and d is of sort Data.
- $\mathit{canPut}(p)$ and $\mathit{empty}(m)$ are respectively translated to the terms $\mathsf{p}(\text{“p”}, *)$ and $\mathsf{m}(\mathsf{id}, *)$, where id is a natural number corresponding to m.
- $P_t(e)$ is translated to a condition on the rewrite rule of the form $\mathsf{if}(\mathsf{t} <= \mathsf{e})$, where t and e are c-semiring variables defined in a different module. If the guard contains multiple predicates over c-semiring values, the generated rewrite rule contains the composite c-semiring value and composite threshold. For every threshold involved in the condition, it is necessary to add $\mathsf{th}(\text{“t”}, \mathsf{t})$ on both the left and the right hand sides of the rewrite rule (it is used but not changed).

Assignments in the command are terms in the right hand side of the rewrite rule.

- $m \mapsto \mathsf{null}$ is translated to the term $\mathsf{m}(\mathsf{id}, *)$.
- $p \mapsto m$ is translated to the term $\mathsf{p}(\text{“p”}, \mathsf{dm})$, where dm is of sort Data and represents the data item in the memory in the left hand side of the rule.
- $m' \mapsto p$ is translated to the two terms $\mathsf{m}(\mathsf{id}, \mathsf{dp})$ and $\mathsf{p}(\text{“p”}, *)$, where dp is the data taken from the port p. The assignment of value p to m results in rewriting the port into a state where $\mathit{canPut}(p)$ is true.

Consider $\mathsf{p}_{\mathsf{move}} \in P$ a port of type output, $m_1, m_2 \in M$ and $m'_1, m'_2 \in M'$. The predicates $P_{t_0}(2)$ and $P_{t_0}(0)$ represent inequalities between values of the weighted c-semiring $\mathbb{W}$. Consider the following formula:

$$\begin{aligned}\phi := \quad & (m_2 \neq * \wedge \mathsf{p}_{\mathsf{move}} = m_2 \wedge m'_2 = m_2 \wedge P_{t_0}(2)) \vee \\ & (m_2 \neq * \wedge m_1 = * \wedge \mathsf{p}_{\mathsf{snap}} = m_2 \wedge m'_1 = m_2 \wedge m'_2 = * \wedge P_{t_0}(0))\end{aligned}$$

As pointed out in the paragraph on logical syntax, each clause of the disjunctive normal form of ϕ is analogous to a transition/behavior of the system: the first clause corresponds to the action move in the state q_Y (represented by the memory variable m_2), and the second clause corresponds to the action snap from state q_Y to q_N (emptying memory m_2 and filling memory m_1).

The formula ϕ has two clauses, and therefore induces two sets of guarded commands. Let the first clause be:

$$\phi_1 := (m_2 \neq * \wedge \mathsf{p}_{\mathsf{move}} = m_2 \wedge m'_2 = m_2 \wedge P_{t_0}(2))$$

The guard set obtained from ϕ_1 is

$$G_{\phi_1} = \{\mathit{nonEmpty}(m_2), \mathit{canPut}(\mathsf{p}_{\mathsf{move}}), P_{t_0}(2)\}$$

and its command set is

$$C_{\phi_1} = \{m'_2 \mapsto m_2, p \mapsto m_2\}$$

The compiler then returns, for each guarded command, its translation to a rewrite rule in Maude. The first rewrite rule shown in Listing 6 corresponds to the first clause of the formula, and the second rewrite rule to the second clause.

```
crl[1] : m(2,d_m2) p(“move", ∗) th(“t0",t0) =>
p(“move", d_m2) m(2, d_m2) th(“t0",t0) if( t0 <= ws(2)) .

crl[2] : m(2,d_m2) m(1, ∗) p(“snap", ∗) th(“t0",t0) =>
p(“snap", d_m2) m(1, d_m2) m(2, ∗ ) th("t0",t0) if( t0 <= ws(0)) .
```

Listing 6: Rewrite rules example.

```
search [1,40] makestart(10,1) =>∗
   F:Facts trace(sl:StepSetList ; {a("move") a("discharge2") s:Step};
                sl1:StepSetList ; {a("move") a("discharge2") s1:Step})
           such that nosnap(sl1:StepSetList) .
```

Listing 7: Reachability query for property ϕ_{w}, with depth search of 40 steps.

The translation of formulas into guarded commands, and the translation of guarded commands to Maude rewrite rules have been implemented in a compiler. An archive for the compiler is accessible at [38], together with some explanations to reproduce the experiments. To give an idea, the compilation time for generating the Maude rewrite system from the Reo circuit encoding the SCAs is less than a few seconds on a Fedora machine, with four 1.6 GHz Intel i5 CPU cores. In the next subsection, we use Maude to verify some properties of the generated rewrite system.

7.4. Verification

Given a soft constraint automaton A with its component action system Σ, as described in Definition 4, the language $L(A)$ is the set of all of its *behaviors*. Analogously, an LTL property ϕ on the alphabet Σ also defines a language $L(A_\phi)$. The inclusion of $L(A) \subseteq L(A_\phi)$ suffices to show whether the automaton A satisfies the property ϕ.

Detection. We use the same example energy-snapshot automata described in Example 1, and now seek to verify whether the following property holds:

$$\phi_{\mathsf{w}} = \succ \Box(\mathsf{move} \to X(\neg\mathsf{move}\; U\; \mathsf{snapshot}))$$

The property ϕ_{w} represents the set of infinite sequences of actions, where in between any two move actions, the action snapshot must appear. The correctness of the rewrite system with regard to the property ϕ_{w}, is defined as a predicate on a trace constructor in Maude:

$$\mathsf{trace} : \mathsf{StepSetList} \to \mathsf{Fact}$$

At runtime, the trace saves the actions performed by the rewrite rules and their associated preferences. The pair of an action together with its c-semiring values is of type Step. A set of steps, described by the sort StepSet, represents a composite action together with the composite c-semiring value. The function trace takes a list of sets of steps, whose type is StepSetList, and returns an element of type Fact.

We extend the notion of *behavior* of an automaton as an infinite sequence of actions accepted by the automaton, by also including the c-semiring value in the behavior. The state of the rewrite system is thus defined by a finite prefix of an infinite sequence of composite actions paired with their c-semiring values. The trace generates this prefix iteratively during the execution of the rewrite system. Finding a counterexample for the property ϕ_{w} can be turned into a reachability problem on the states of the rewrite system (e.g., on an element of the trace). The reachability query in Listing 7 specifies that each sequence of actions (of sort StepSetList) between two move actions must not contain an action snapshot. For all finite prefixes of the *behavior* of the (composed) automaton, Maude will search for a violation of the property ϕ_{w}, i.e., a sequence $\langle \sigma_1, \mathsf{move}_2, \sigma_2, \mathsf{move}_2 \rangle$ where $\sigma_1, \sigma_2 \in \Sigma^*$ and $\mathsf{snapshot}_1$ does not appear in σ_2.

The number of counterexamples and the depth of the search are specified in brackets after the search command. The threshold value for A_{e} and A_{s} are given as arguments for the initial state with $\mathsf{makestart}(10, 1)$. In this example, $t_{\mathsf{e}} = 10$ and $t_{\mathsf{s}} = 1$. The composition is the join composition (of SCA), therefore $t_{\mathsf{e,s}} = t_{\mathsf{e}} \otimes t_{\mathsf{s}} = 11$.

With an initial composite threshold $t_{\mathsf{e,s}} = 11$, the search command of Listing 7 returns a counterexample: it is possible to perform move_2 twice from the initial state. More generally, all sequences starting with the prefix $\langle \mathsf{move}_2, \mathsf{move}_2 \rangle$ are detected as counterexamples. We refer to δ as the finite prefix verified by Maude. We then let $d_{A_{\mathsf{e,s}}}(\delta)$, be the diagnostic preference of a finite sequence of actions. In the case where $\delta = \langle \mathsf{move}_2, \mathsf{move}_2 \rangle$, $d_{A_{\mathsf{e,s}}}(\delta) = 7$.

Diagnosis. Given $\delta = \langle \mathsf{move}_2, \mathsf{move}_2 \rangle$ and $d_{A_{\mathsf{e,s}}}(\delta) = 7$, can we localize the minimal set of suspect components of $A_{\mathsf{e,s}}$?

The diagnosis procedure described in Section 6 gives an algorithm to find the minimal set of suspect component sets, and defines the new threshold value in terms of the diagnostic preference.

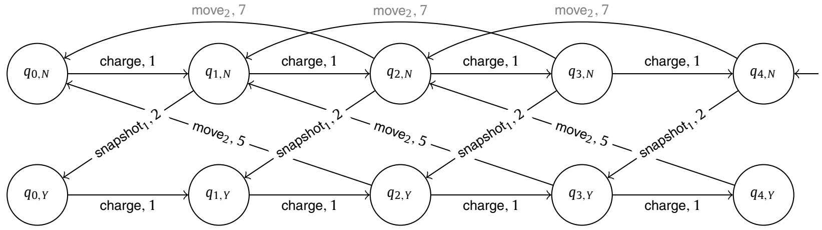


Fig. 12. Resulting composition of SCAs A_e and A_s when $t_e = 5$ and $t_s = 1$.

```
search [1, 100] makestart(10,1) =>* F:Facts m(5,d:Data) .
```

Listing 8: Reachability query for not running out of energy.

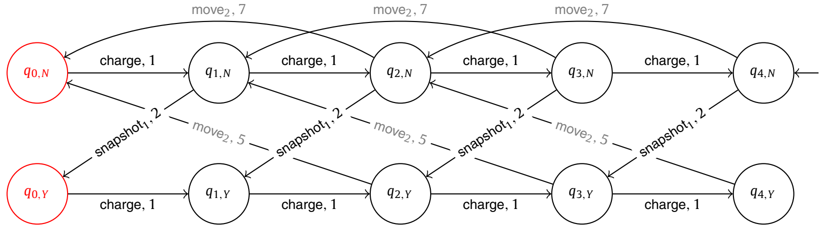


Fig. 13. Resulting composition of SCAs A_e and A_s when $t_e = 1$ and $t_s = 1$. (For interpretation of the colors in the figure, the reader is referred to the web version of this article.)

We first set the threshold $t_s = \mathbf{1}_{\mathbb{W}}$ and leave t_e as is. The composite threshold is then $t_s \otimes t_e = 0 + 10 = 10 \leq 7$. Since raising the threshold t_s to its maximum did not raise the composite threshold beyond the diagnosis threshold value of 7, we conclude that the component A_s is *innocuous*.

On the other hand, setting $t_e = \mathbf{1}_{\mathbb{W}}$ and leaving $t_s = 1$ results in $t_e \otimes t_s = 0 + 1 > 7$. In this case, the composite threshold is now greater than the diagnostic preference, based on which we consider A_e as a *suspect*. We find the threshold $t_e = 5$ as the minimal threshold value from the suspect component that raises the composite threshold higher than the diagnostic preference.

We repeat the search command for this new threshold: $t_e = 5$ and $t_s = 1$. The result of the search command is now satisfying: Maude reports no counterexamples (in a search up to depth 40). To confirm our intuition of our bounded model checking result, we examine the resulting soft component automata with its enabled transitions as presented in Fig. 12. Transitions containing a c-semiring value lower than the threshold are represented in gray. Those transitions are not part of the definition of the language anymore.

Reachability queries in Maude are written manually, and a future work would be to find an automatic translation from a useful subset of our LTL variant to reachability queries in Maude. An extension of our current Maude implementation can proceed to examine the trace within Maude's meta-level. In this way, Maude will be able to dynamically compute the diagnostic preference and subsequently modify the threshold values accordingly.

A second property we would like to verify in our example, which we denote as ϕ_e, represents the case where the automaton never runs out of energy, i.e., it never reaches state $q_{0,N}$ or $q_{0,Y}$.

Applying the same procedure as for the property ϕ_w, we now adjust the threshold such that all traces do not reach a state in which memory m_5 has data (corresponding to the states $q_{0,N}$ and $q_{0,Y}$ where the component is out of energy). Although this property is not straightforwardly representable within the alphabet Σ, we can specify this property as a reachability query in Maude as shown in Listing 8. The property is satisfied if the search command does not find any path where m_5 has data. These states are represented in red in Fig. 13. As a consequence of the diagnosis algorithm, the threshold t_e is set to 1 and t_s remains at its initial value 1. All transitions in gray are now filtered out, and the property of Listing 8 is verified.

Observe that although the diagnosis algorithm correctly adjusted the thresholds to cope with the undesired behavior, it has also eliminated the desirable behavior of moving and taking snapshots. Ideally, we want the outcome of the diagnosis process to remove the faulty behaviors while keeping desired behaviors. This example suggests the need for additional techniques to adjust action preferences as well as component thresholds, in order to keep the desired behavior while satisfying some properties.

Our examples demonstrate that the threshold assigned to a component plays an important role in diagnosis. The relation between the functionality of an automaton in a composition and the value of its threshold should be carefully considered by the designer of a soft component automaton. Moreover, the designer must keep in mind that components with lower threshold values are more likely to be detected as *suspect*.

We showed how a designer can modify system component thresholds to eliminate undesired behaviors. By raising some thresholds, some transitions will no longer be allowed. As we suggested, this verification procedure can be automated and in fact, using reflection, can be carried out by the system at runtime, where the system keeps verifying the trace produced with regard to a set of properties. Detection of deviations would then trigger the diagnosis mechanism to localize the set of suspect components and allow new threshold values to be set.

8. Discussion

In this paper, we proposed a framework that facilitates the construction of autonomous agents in a compositional fashion. We furthermore considered an LTL-like logic for verification of the constructed models that takes their compositional nature into account, and showed the added value of operators related to composition in verifying properties of the interface between components. We also provided a decision procedure for the proposed logic.

The agents in our proposed framework are “soft”, in that their actions have preferences, which may or may not make an action feasible depending on the value of a preference threshold. The designer can *decrease* the value of this threshold to allow for more behavior, possibly to accommodate the preferences of another component, or *increase* it to restrict undesired behavior observed at run-time or counterexamples to safety assertions found at design-time. We considered a simple method to raise the threshold enough to exclude a given behavior. However, this method may overapproximate in the presence of partially ordered preferences and possibly exclude desired behavior.

In case of a composed system, one can also find out which components’ thresholds can be thought of as *suspect* for allowing a certain behavior. This information can give the designer a hint on how to adjust the system — for example, if the threshold of an energy management component turns out to be suspect for the inclusion of some undesired behavior, perhaps the component’s threshold needs to be more conservative with regard to energy expenses to avoid the undesired behavior. We stress that responsibility may be assigned to a *set* of components as a whole, if their composed threshold is suspect for allowing some undesired behavior, which is possible when preferences are partially ordered.

Lastly, we showed that automata can be compiled into the Reo coordination language, which can in turn be compiled into Maude. The latter allows us to ask verification questions close to those allowed by our proposed dialect of LTL; the results of those verification questions can then be used as input to the diagnostics procedure. Using Reo as an intermediate language also leaves open the possibility of compiling the Reo incarnation of a model into other target languages, or using existing tools for verification and optimization of Reo models.

Preferences are a mechanism for an agent to evaluate multiple options for action, but they do not say how to choose among acceptable actions. Maude has a rich language for specifying strategies [39] and Maude’s reflection capability allows the user to specify strategies beyond the power of the strategy language. This capability could be used to augment preferences to further constrain the allowed executions, for example choosing maximally preferred actions.

9. Further work

Throughout our investigation, the tools for verification and diagnosis were driven by the compositional nature of the framework. As a result, they apply not only to the “grand composition” of all components of the system, but also to subcomponents (which may themselves be composed of sub-subcomponents). What is missing is a way to “lift” verified properties of subcomponents to the level of a composed system, possibly with a side condition on the interface between the subcomponent where the property holds and the subcomponent representing the rest of the system, along the lines of the interface verification in Section 5.1.

If we assume that agents have low-latency and noiseless communication channels, one can also think of a multi-agent system as the composition of SCAs, each of which represents an agent. As such, our methods may also apply to verification and diagnosis of multi-agent systems. However, this assumption may not hold. One way to model multi-agent systems communicating through high-latency and/or noisy channels entails inserting “glue components” that mediate the communication between agents, by introducing delay or noise. Another method would be to introduce a new form of composition for loosely coupled systems.

Constraint automata can be run by a distributed algorithm [44]; this facilitates their deployment in a setting where independent but communicating processes are responsible for the execution of subcomponents. A similar distributed algorithm for soft component automata, which generalize constraint automata, would also be useful. The problem here lies in accommodating preferences: an action may well have a very high preference in an individual component, but its composed preference may turn out to be smaller, and indeed the highest-preference composed action may capture a component action of lower preference. The challenge thus becomes to reconcile the preferences of individual components in a distributed manner.

Finding an appropriate threshold value also deserves further attention. In particular, a method to adjust the threshold value *at run-time*, would be useful, so as to allow an agent to relax its goals as gracefully as possible if its current goal appears unachievable, and raise the bar when circumstances improve.

Koehler and Clarke [35] wrote about decomposition of port automata, which can be seen as a special case of soft component automata, and showed that all port automata are compositions of port automata from a small set of generators. This result is interesting from a design viewpoint, because it tells us that the mechanism represented by a port automaton can eventually be traced back to the mechanisms represented by the generators. We would like to see if this result generalizes to the setting of component action systems.

Acknowledgements

The authors would like to thank Vivek Nigam and the anonymous referees for their valuable feedback. This work was partially supported by ONR grant N00014-15-1-2202 and the ERC Starting Grant ProFoundNet (grant code 679127).

References

[1] F. Arbab, Reo: a channel-based coordination model for component composition, Math. Struct. Comput. Sci. 14 (3) (2004) 329–366.
[2] F. Arbab, F. Santini, Preference and similarity-based behavioral discovery of services, in: Proc. Web Services and Formal Methods (WS-FM), 2012, pp. 118–133.
[3] C. Baier, T. Blechmann, J. Klein, S. Klüppelholz, W. Leister, Design and verification of systems with exogenous coordination using Vereofy, in: Proc. Int. Symp. on Leveraging Applications (ISoLA), 2010, pp. 97–111.
[4] C. Baier, J. Klein, S. Klüppelholz, Synthesis of Reo connectors for strategies and controllers, Fundam. Inform. 130 (2014) 1–20.
[5] C. Baier, M. Sirjani, F. Arbab, J. Rutten, Modeling component connectors in Reo by constraint automata, Sci. Comput. Program. 61 (2006) 75–113.
[6] V.G. Bharadwaj, J.S. Baras, Towards automated negotiation of access control policies, in: Proc. Policies for Distributed Systems (POLICY), 2003, pp. 111–119.
[7] S. Bistarelli, Semirings for Soft Constraint Solving and Programming, LNCS, vol. 2962, Springer, 2004.
[8] S. Bistarelli, U. Montanari, F. Rossi, Constraint solving over semirings, in: Proc. Int. Joint Conference on Artificial Intelligence (IJCAI), 1995, pp. 624–630.
[9] S. Bistarelli, U. Montanari, F. Rossi, Semiring-based constraint satisfaction and optimization, J. ACM 44 (2) (1997) 201–236.
[10] J.R. Büchi, On a decision method in restricted second order arithmetic, in: Proc. Logic, Methodology and Philosophy of Science, Stanford Univ. Press, Stanford, CA, 1962, pp. 1–11.
[11] P. Casanova, D. Garlan, B.R. Schmerl, R. Abreu, Diagnosing unobserved components in self-adaptive systems, in: Proc. Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2014, pp. 75–84.
[12] D. Clarke, J. Proença, A. Lazovik, F. Arbab, Channel-based coordination via constraint satisfaction, Sci. Comput. Program. 76 (8) (2011) 681–710.
[13] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C. Talcott, All About Maude: A High-Performance Logical Framework, LNCS, vol. 4350, Springer, 2007.
[14] R. Debouk, S. Lafortune, D. Teneketzis, Coordinated decentralized protocols for failure diagnosis of discrete event systems, Discrete Event Dyn. Syst. 10 (1–2) (2000) 33–86.
[15] K. Dokter, F. Arbab, Rule-based form for stream constraints, in: Proc. Coordination Models and Languages (COORDINATION), 2018, pp. 142–161.
[16] K. Dokter, F. Arbab, Treo: textual syntax for Reo connectors, https://arxiv.org/abs/1806.09852, 2018.
[17] M. Droste, P. Gastin, Weighted automata and weighted logics, Theor. Comput. Sci. 380 (1) (2007) 69–86.
[18] M. Droste, W. Kuich, H. Vogler, Handbook of Weighted Automata, 1st edn., Springer, 2009.
[19] M. Droste, G. Rahonis, Weighted automata and weighted logics on infinite words, Russ. Math. 54 (2010) 26–45.
[20] E.A. Emerson, C. Lei, Temporal reasoning under generalized fairness constraints, in: Proc. Theoretical Aspects of Computer Science (STACS), 1986, pp. 21–36.
[21] E.A. Emerson, C. Lei, Modalities for model checking: branching time logic strikes back, Sci. Comput. Program. 8 (3) (1987) 275–306.
[22] F. Gadducci, M.M. Hölzl, G.V. Monreale, M. Wirsing, Soft constraints for lexicographic orders, in: Proc. Mexican Int. Conference on Artificial Intelligence (MICAI), 2013, pp. 68–79.
[23] G. Gössler, L. Astefanoaei, Blaming in component-based real-time systems, in: Proc. Embedded Software (EMSOFT), 2014, pp. 7:1–7:10.
[24] G. Gössler, J. Stefani, Fault ascription in concurrent systems, in: Proc. Trustworthy Global Computing (TGC), 2015, pp. 79–94.
[25] M.M. Hölzl, M. Meier, M. Wirsing, Which soft constraints do you prefer?, Electron. Notes Theor. Comput. Sci. 238 (3) (2009) 189–205.
[26] H. Hüttel, K.G. Larsen, The use of static constructs in a modal process logic, in: Proc. Symp. on Logical Foundations of Computer Science, 1989, pp. 163–180.
[27] S.S. Jongmans, F. Arbab, Overview of thirty semantic formalisms for Reo, Sci. Ann. Comput. Sci. 22 (2012) 201–251.
[28] S.S.T. Jongmans, Automata-Theoretic Protocol Programming, Ph.D. thesis, Universiteit Leiden, 2015, http://hdl.handle.net/1887/38223.
[29] S.T. Jongmans, T. Kappé, F. Arbab, Constraint automata with memory cells and their composition, Sci. Comput. Program. 146 (2017) 50–86.
[30] T. Kappé, Logic for Soft Component Automata, Master's thesis, Leiden University, Leiden, the Netherlands, 2016.
[31] T. Kappé, F. Arbab, C. Talcott, A component-oriented framework for autonomous agents, https://arxiv.org/abs/1708.00072, 2017.
[32] T. Kappé, F. Arbab, C.L. Talcott, A compositional framework for preference-aware agents, in: Proc. Workshop on Verification and Validation of Cyber-Physical Systems (V2CPS), 2016, pp. 21–35.
[33] T. Kappé, F. Arbab, C.L. Talcott, A component-oriented framework for autonomous agents, in: Proc. Formal Aspects of Component Software (FACS), 2017, pp. 20–38.
[34] M. Kim, I.A. Mason, C. Talcott, Detection and Diagnosis of Deviations in Distributed Systems of Autonomous Agents, Technical Report, 2018, available from https://github.com/SRI-CSL/SoftAgentsDiagnosis.git.
[35] C. Koehler, D. Clarke, Decomposing port automata, in: Proc. ACM Symp. on Applied Computing (SAC), 2009, pp. 1369–1373.
[36] O. Kupferman, M.Y. Vardi, Weak alternating automata are not that weak, ACM Trans. Comput. Log. 2 (3) (2001) 408–429.
[37] J. Laurent, J. Yang, W. Fontana, Counterfactual resimulation for causal analysis of rule-based models, in: Proc. International Joint Conference on Artificial Intelligence (IJCAI), 2018, pp. 1182–1890.
[38] B. Lion, T. Kappé, C. Talcott, F. Arbab, Treo-to-Maude, https://doi.org/10.5281/zenodo.2583974, 2019.
[39] N. Martí-Oliet, J. Meseguer, A. Verdejo, A rewriting semantics for Maude strategies, Electron. Notes Theor. Comput. Sci. (2009) 227–247.
[40] I.A. Mason, V. Nigam, C. Talcott, A. Brito, A framework for analyzing adaptive autonomous aerial vehicles, in: Proc. Workshop on Formal Co-Simulation of Cyber-Physical Systems (CoSim), 2017.
[41] S. Miyano, T. Hayashi, Alternating finite automata on omega-words, Theor. Comput. Sci. 32 (1984) 321–330.
[42] D.E. Muller, A. Saoudi, P.E. Schupp, Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time, in: Proc. Logic in Computer Science (LICS), 1988, pp. 422–427.
[43] J. Neidig, J. Lunze, Decentralised diagnosis of automata networks, IFAC Proc. Vol. 38 (1) (2005) 400–405.

[44] J. Proença, D. Clarke, E.P. de Vink, F. Arbab, Decoupled execution of synchronous coordination models via behavioural automata, in: Proc. Foundations of Coordination Languages and Software Architectures (FOCLASA), 2011, pp. 65–79.

[45] J.C. Reynolds, GEDANKEN - a simple typeless language based on the principle of completeness and the reference concept, Commun. ACM 13 (5) (1970) 308–319.

[46] J.J.M.M. Rutten, A coinductive calculus of streams, Math. Struct. Comput. Sci. 15 (1) (2005) 93–147.

[47] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, D. Teneketzis, Failure diagnosis using discrete-event models, IEEE Trans. Control Syst. Technol. 4 (2) (1996) 105–124.

[48] M.P. Schützenberger, On the definition of a family of automata, Inf. Control 4 (2) (1961) 245–270.

[49] C.L. Talcott, F. Arbab, M. Yadav, Soft agents: exploring soft constraints to model robust adaptive distributed cyber-physical agent systems, in: Software, Services, and Systems – Essays Dedicated to Martin Wirsing on the Occasion of His Retirement from the Chair of Programming and Software Engineering, 2015, pp. 273–290.

[50] C.L. Talcott, V. Nigam, F. Arbab, T. Kappé, Formal specification and analysis of robust adaptive distributed cyber-physical systems, in: Formal Methods for the Quantitative Evaluation of Collective Adaptive Systems – Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2016), 2016, pp. 1–35.

[51] M.Y. Vardi, An automata-theoretic approach to linear temporal logic, in: Proc. Logics for Concurrency – Structure Versus Automata (Banff Higher Order Workshop), 1995, pp. 238–266.

[52] M.Y. Vardi, P. Wolper, Reasoning about infinite computations, Inf. Comput. 115 (1) (1994) 1–37.

[53] G. Winskel, An introduction to event structures, in: Proc. Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, 1988, pp. 364–397.