Fundamenta Informaticae XX (2004) 1–31
IOS Press

Synthesis of Open Reactive Systems from Scenario-Based Specifications

Yves Bontemps* **and Pierre-Yves Schobbens**
Computer Science Department, University of Namur
Namur, Belgium
ybo@info.fundp.ac.be
pys@info.fundp.ac.be

Christof Löding
Lehrstuhl für Informatik VII, RWTH Aachen
Aachen, Germany
loeding@informatik.rwth-aachen.de

Abstract. We propose here Live Sequence Charts with a new, game-based semantics to model interactions between the system and its environment. For constructing programs automatically, we give an algorithm to synthesize either a strategy for the system ensuring that the specification is respected, or, if the specification is unimplementable, a strategy for the environment forcing the system to fail. We introduce the concept of mercifulness, a desirable property of the synthesized program. We give a polynomial time algorithm for synthesizing merciful winning strategies.

1. Introduction

The problem of constructing open reactive systems is now of capital importance, since the pervasiveness of open, distributed systems in our societies has made us heavily dependent on such systems, while their development is known to be error-prone [29]. Methods to ensure the development of such reliable systems are thus a more than urgent need. Formal methods have often been cited for their promise of reliability. However, they encounter three main obstacles:

Address for correspondence: FUNDP, Institut d'Informatique, 21, rue Grandgagnage, B5000 - Namur (Belgium)
* FNRS Research Fellow. Work done partly during a stay at Lehrstuhl für Informatik VII, RWTH Aachen, funded by an FNRS "crédit pour bref séjour" grant.

1. Is the semantic basis of the formal method adequate for its application domain? For instance, most formal developments in industry use methods devised for closed sequential systems [17], even when the domain requires open reactive models, so that the guarantees provided end up being weak in spite of the large effort devoted to this development. It is widely held, [43, 56, 2, 1], that game-based semantics are the adequate basis for open reactive systems.
2. Is the initial specification an adequate description of the real-life problem? There cannot be an absolute answer to this question, but it is clear that formalisms that are close to the intuition of the users of the system provide a much better basis for methods addressing this problem. Scenario-based techniques have shown an advantage in many studies, [58, 7], specially if they are presented in a graphical form.
3. Is the formal development achievable with reasonable human effort? Most current formal techniques are based on *a posteriori* verification, where a very large effort is devoted to prove that the final implementation refines the initial specification. In most cases, errors have been introduced and the effort must be started again after corrections. It seems more efficient to ensure correctness by construction. Here, we go a step further and *synthesize* the implementation from the specification. We do not, however, consider this as the final answer: since the synthesized implementation can reveal problems in the specification, we inscribe this work in the larger framework of *round-trip engineering*, where the connection backwards to the specification is also possible. This aspect will not be addressed in this short paper, however.

We base our work on Message Sequence Charts (MSCs) [30] which is a well-established notation used in telecommunication industry, that is now spreading in many other industries and has been included as *sequence diagrams* in UML [42]. It is very intuitive and readable, but engineers use it with several different meanings that are not reflected in its syntax nor its semantics. We hope here to capture an important part of this implicit meaning by using a game-theoretic semantics, namely choice, response, control, strategies.

The paper thus brings several contributions: In section 2, it extends MSCs to incorporate the missing semantic notions, by following the syntax of Live Sequence Charts (LSCs) [16] but not their semantics. In Section 3, we show how this language can be understood as a game-theoretic specification language giving responsibilities and powers to its players, even though no explicit syntax to express these concepts is introduced. In Section 4, we show how to synthesize a strategy, which is actually a finite memory program implementing the system under development. A comparison to related work is also given. In Section 5, we introduce the concept of mercifulness, a desirable property of the synthesized program.

2. Live Sequence Charts

2.1. A Quick Tour of the Language

Nowadays, scenario-based approaches to software engineering have proven their practical value. Two popular scenario-based notations are the use-case notation [31] and Message Sequence Charts [30].

However, these scenarios, that group typical sequences of events, are but examples of interactions between the different actors and the system under development (SUD). They are therefore not adapted to the *specification* of this system, as they do not draw the line between admissible and inadmissible behaviors.

Moving directly from examples, presented as scenarios, to some expressive specification language, such as state machines or temporal logics, is too big a leap to be taken. This paradigm shift would increase the risk to state properties which are not desired by the user. In order to obtain a smoother transition from examples, stated as scenarios, to the specification of the system, a scenario-based *specification* language is needed [18].

For this purpose, Live Sequence Charts (LSCs) have been introduced by Damm and Harel in 1998 [16], as an extension of Message Sequence Charts whose semantics [15], a mere partial order between events, was felt to be too weak.

Indeed, MSCs were silent about two questions:

1. *message abstraction*, i.e. whether messages *not* appearing explicitly in the chart may occur at will during its execution, or whether they are forbidden by their mere absence. This notion has recently been added to UML Interaction Diagrams [42];

2. chart status, i.e. whether the scenario is mandatory (or *universal*), i.e. the system shall *always* behave as described, or provisional (or *existential*), i.e. the scenario gives a possible execution of the system.

In the remainder of this section, we will describe the variant of LSCs in which we are interested. This variant differs from the original definition [16] by ignoring conditions, structuring constructs, modalities (hot/cold) on locations and messages, and by adding initial charts. We consider only instantaneous messages, too.

An LSC is made of *instances*, depicted by vertical lines, along which time flows, from top to bottom. For example, Fig. 1(a) contains two instances: `Customer` and `Machine`. *Locations* are the points along these lines where *events*, such as `askCocoa`, occur. Along an instance line, locations are ordered from top to bottom. However, one can relax this order and put locations into the same *co-region*. Graphically, a small dashed line is drawn along the unordered locations, e.g. `askCocoa` and `insertCoin` in Fig.1(a). This means that the user may first insert a coin and then ask for some hot chocolate, or proceed the other way round.

Instances interact by message passing, displayed as named arrows. For simplicity, we will assume here that messages are instantaneous. Non-instantaneous messages can still be represented in our setting by modeling communication channels as instances.

LSCs [16, 32] have one of the following three modes: initial, existential and universal. An initial scenario is surrounded by a pentagonal box, as in Fig. 2(a). These scenarios must be observed at the beginning of every execution of the SUD. An existential chart (Fig. 2(b)) has a dashed-line border. It must be observed in some execution of the SUD. A universal chart (Fig. 1) is made of two superimposed parts: the upper one, surrounded by an hexagonal dashed-line box, is the *prechart*. The lower part, drawn within a solid-line rectangle is the *main chart*. Whenever the behavior contained in the prechart is matched, the run must obey the behavior described by the main chart, afterward. It thus expresses the specification pattern of *response* [20].

In LSCs, one can state which events are *restricted* or, in UML parlance, *considered*. These events may only occur when they are explicitly shown in the chart. Other, *unconstrained* events may occur at will. By default, every event appearing in a chart is supposed to be restricted. For instance, in Fig. 1(a) events `pourWater` and `fill` (from scenarios 1(c) and 1(d)) may occur between `prepCocoa` and `serveCocoa`. On the contrary, message `moneyBack` may not occur during chart 1(a) execution, because it

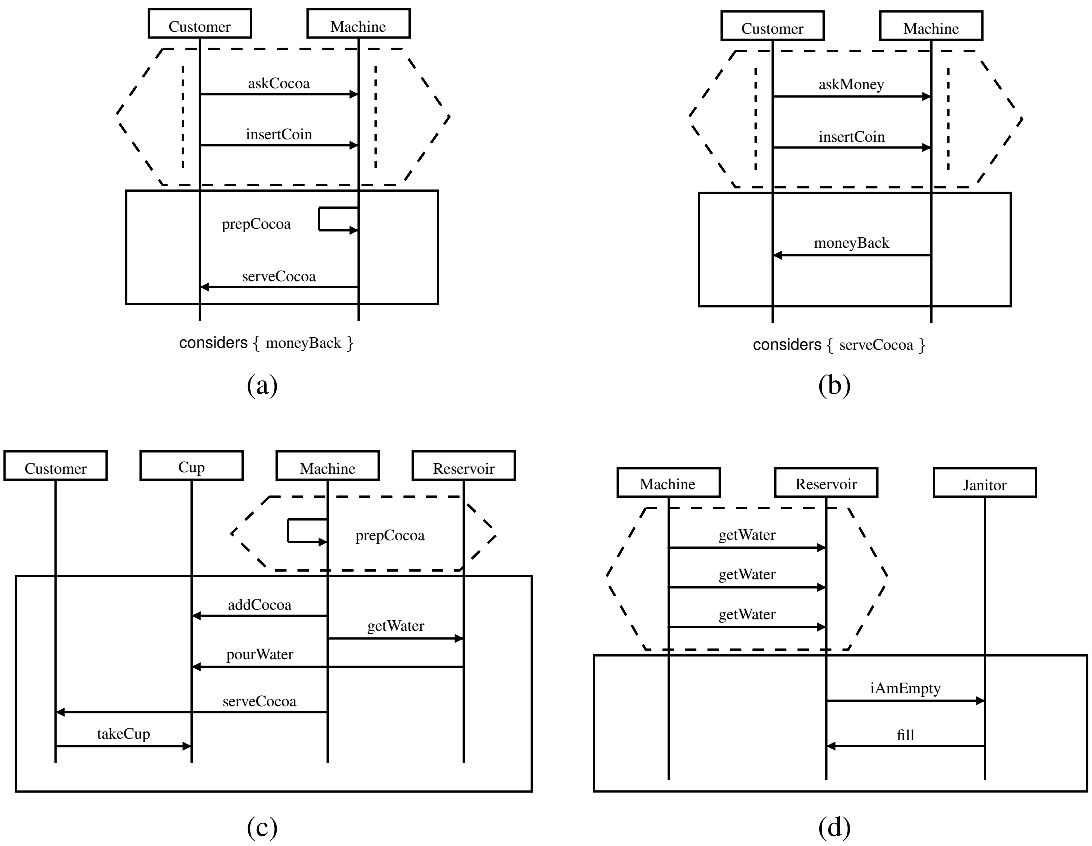

Figure 1. Universal LSCs

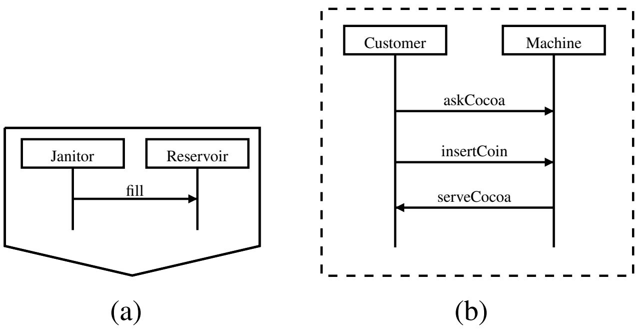

Figure 2. Initial and Existential LSCs

is restricted by a "considers" clause. This precisely states that, if we are serving chocolate, money will not be given back.

2.2. Syntax and Semantics

We are given a finite set of *agents* Ag and a finite set of *message names* $\mathcal{M}$. Then, $\Sigma = Ag \times \mathcal{M} \times Ag$ is the set of all possible *events*, (a_1, m, a_2) meaning that a_1 sends message m to a_2. Every event is *controlled* by its sender.

A *specification* is a set of LSCs, which, in turn, consist of basic charts. A *basic chart* is similar to an MSC. It is a tuple

$$C = \langle \Sigma(C), A, (L_a, <_a)_{a \in A}, \phi_C, msg \rangle,$$

where

$\Sigma(C) \subseteq \Sigma$ are the events restricted in C.

$A \subseteq Ag$ is a finite set of agents involved in C;

L_a are the locations appearing on a's instance line. We will use L_C as a shortcut for $\bigcup_{a \in A} L_a$;

$<_a \subseteq L_a \times L_a$ is a strict partial order (transitive, anti-symmetric, irreflexive relation) representing the order in which L_a locations should be reached. Because of co-regions, this is *not* a total order on L_a;

$\phi_C : L_C \to \Sigma(C)$ labels locations with restricted events. When the basic chart that we are dealing with will be clear from the context, we will drop the subscript C and simply write ϕ.

msg is a *communication function*, which is represented by arrows in the graphical form. It is a bijective function such that:

1. the domain and co-domain of *msg* partition L_C.
2. $\forall l \in dom(msg) : \phi_C(l) = \phi_C(msg(l))$.
3. $\forall l \in dom(msg) : l \in L_{\pi_1(\phi_C(l))}$ and $msg(l) \in L_{\pi_3(\phi_C(l))}$, where $\pi_j(t)$ retrieves the j-th component of a tuple t.

Two locations belonging to the same co-region must bear distinct labels.

Given an arbitrary basic chart, one can build a (strict) partial order $\prec_C$ as follows. First, take the smallest preorder (transitive, reflexive, not necessarily anti-symmetric relation) $\preceq$, such that

1. $l <_a l'$, for some $a \in Ag$, implies that $l \preceq l'$,
2. $l \in dom(msg)$ implies that $l \preceq msg(l)$ and $msg(l) \preceq l$.

A preorder defines equivalence classes, by letting $l \sim l'$ iff $l \preceq l'$ and $l' \preceq l$. A basic chart in which all locations belonging to the same equivalence class are labeled with the same event, is dubbed *well-formed*. We will only consider well-formed basic charts. Graphically, this constraint will be met by not letting messages cross each other. By restricting ourselves to well-formed basic charts, we can lift ϕ to equivalence classes. By abuse of language, we will let L_C denote the equivalence classes of C and ϕ_C

be the labeling function lifted to equivalence classes. When lifting $\preceq$ to equivalence classes ($[l]_\sim \leq [l']_\sim$ iff $l \preceq l'$), we end up with a partial order.

Thus, every well-formed basic chart C defines a *labeled partial order* (lpo) $\langle L_C, \prec_C, \phi_C \rangle$, i.e. a strict partial order, whose nodes are labeled with the symbols of some fixed alphabet (here, Σ). By $C_1 \cdot C_2$, we will denote the concatenation of the partial orders of C_1 and C_2. This operation is defined if locations of both lpos are disjoint. It results in the component-wise union of the two lpos, plus edges ensuring that every location from C_2 are greater than C_1 locations.

We will let C, possibly with subscripts, range over basic charts. An existential chart is of the form $\Diamond C$, an initial chart of the form $\rhd C$ and a universal chart is denoted $\Box(C_1, C_2)$, where C_1 is the prechart and C_2 the main chart. Regarding universal charts, we require that $\Sigma(C_1) = \Sigma(C_2)$.

A *linearization* of a partial order is a total order extending this partial order. Thus, the linearization of a lpo is a sequence of events.

Definition 2.1. ($r \models_B C$)
A sequence $r = e_1 \dots e_n \in \Sigma^*$ *fulfills a basic chart* C ($r \models_B C$) iff $r|_{\Sigma(C)}$ is a linearization of the lpo $\langle L_C, \prec_C, \phi_C \rangle$. The projection operator $|_{\Sigma(C)}$ removes from a given sequence all events that are not in $\Sigma(C)$. Formally, $\epsilon|_A = \epsilon$ and $we|_A = w|_A$ if $e \in A$ or $we|_A = (w|_A)e$, if $e \notin A$.

We are now in a position to determine which runs are accepted by an LSC.

Definition 2.2. ($r \models_L S$)
An infinite run $r = e_1 e_2 \dots$ will satisfy an LSC S iff

1. $S = \rhd C$ and $\exists i : i \geq 1 : e_1 \dots e_i \models_B C$
2. $S = \Diamond C$ and $\exists i, j : 1 \leq i \leq j : e_i \dots e_j \models_B C$;
3. $S = \Box(C_1, C_2)$ and $\forall i, j : j \geq i \geq 1 :$

$$(e_i \dots e_j \models_B C_1) \implies (\exists k : i \leq j \leq k : e_j \dots e_k \models_B C_2).$$

Example 2.1. For example, consider the following runs and the universal LSC of Fig. 1(a).

$$(\texttt{askCocoa} \cdot \texttt{insertCoin} \cdot \texttt{prepCocoa} \cdot \texttt{serveCocoa})^\omega \tag{1}$$
$$(\texttt{askCocoa} \cdot \texttt{moneyBack})^\omega \tag{2}$$
$$(\texttt{askCocoa} \cdot \texttt{insertCoin} \cdot \texttt{prepCocoa} \cdot \texttt{moneyBack} \cdot \texttt{serveCocoa})^\omega \tag{3}$$
$$\texttt{askCocoa} \cdot \texttt{insertCoin} \cdot \texttt{prepCocoa} \cdot (\texttt{iAmEmpty})^\omega \tag{4}$$

The runs (1) and (2) are models of Fig. 1(a), because, in the former, the prechart is linearized by `askCocoa` $\cdot$ `insertCoin`, and is immediately followed by a linearization of the main chart and, in the latter, the prechart is never matched. The runs (3) and (4) are *not* models of the LSC, because, in both of them, the prechart is matched but, in run (3), event `moneyBack` appears in an unexpected way (wrt main chart) and, in (4), event `serveCocoa` never happens.

A specification $\mathcal{S}$ is a set of LSCs. It can be partitioned into three self-explanatory subsets: $\mathcal{S}_\Diamond$, $\mathcal{S}_\rhd$ and $\mathcal{S}_\Box$.

Definition 2.3. ($R \models_S \mathcal{S}$)
A set of runs R *is a model of* $\mathcal{S}$ ($R \models_S \mathcal{S}$) iff

1. $\forall S \in \mathcal{S}_{\triangleright} \cup \mathcal{S}_{\Box} : \forall r \in R : r \models_L S$
2. $\forall S \in \mathcal{S}_{\Diamond} : \exists r \in R : r \models_L S$.

2.3. Partial Orders and Cuts

The semantics of LSCs relies heavily on linearization of their underlying lpos. Since our algorithms will have to deal with these linearizations, we need some way to recognize linearizations. The notion of cuts provides us with such an operational machinery.

2.3.1. Operational Machinery

We will make use of an operational way to compute the set of linearizations of an lpo. States will be the sets of locations "already reached", that we will call "cuts". Formally, a *cut* in an lpo $\langle L, <, \phi \rangle$ is a subset c of L that is *downward closed*: $\forall l \in c : \forall l' \in L : l' < l \implies l' \in c$. In a cut c, we can reach another cut c' by adding a location l, written $c \xrightarrow{l} c'$, iff $l \notin c$, $c' = \{l\} \cup c$.

Lemma 2.1. (Cuts-linearization)
Let $\mathcal{L} = \langle L, <, \phi \rangle$ be an lpo, then, $\emptyset \xrightarrow{l_1} c_1 \xrightarrow{l_2} c_2 \dots c_{n-1} \xrightarrow{l_n} c_n = L \iff \phi(l_1) \cdot \ldots \cdot \phi(l_n)$ is a linearization of $\mathcal{L}$, where $l_i \in L$, for $1 \leq i \leq n$.

This lemma is easy to verify, by induction on the length of the linearization.

Definition 2.4. (Generated cut)
A finite word $w \in \Sigma^*$ *generates* a cut c in an lpo $\mathcal{L}$ iff some suffix of w linearizes c.

We propose the following recursive definition of $gen(w, \mathcal{L})$, the set of cuts in $\mathcal{L}$ generated by w. We then show that our operational machinery is actually correct, i.e. actually captures all generated cuts.

Definition 2.5. ($gen(w, \mathcal{L})$)

$$
\begin{aligned}
gen(\epsilon, \mathcal{L}) &= \{\emptyset\} \\
gen(w \cdot a, \mathcal{L}) &= \{\emptyset\} \cup \left\{ c' \;\middle|\; \begin{array}{l} \exists c \in gen(w, \mathcal{L}), l : \\ \phi(l) = a \\ c \xrightarrow{l} c' \end{array} \right\}
\end{aligned}
$$

Proposition 2.1. For every $w \in \Sigma^*$, for every lpo $\mathcal{L}$, $c \in gen(w, \mathcal{L}) \iff c$ is generated by w in $\mathcal{L}$.

Proof:
The proof is by induction on the length of w and relies on the recursive definition of *gen*. If $w = \epsilon$, then, the only cut that it linearizes is $\emptyset$. This closes the basis case.

If $w = w' \cdot a$, then, the induction hypothesis is that $c \in gen(w', \mathcal{L}) \iff w'$ generates c. Pick an arbitrary cut c' in $\mathcal{L}$. If $c' = \emptyset$, then c' is generated by the empty suffix of w and $c' \in gen(w, \mathcal{L})$ by definition of $gen(w, \mathcal{L})$.

Consider the case $c' \neq \emptyset$. If $c' \in gen(w, \mathcal{L})$, then there is $c \in gen(w', \mathcal{L})$ such that $c \xrightarrow{l} c'$ and $\phi(l) = a$. By the induction hypothesis c is linearized by some suffix w'' of w'. Using Lemma 2.1 we obtain that c' is linearized by $w'' \cdot a$, which is a suffix of w.

The other direction is similar and also relies on Lemma 2.1. □

2.3.2. Size

The set $Gen(\mathcal{L}) = \{gen(\mathcal{L}, w) | w \in \Sigma^*\}$ is not completely chaotic. The various elements $\gamma \in Gen(\mathcal{L})$ are constructed in a disciplined way: all cuts belonging to γ are generated by a same finite word, say $w \in \Sigma^*$. From this, we can show that all cuts in γ are linearly ordered by set inclusion and obtain that Gen is exponentially smaller than $2^{2^{|\mathcal{L}|}}$, when $\mathcal{L}$ is deterministic. A deterministic lpo is an lpo in which identically labeled locations are ordered. Well-formed LSCs yield only deterministic lpo, because identically labeled locations may not belong to the same co-regions.

Definition 2.6. (Deterministic lpo)
We call an lpo $\mathcal{L} = \langle L, \phi, < \rangle$ *deterministic* iff $\forall l, l' \in L$: $\phi(l) = \phi(l') \implies l \leq l'$ or $l' \leq l$.

The name "deterministic lpo" comes from the fact that transition systems associated with these orders are deterministic.

Proposition 2.2. (Deterministic lpo)
Take an lpo $\mathcal{L} = \langle L, \phi, < \rangle$. This lpo is deterministic iff,

$$\forall c, c', c'' \in cuts(\mathcal{L}) : \forall l, l' : c \xrightarrow{l} c' \wedge c \xrightarrow{l'} c'' \implies (\phi(l) = \phi(l') \iff c' = c'')$$

Lemma 2.2. (Linear ordering of cuts)
Assume that $\mathcal{L}$ is deterministic. Then, for every $w \in \Sigma^*$, $gen(\mathcal{L}, w)$ is totally ordered by $\subseteq$.

Proof:
The proof is by induction on the length of w. For the induction base just note that $gen(\epsilon, \mathcal{L}) = \{\emptyset\}$, which is clearly linearly ordered by $\subseteq$ (since it is a singleton).

We turn to the inductive case. Based on our induction hypothesis that $gen(w, \mathcal{L})$ is totally ordered we show for an arbitrary $a \in \Sigma$ that $gen(w \cdot a, \mathcal{L})$ is also totally ordered. Let $\gamma_1', \gamma_2' \in gen(w \cdot a, \mathcal{L})$. If $\gamma_1' = \gamma_2'$ or if one of them equals the empty set, then they are clearly comparable w.r.t. $\subseteq$.

Hence, we can assume that $\gamma_1' \neq \gamma_2'$ and $\gamma_1' \neq \emptyset \neq \gamma_2'$. By definition of $gen(w \cdot a, \mathcal{L})$ there are $\gamma_1, \gamma_2 \in gen(w, \mathcal{L})$ and l_1, l_2 such that $\gamma_i' = \gamma_i \cup \{l_i\}$ and $\phi(l_i) = a$ ($i = 1, 2$). By induction hypothesis, $\gamma_1 \subseteq \gamma_2$ (the case $\gamma_2 \subseteq \gamma_1$ is symmetric). Consider the following two cases:

1. $\gamma_1 = \gamma_2$. Since $\mathcal{L}$ is deterministic and $\phi(l_1) = \phi(l_2) = a$, we get $\gamma_1' = \gamma_2'$.

2. $\gamma_1 \subset \gamma_2$. If $l_1 \in \gamma_2$, then $\gamma_1' = \gamma_1 \cup \{l_1\} \subseteq \gamma_2 \subset \gamma_2'$.

   If $l_1 \notin \gamma_2$, then, since for every $l' < l_1, l' \in \gamma_1$ and $\gamma_1 \subset \gamma_2$, it must be the case that $\forall l' < l_1, l' \in \gamma_2$. Thus, $\gamma_2 \xrightarrow{l_1} \gamma_2''$ for some γ_2'', by definition of $\longrightarrow$. Because $\mathcal{L}$ is deterministic and $\phi(l_1) = \phi(l_2)$, by hypothesis, we can apply Prop. 2.2 and obtain $\gamma_2 \cup \{l_1\} = \gamma_2'' = \gamma_2' = \gamma_2 \cup \{l_2\}$.

□

Lemma 2.3. (Number of chains)
For a set L of size n, define *Chains*$(L) \subset 2^{2^L}$ as $\{L' \in 2^{2^L} | \forall C, C' \in L' : C \subseteq C' \vee C' \subseteq C\}$. The size of *Chains*$(L)$ is $2^{O(n \log n)}$.

Proof:
Every $L' \in$ *Chains*(L) is of the form $\{C_1, \ldots, C_k\}$, with $C_i \subset C_{i+1}$ $(i = 1, \ldots, k-1)$, $C_k \subseteq L$ and therefore, $k \leq n$. L' can then be coded as a function, mapping every element x of L to the first j such that $x \in C_j$, or 0 if x does not appear in any C_j. There are at most $(n+1)^n$ such functions. Hence, $|$*Chains*$(L)| = 2^{O(n \log n)}$. □

Combining Lemmas 2.2 and 2.3, we obtain that, when $\mathcal{L}$ is deterministic, it is exponentially smaller than 2^{2^n}.

Proposition 2.3. For every deterministic lpo $\mathcal{L}$, $|Gen(\mathcal{L})| = 2^{O(n \log n)}$.

3. Liveness and safety

In this section, we show that universal scenarios boil down to two properties: a *safety* property, stating that bad things shall not happen, and a *liveness* property, expressing that good things shall eventually happen. In the context of trace theory, it is well-known that every property can be expressed as a boolean combination of liveness and safety [4]. Hence, our theorem does not come as a surprise. However, we will use the fact that, when dealing with universal LSCs, liveness and safety can be *shared* among the different events in Σ, yielding *one* safety and *one* liveness condition *per* event.

We skip the treatment of initial scenarios because they can be considered as a special case of universal charts, activated by a single event `start`, artificially added to occur once, at the beginning of every run.

Example 3.1. First, consider the sequence of events

$$w = \texttt{getWater} \cdot \texttt{pourWater} \cdot \texttt{getWater} \cdot \texttt{pourWater} \cdot \texttt{getWater} \cdot \texttt{pourWater}.$$

The cuts generated in the LSC of Fig. 1(d) by this sequence are displayed as thick dotted lines in Fig. 3(a). This example illustrates that, after some sequence of events, *several* cuts can be generated, because every suffix of w must be taken into account.

Now, consider the following sequence of events and the scenario of Fig. 1(a):

$$w' = \texttt{insertCoin} \cdot \texttt{askCocoa} \cdot \texttt{prepCocoa}.$$

After this sequence, a cut in the main chart is reached (see Fig. 3(b)). We know that every continuation of w' must include the occurrence of `serveCocoa`, in order to satisfy the LSC. In the vocabulary used below, we say that this event is *required*.

However, in these continuations, some events *must not* occur before `serveCocoa`, viz. those events that are restricted by this LSC but are not "next" events: $\{\texttt{insertCoin}, \texttt{askCocoa}, \texttt{prepCocoa}, \texttt{moneyBack}\}$. These events will be called *forbidden*. We will show below that runs in which required events eventually occur and in which forbidden events never happen are models of universal LSCs. In the rest of this section, we will formalize these notions.

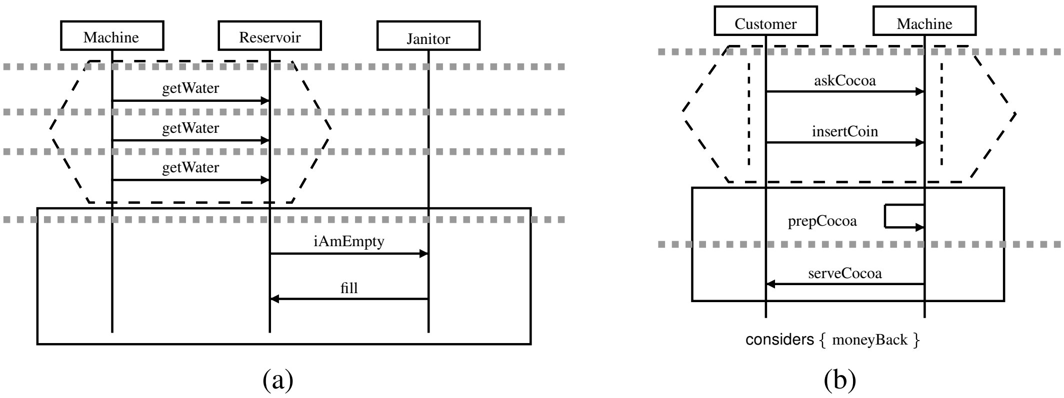

Figure 3. Cut in a chart

Definition 3.1. (Forbidden event, Safety)
Consider a specification $\mathcal{S}$ and a finite run $w \in \Sigma^*$. We say that this specification *forbids* $e \in \Sigma$ after w (*forbid*(w, e)), iff there is a scenario $\Box(C_1, C_2) \in \mathcal{S}$ such that $\exists c \in$ *gen*$(w, C_1 \cdot C_2)$ satisfying the following conditions:

1. $L_{C_1} \subseteq c \subset L_{C_1 \cdot C_2}$; the chart is active;
2. e is restricted by $\Box(C_1, C_2)$;
3. $\forall l \in L : \phi(l) = e \implies \nexists c' : c \xrightarrow{l} c'$.

A run $e_0 e_1 e_2 \ldots \in \Sigma^\omega$ is *e-safe* iff $\forall i : 0 \leq i :$ *forbid*$(e_1 \ldots e_i, e) \implies e \neq e_{i+1}$.

If we turn to the *obligations* imposed by a specification $\mathcal{S}$, one can make a similar characterization, in terms of cuts, of the events that are bound to occur in a given state.

Definition 3.2. (Required event, Liveness)
In $\mathcal{S}$, $w \in \Sigma^*$ is said to *require* $e \in \Sigma$ (*require*(w, e)), iff there is a scenario $\Box(C_1, C_2) \in \mathcal{S}$ such that $\exists c \in$ *gen*$(w, C_1 \cdot C_2)$, satisfying

1. e is restricted by $\Box(C_1, C_2)$;
2. $L_{C_1} \subseteq c \subset L_{C_1 \cdot C_2}$;
3. $\exists c' : \exists l : (c \xrightarrow{l} c') \wedge e = \phi(l)$.

A run $e_1 e_2 \ldots$ is *e-live* iff $\forall i : 1 \leq i :$ *require*$(e_1 \ldots e_i, e) \implies (\exists k : i \leq k : e = e_k)$.

Theorem 3.1. (Liveness + Safety = LSCs)
For every specification $\mathcal{S}$ containing only universal scenarios, and every set of infinite runs R

$$R \models_S \mathcal{S} \iff \forall r \in R : \forall e \in \Sigma : r \text{ is } e\text{-safe and } e\text{-live}$$

Proof:

The $\Rightarrow$ direction is easily shown. Decompose it into two assertions, use the definitions of satisfaction, liveness and safety and apply Lemma 2.1.

The $\Leftarrow$ direction is shown by contradiction. Assume that r is e-safe and e-live but $r \nvDash S$. From the latter, we deduce that there is some scenario $\Box(C_1, C_2)$ such that, there exist indexes i, k with the following properties:

$$r_i \dots r_k|_{\Sigma(S)} \text{ is a linearization of } C_1$$

and

$$\nexists j \geq k : \begin{cases} \phi_2(l_1) \dots \phi_2(l_n) = r_k \dots r_j|_{\Sigma(C_2)} \\ \emptyset \xrightarrow{l_1} c_1 \xrightarrow{l_2} \dots \xrightarrow{l_n} L_2 \end{cases}$$

with L_2 being the set of locations and ϕ_2 the labeling function of C_2. Remark that, since $\Box(C_1, C_2)$ is deterministic, there is at most one cut reached on every $r_k \dots r_j$. Furthermore, for every j, either there is no cut reached on $r_k \dots r_{j+1}|_{\Sigma(C_2)}$ or the cut reached on $r_k \dots r_{j+1}|_{\Sigma(C_2)}$ is bigger, wrt set inclusion, than the cut reached on $r_k \dots r_j|_{\Sigma(C_2)}$. Find the smallest $j*$ such that

1. $r_k \dots r_{j*}|_{\Sigma(C_2)}$ reaches $c* \subset L_2$,
2. $\forall j \geq j* : r_k \dots r_j|_{\Sigma(C_2)}$ reaches c implies that $c = c*$.

Clearly, $j*$ must exist, because the sequence of cuts is monotonically increasing, over a finite domain, wrt the following order: $c \leq c'$ if c' is empty or c and c' are nonempty and $c \subseteq c'$.

Notice that, because $c* \subset L_2$, there must exist some e such that

- $c* \xrightarrow{l} c' \subseteq L_2$
- $\phi_2(l) = e$.

Now, we perform a case split:

$\exists j > j* : r_j \in \Sigma(C_2)$: since $\forall l : c* \xrightarrow{l} c' \subseteq L_2 \implies \phi_2(l) \neq r_j$, the run is not e-safe. Otherwise, we would have found a cut which is greater than $c*$.

$\forall j > j* : r_j \notin \Sigma(C_2)$, in particular, $\forall j > j* : r_j \neq e$, which implies that the run is not e-live.

$\Box$

4. Consistency and Synthesis

4.1. Rationale

In software engineering, a distinction must be made between what can be expected from the environment, and what must be guaranteed by the SUD. Here, the SUD will be delineated as a set of agents. As stated by theorem 3.1, LSCs impose safety and liveness conditions for each event, and each event is controlled by its sender. These conditions are thus assumptions when the agent belongs to the environment, and have to be guaranteed by the SUD when the agent belongs to the SUD.

For example, the `Janitor` agent in our running example must fill the `Reservoir` when it is empty, Fig. 1(d). This is therefore an assumption on the environment behavior.

Considering the distinction between the system and its environment some specifications are problematic, because it is impossible to implement them relying solely on the assumptions provided on the environment behavior. There is a well-behaving environment forcing any computer program to violate the system's constraints. Our goal is to detect such inconsistent LSCs specifications.

For instance, in our running example, it is impossible to build a program for `Machine` which would respect the specification of Fig. 1. Indeed, even though this specification is *satisfiable*, i.e. there is some run satisfying it, for example:

$$(\texttt{insertCoin} \cdot \texttt{claimMoney} \cdot \texttt{moneyBack})^\omega,$$

the following prefix leads the system to a deadlock:

$$\texttt{insertCoin} \cdot \texttt{claimMoney} \cdot \texttt{askCocoa} \cdot \texttt{prepCocoa} \cdot \texttt{addCocoa} \cdot \texttt{getWater} \cdot \texttt{pourWater}.$$

In this case, the scenario of Fig. 1(a) imposes that `serveCocoa` happens, but forbids `moneyBack`, while Fig. 1(b) imposes the reverse constraints. Since the customer is not within the border the SUD, and thus not under its control, it is able to input $\texttt{giveMoney} \cdot \texttt{claimMoney} \cdot \texttt{askCocoa}$ and crash whatever implementation of `Machine`. Notice that this behavior of `Customer` does not violate any assumptions about him. A specification is realizable [2] if there exists a program fulfilling it, without having to restrict its environment beyond the given assumptions. This realizability issue arises because of several interacting and overlapping scenarios. These scenarios may state inconsistent requirements about the behavior of the system. Our goal is to provide a tool for discovering these inconsistencies. This tool proves the implementability of specifications in a constructive manner: if the LSCs are realizable, a possible implementation is synthesized.

4.2. Problem Statement

Assume that *Ag* is partitioned into two teams: E and S, the environment and the system. We start by determining which constraints $\mathcal{S}$ puts on the environment.

Definition 4.1. ($\mathsf{WB}(\Sigma_E)$)
The environment will be *well-behaving* in a run r iff

$$\forall e \in \Sigma_E : r \text{ is } e\text{-safe and } e\text{-live}$$

The set of all runs in which the environment is well-behaving is denoted $\mathsf{WB}(\Sigma_E)$.

Following [43], a program will be modeled by a strategy. A strategy is a function telling which actions the system should make, depending on the past. Formally, a strategy for the system is a function $f : \Sigma^* \to {\Sigma_S}^*$. Given any finite run r, $f(r)$ determines what the answer of the system will be. We view the game between the system and its environment as turn-based. The environment submits a sequence of events e, then the system answers according to f, i.e. performs $f(e)$. When this sequence finishes, the environment performs a new sequence e'. Then, the system answers by following a sequence of steps $f(e \cdot f(e) \cdot e')$. The decision to model the game as turn-based ensures game determinacy [41].

This turn-based execution is similar to the typical execution of many reactive systems, where the system is infinitely faster than its environment. It follows the *super-step* execution presented in the asynchronous time scheme execution of statecharts [27]. A super-step is "(...) a series of steps, initiated by external changes and proceeding until reaching a stable status."

Our model allows the environment to submit *several changes* simultaneously, by performing a sequence of events "at once".

The *outcome* of a strategy is the set of runs that are induced by a strategy f. Formally,

$$Out(f) = \{u_0 w_0 u_1 w_1 \dots | \forall i \geq 0 : u_i \in {\Sigma_E}^+ \text{ and } w_i = f(u_0 w_0 \dots u_i)\}.$$

Definition 4.2. (Consistency)
A specification $\mathcal{S}$, containing only universal scenarios, is *consistent* iff

$$\exists f : \forall w \in Out(f) : \begin{cases} w \text{ is } \Sigma_E\text{-safe} \implies w \text{ is } \Sigma_S\text{-safe} \\ w \text{ is } \Sigma_E\text{-live} \implies w \text{ is } \Sigma_S\text{-live,} \end{cases}$$

wrt $\mathcal{S}$.

If such a strategy f exists, Theorem 3.1 guarantees that every execution induced by f in which the environment assumptions are met satisfies the specification,

$$Out(f) \cap \mathsf{WB}(\Sigma_E) \models \mathcal{S}.$$

We prefer our consistency condition which relates environment safety to system safety, and liveness assumptions to liveness commitments, to the condition

$$\Sigma_E\text{-safe} \wedge \Sigma_E\text{-live} \implies \Sigma_S\text{-safe} \wedge \Sigma_S\text{-live}.$$

This condition would allow the system to make unsafe moves disabling the possibility for the environment to be live. We give an example of this. We are asked to build a controller for a car lift. The controller can set the engine to three positions: up, neutral or down. It is expected that, when the engine is on position "down", the car lift will be going downwards. This is a liveness assumption on the environment. However, we require that the engine must be on "neutral" position when the brakes are on. Otherwise, the transmission breaks down and the car lift can never move. It is possible to design an implementation fulfilling the condition above, yet naturely incorrect. The controller first breaks the transmission by setting the brakes on and putting the engine to position "up"'. Then, it sets the engine to position "down". Since the environment cannot fulfill its liveness constraints (the car lift cannot go down, for the transmission is broken), the condition above is vacuously verified, hence the implementation is correct. Our condition in Def.4.2 is stronger and rules out that situation. Def 4.2 clearly separates constraints that can be finitely falsified (safety) from conditions that can only be falsified by infinite runs (liveness) [4].

4.3. Algorithm

We propose an algorithm for checking that a universal LSC specification $\{S_1, \dots, S_n\}$ is consistent. Assume that an arbitrary order is given to the symbols in Σ_S and Σ_E.

First of all, we define a transition system for universal scenarios.

Definition 4.3. ($\gamma \stackrel{e}{\Longrightarrow} \gamma'$)
Consider a universal LSC $S = \Box(C_1, C_2)$. We let $Gen(S) = Gen(C_1 \cdot C_2)$. Then, for every γ, γ' in $Gen(S)$ and every $e \in \Sigma$, $\gamma \stackrel{e}{\Longrightarrow} \gamma'$ iff

- e is not restricted by S and $\gamma = \gamma'$,
- or e is restricted by S and
  - $\forall c \in \gamma : L_{C_1} \subseteq c \subset L_{C_1 \cdot C_2} \implies \exists c' : c \stackrel{l}{\longrightarrow} c',$ with $\phi(l) = e$
  - and $\gamma' = \{\emptyset\} \cup \{c' | \exists c \in \gamma : c \stackrel{l}{\longrightarrow} c',$ with $\phi(l) = e\}$.

By inspection, one can show that

Proposition 4.1. $\forall w \in \Sigma^* : \forall \gamma : \{\emptyset\} \stackrel{w}{\Longrightarrow} \gamma$ iff w is Σ-safe and $\gamma = gen(w, C_1 \cdot C_2)$.

In order to encode the turn-based approach presented above, we add two dummy events: τ_0 and τ_1. The environment is asked to mark the end of their finite sequence of events by τ_1. The system should do so with τ_0. So, a run $u_0 w_0 u_1 w_1 \ldots$, with $u_i \in {\Sigma_E}^+$ and $w_i \in {\Sigma_S}^+$ will be encoded as $u_0 \tau_1 w_0 \tau_0 u_1 \tau_1 w_1 \tau_0 \ldots$. Then, we add two scenarios, in order to keep players from providing infinite sequences of inputs: τ_0 and τ_1 shall occur infinitely often. Each player is thus responsible for his τ event.

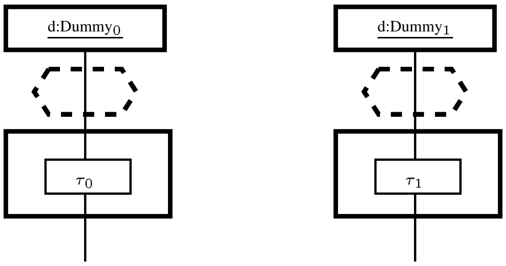

Figure 4. Fairness scenarios

We build a *game graph* [23], i.e. a graph of the form

$$G_{\mathcal{S}} = \langle V, V_0, \Delta, \Omega \rangle,$$

where

- V is a set of vertices, defined here as

$$V = (\{0,1\} \times \Sigma \times \mathit{Gen}(S_1) \times \ldots \times \mathit{Gen}(S_n) \times [|\Sigma_S| + 1] \times [|\Sigma_E| + 1]) \cup \{\mathit{sink}_0, \mathit{sink}_1\}.$$

  A vertex of the form $(i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1)$ has the following meaning:

  - i is a flag indicating whether player 0 (i.e. the system) or player 1 should play the next move;
  - e is the last event that has been done.

- $\gamma_j (1 \leq j \leq n)$ is the set of cuts that are currently active in S_j.
- c_0 is the index of the next event in Σ_S for which the liveness condition must be verified. Intuitively, we will consider that this event has fulfilled its condition if it occurs next, or it is not required in the current state.
- c_1 is the same as c_0 for Σ_E events.

The *sink*$_i$ state means that player i has violated one of its safety conditions.

- V_0 are vertices belonging to player 0.

$$V_0 = \{\mathit{sink}_1\} \cup \{(i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1) | i = 0\}$$

- Δ is a transition relation, we define it as follows:
  - The move does not violate any safety condition:

$$\Delta((i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1), (i', e', \gamma'_1, \ldots, \gamma'_n, c'_0, c'_1))$$

  if
  1. $i = 1 - i$ iff $e' = \tau_i$;
  2. $e' \in \Sigma_S$ iff $i = 0$;
  3. $\gamma_j \overset{e}{\Longrightarrow} \gamma'_j$, $(1 \leq j \leq n)$;
  4. if $c_0 = |\Sigma_S| + 1$, $c'_0 = 1$, otherwise, assuming that $a \in \Sigma_S$ is the c_0-th symbol, then, if $e' = a$ or $\forall j : 1 \leq j \leq n : \gamma_j$ does not require a, $c'_0 = c_0 + 1$, else $c'_0 = c_0$.
  5. c'_1, cfr. c_0, *mutatis mutandis*.
  - The move violates a safety condition:

$$\Delta((i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1), \mathit{sink}_i)$$

  if, in the case where $i = 0$, for some $e' \in \Sigma_S$, $\gamma_j \overset{e'}{\Longrightarrow} \gamma'_j$ is not defined. The case for $i = 1$ is identical, except that $e' \in \Sigma_E$.
  - It is impossible to leave a sink state:

$$\Delta(\mathit{sink}_i, \mathit{sink}_i)$$

- $\Omega \subseteq V^\omega$ is a set of winning runs for player 0. Here we use a Streett condition, with only one pair. A Streett condition is a set of pairs of sets of states: $\{(E_i, F_i)\}$. This condition accepts a run if, for every i, if E_i is visited infinitely often, then so is F_i. Here, we have

$$\Omega = \text{Streett}(\{(E, F)\})$$

  with

  - $E = \{(i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1) | c_1 = |\Sigma_E| + 1\} \cup \{\mathit{sink}_0\}$,

- $F = \{(i, e, \gamma_1, \ldots, \gamma_n, c_0, c_1) | c_0 = |\Sigma_S| + 1\} \cup \{sink_1\}$. This encodes the consistency condition defined above: if the game goes infinitely often in states of E, where the liveness of the environment is ensured, then the system must do so, too.

We only need to relate the consistency of $\mathcal{S}$ to the existence of some winning strategy on $G_{\mathcal{S}}$.

Proposition 4.2. The specification $\mathcal{S}$ is consistent iff

$$\exists f : f \text{ is winning on } G_{\mathcal{S}} \text{ from } (1, e, \{\emptyset\}, \ldots, \{\emptyset\}, 1, 1), \quad \text{for any } e \in \Sigma$$

We turn our Streett game graph with a single pair into a parity game graph, with 3 colors. As a reminder, a parity game graph is a game graph with a mapping $\Omega : V \to [k]$, for some $k \in \mathbb{N}$. A play is winning in such a graph if the maximal color occurring infinitely often in this play is even. In our situation, we choose the following coloring:

- $\Omega(v) = 2$, if $v \in F$;
- $\Omega(v) = 1$, if $v \in E \setminus F$;
- $\Omega(v) = 0$, otherwise.

Thus, one can reuse existing algorithms for solving parity games [23]. We have implemented our algorithm relying on the algorithm of [57].

4.4. Complexity

The consistency problem is both co-NP-hard and NP-hard.

We reduced 3-SAT to the "negative" problem of LSC consistency ("is the specification inconsistent?"). For every propositional variable p, there are two messages p and $\neg p$. The environment controls these messages and hence "chooses" valuations. For each clause C, there are also two messages C and $\neg C$, which are controlled by the system. The proper evaluation of these clauses is encoded in LSCs. Roughly, when the three literals of C have received a bad value, the system triggers $\neg C$, whereas it emits C if one of those literals is correctly valuated. The formula will thus be evaluated to true when the system will have matched all clauses, in *any* order. When this happens, the specification dooms the system to a deadlock situation, in which two events are both required and forbidden. The specification is thus *not* consistent when the formula is satisfiable.

The reduction of 3-SAT to the positive consistency problem works as follows: we require the system to provide a satisfying valuation. In that case, the environment is driven into some deadlock thus, ensuring that every possible outcome is not Σ_E-live and trivially matching the consistency condition. The trick is that the scenarios of the specification are such that there are no possible Σ_S-live runs. Hence, the system will lose if it cannot find a good valuation for the formula. Thus, the specification is consistent iff there is a strategy for the system providing a correct valuation for the formula.

Solving parity games with 3 colors requires only polynomial time, in the size of the game graph. Since $G_{\mathcal{S}}$ is defined as the product of generated cuts and $\Sigma_E = \Sigma \setminus \Sigma_S$, we can deduce that the running time of our algorithm is

$$O\left(|\Sigma|\,|\Sigma_S|\,\left(|\Sigma| - |\Sigma_S|\right)\,2^{s\,n\,\log n}\right)$$

where s is the number of scenarios in the specification and n is the maximal number of locations appearing in a chart. This relies on Prop. 2.3.

4.5. Related work

There has been much work in synthesis of concurrent reactive systems. The initial question is due to Church [14] and has been solved by Büchi and Landweber in [13]. Pnueli and Rosner describe how to synthesize reactive modules from LTL specifications, either in a synchronous or asynchronous setting [46]. They reduce the realizability problem for LTL to the satisfiability problem for CTL*. Basically, the set of variables occurring in an LTL formula is partitioned between environment variables (say, x) and system variables (say, y). The full computation tree is a tree such that, for every x value, each node has a successor "corresponding" to x. Formally, if X is the domain of x, the full tree on X is X^*. The problem amounts to finding such a tree whose nodes are labeled by y values, such that every path in the tree fulfills the given LTL formula.

This work is extended to cope with partial information, generalized parallelism and real-time systems by [60]. The approach is trace-based and the realizability problem is exactly given as here: find a strategy for the program such that whatever strategy the environment chooses, the specification is met. Again, Rabin tree automata are at the heart of the approach for the finite-state case, together with Safra's determinization procedure [47]. This makes the procedures hard to implement.

We have shown how to decide the realizability property for LSCs, using existing game theoretical algorithms and automata-theoretic techniques. Our solution, for the particular case of LSCs, is more interesting than first translating LSCs to LTL [11] and then using the procedure of [43] to check its realizability, because it is more direct. In particular, we have shown that our solution is simply exponential. If we first translate LSCs to LTL, we will have to build an acyclic automaton, which already contains exponentially more states than the LSC. Then, this automaton will be traversed to construct an LTL formula [11]. Here, we build exactly the same automaton but directly use it to solve the problem in which we are interested.

The so-called "supervisory control synthesis" problem for Discrete Event Systems (DES) focuses on the controllability of finite languages, generated by a plant [45]. Basically, a plant generates some prefix-closed language L, a subset of which is marked, L_m. We are given some language $K \subseteq L_m$. Then, we would like to know whether we can build a strategy, disabling some transitions of the plant, such that the marked "closed-loop" behavior of the plant is K. The computational complexity of this problem makes it difficult to apply practically, hence horizontal and vertical decomposition have been devised as means to make solutions more tractable. In [48], a survey of this area is presented. The problem of DES supervisory control has been extended to real-time [38], branching-time specifications [8], infinite behaviors [50, 51] and partial information/decentralized supervision for which results are mostly negative [53, 37, 44]. In spirit, our solution is not too different from the one presented in [50, 51]. The basic difference between our work and theirs is that we do not require the "plant" and the "legal language" to be explicitly defined. On the contrary, they are described in a succinct and intuitive way, using LSCs. Actually, we could have reduced our problem to a supervisory control problem for DES, but we chose to follow a reduction to game solving problems, with which we are more familiar and, we believe, are easier for the reader to grasp. The two areas are closely related,differing mostly in terms of vocabulary. Here, we deal with pure liveness assumptions. These assumptions can be encoded in the plant as in [50]. Because of the restricted form of our acceptance condition, we have an efficient algorithm (wrt the size of the plant). Another approach to obtain efficient algorithms, as in [49], is to add a strong fairness assumption, which makes the synthesis of controllers for Rabin automata computable in polynomial time, modulo a modification of the synthesis routine.

Our solution synthesizes a global strategy, for all agents of the reactive systems. Nevertheless, it seems that practically, synthesizing a distributed implementation would be of greater interest. It turns out that the realizability problem over fixed architectures is undecidable for almost every interesting architecture [46]. Rosner presented a type of architecture for which the problem becomes decidable, which he calls *hierarchical architectures*. In this architecture, one agent receives all inputs and the information flow between agents induces a tree, hence the term "hierarchical". Observing that decidability in this case comes from the fact that the "root" agent can always simulate the behavior of every other agent, Kupferman and Vardi came with other architectures for which realizability is also decidable [35]. These architectures simply require the agents to be ordered either linearly or in cycle. More generally, Madhusudhan and Thiagarajan present three properties making the distributed controller synthesis problem decidable [39]. These restrictions are: trace-closure of specification, strategy must be com-rigid and clocked. As soon as one of them is dropped, the problem becomes undecidable. LSC specifications are not trace-closed.

Facing this undecidability result, one could tackle the "distribution" problem by first synthesizing a global strategy and then trying to distribute it over the various agents [40, 21]. This technique is sound and not complete, in the sense that there might exist some distributed implementation even though the centralized solution is not distributable.

In the realm of scenarios, previous research on "synthesis" followed three directions: induction, compilation and controller synthesis. Every approach finds a justification, depending on the purpose of the scenarios and the phase of the software life cycle in which it is used [7, 58].

Induction: Typically, scenarios are considered as a partial view on the behavior of the future system. A set of scenarios is thus a finite set of example computations. The problem is to induce, from these examples, a universal rule describing every acceptable behavior of the system. The user has some (regular) set of behaviors W in mind and gives a finite set of examples $E \subseteq W$. We are asked to build an automaton $\mathcal{A}$ recognizing W. [33] have used an algorithm, due to Biermann and Krishnaswamy [10], which computes the "best" deterministic $\mathcal{A}_E$. At the limit, the language of the synthesized automata converges to W. Practically, this automaton needs to be checked, in order to ensure that it does not contain inappropriate behaviors. It has been integrated in the CASE tool FUJABA (From UML to Java and Back Again) [19].

Hsia *et al.* use example scenarios, given as an execution tree, to build a grammar and analyze the specification, wrt consistency criteria, using automata analysis techniques [28] . By hand, they produce, from such a set of examples, a prototype of the future system. This prototype can then be used to validate the specification with the end-user.

Compilation: Scenarios can also be seen as a *complete* description of the future system. Thus, the desired behavior W equals the given traces E. This is especially useful if the system to be built is closed. Then, state machines must be built from the various scenarios, projecting them onto every instance [34, 59, 54, 5]. The main problem that arises then is that there might be some discrepancies between the global view of the behavior, induced by the scenario and the local view that every instance has of this behavior. When recomposing the full system from the individual state machines, yielding a set of behaviors C, it might be that $C \supset W$. These additional scenarios, $C \setminus W$, are called "implied scenarios" [5]. Techniques have been developed to detect such implied scenarios and report on them [55, 5]. In [22], the authors give syntactical restrictions, namely *causality*, which ensures that MSCs, with control flow constructs, such as iteration or choice, can be distributed, i.e. that the liveness/safety constraints imposed by lifelines match the global constraint stated by the considered MSC.

One result, which is in spirit close to ours, is due to Desharnais *et al.* [18], except that their setting is state-oriented, whereas most researchers on scenarios, including us, focused on an event-based setting. In [18], the authors present a way to represent scenarios as a relation between states. This presentation can be graphical, thanks to relational transition systems. They make a distinction between environment moves and system moves, allowing moves "within" the environment and "within" the system, as we do it. A scenario is assumed to describe possible environment inputs and all legal system reactions. The authors propose an operator for integrating scenarios, based on the "demonic meet" operator. Like in our work, the integration of two scenarios relative to the same input obliges the system to answer as specified by both scenarios.

Controller synthesis: this approach corresponds to what we have proposed here. One uses an expressive scenario-based specification language and then tries to synthesize a program satisfying it. LSC is so far the only candidate language for this approach, thanks to its higher expressiveness.

The main work in this field is due to Harel and Kugler [24]. They deal with very simple LSCs: precharts contain only one environment message ($\in A_{in}$) and main charts contain only system messages ($\in A_{out}$). They choose the super-step approach of [27]: the environment provides one input and the system answers with a sequence of output messages: $(A_{in}A_{out}^*)^\omega$. They show that realizability is equivalent to a *consistency* condition. This condition asserts the existence of a nonempty regular language $L \subseteq (A_{in}A_{out}^*)^*$ such that

1. L contains one execution for every existential chart,
2. L satisfies all universal charts,
3. for every $w \in L$ and every $a \in A_{in}$, there is some $r \in A_{out}^*$ such that $war \in L$.
4. for every $xyz \in L$ such that $y \in A_{in}$, $x \in L$.

They build a minimal deterministic automaton recognizing the intersection of universal LSCs and progressively prune it to remove "bad states", i.e. states that do not satisfy condition (3). The resulting automaton can then be transformed to some strategy automaton. Our solution is an extension of their work. Firstly, we tackle the problem for more general LSCs, allowing choice constructs, complex precharts and environment messages in the main chart. The latter extension enabled us to use LSCs in assume/guarantee development of reactive systems. Secondly, instead of relying on an ad-hoc technique, we proposed a reduction to parity games, for which a range of results and algorithms is available.

They also propose three strategies to distribute the synthesized strategy among the various agents:

1. build a central controller, driving the execution of the individual agents;
2. duplicate the central controller in every agent;
3. duplicate the central controller and remove states that are not relevant to the object in question.

Harel and Marelly have developed a technique called "play-out" for executing LSC specifications [26]. In this approach, the interpreter (called play-engine) follows a built-in strategy: it selects (in a predefined way) a required event and performs it. Of course, this strategy can lead to deadlocks. To solve this problem, Harel *et al.* have developed a "smart play-out" algorithm [25]: they use model-checking to compute a "non-blocking" strategy. This strategy proposes a sequence of events which leads to a state in which there are no more required events, for the system, if such a sequence exists. However, even though this approach ensures that a successful "super-step" s starting at state q will be found if it exists, it might be that, for some environment input, say i, there is no super-step starting at state q' with $q \xrightarrow{s \cdot i} q'$. This shows an advantage of our method, which ensures that such situations will be avoided, at the price of a higher complexity.

In our work, we *verify* that a specification is consistent, by performing a full search on the state space, while Harel and Marelly use play-out to *validate* a specification, by letting a user execute it. Our work could be integrated with theirs: if the specification is inconsistent, our algorithm builds a *counter-strategy*, i.e. a strategy for the environment, making the system fail. This strategy could be given to the play-out engine to illustrate the flaws of the LSC specification.

We adopted the popular *super-step* approach, in order to model interactions between a system and its environment. However, we plan to consider other interaction models, based on real-time games, as they provide a more intuitive and practical approach [3]. Other approaches, like those of synchronous languages [9], can be simply integrated in our model, by changing the conditions on the occurrence of τ_0 and τ_1 events.

Our solution has been extended in [12], to integrate choice constructs, improving the expressiveness of the language.

5. Mercifulness

Streett pairs have been used to encode games with assume/guarantee principle. We used Streett pairs (E, F) to describe, in the E part, the assumptions that the environment has to fulfill, and, in the F part, the properties the system has to ensure once they have been met.

This approach is common in software engineering, where components are never developed nor deployed in isolation. Rather, some hypothesis about their environment are needed. In the transformational paradigm they are usually given as preconditions. In the reactive world, they must be stated as assumptions on the behavior of the environment. The specification was stated to be consistent if there was some strategy f such that $\forall r \in Out(f) : \inf(r) \cap E \neq \emptyset \implies \inf(r) \cap F \neq \emptyset$.

However, it is possible in some cases that the SUD is implemented by a strategy which does not allow any run satisfying E: $\forall r \in Out(f) : \inf(r) \cap E = \emptyset$. Then, the specification is consistent. However, there might be some other possible program in which E might be infinitely often visited. Intuitively, it seems that a strategy preferring runs visiting E finitely often over runs visiting both E and F infinitely often should be ruled out. Indeed, we do not want the system to use such a strategy when a more "merciful" solution exists.

In this section, we introduce the notion of mercifulness, which rules out such problematic specifications and solutions. This property is not a linear property anymore; it is branching and cooperative. This implies that we do not require from the synthesized system to force E to be visited infinitely often but only to leave the opportunity to visit E infinitely often provided the environment cooperates to

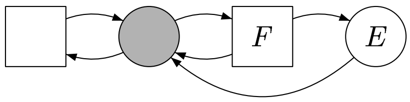

Figure 5. Merciful game

achieve this goal. Mercifulness is weaker than machine closure of environment assumptions [1], in which one requires that the environment may *force* every run to be well-behaved. Some specifications present assumptions that are not machine-closed, although they have a merciful strategy implementing them.

Consider the game graph of fig. 5, where player 0 vertices are displayed as circles. Every state is winning. Consider the gray vertex. If the strategy of player 0 is to always move left, he will win, as no E state will be visited infinitely often. This strategy is merciless: although there is a path visiting both E and F infinitely often, it forces the play not to visit E. A strategy that always moves right is merciful: it gives the opportunity to player 1 to visit E.

In this section, we consider a game played on a game graph $G = \langle V, V_0, \Delta, \Omega \rangle$, with

- $V_1 = V \setminus V_0$;
- $\Delta \subseteq (V_0 \times V_1) \cup (V_1 \times V_0)$ and $\forall v \in V : \exists v'' \in V : \Delta(v, v')$, i.e. we assume that the initial game graph is bipartite and that there is no dead end;
- $\Omega : \mathit{Streett}(\{(E, F)\})$

Requiring game graphs to be strictly alternating between players is done without loss of generality. Idling steps can always be added to meet this requirement.

By $\Pi_G(v)$, we will denote the set of all infinite paths in G that are rooted in v:

$$\Pi_G(v) = \{v_0 v_1 \ldots | v_0 = v \wedge \forall i : i \geq 1 : \exists e \in \Sigma : (v_{i-1}, e, v_i) \in \Delta\}.$$

For a strategy σ, a σ-prefix is a finite word $w \in V^*$ such that w is the prefix of some outcome of σ. A strategy is called merciful if for every σ-prefix that admits an infinite extension that visits E infinitely often there exists such an extension that is played according to σ. This is captured by the following definition, where $\mathit{last}(w)$ denotes the last vertex of w.

Definition 5.1. (Merciful strategy)
A strategy σ is said to be *merciful* iff, for every σ-prefix $w \in V^*$,

$$\begin{gathered}\exists w' \in \Pi_G(\mathit{last}(w)) : \inf(w \cdot w') \cap E \neq \emptyset \\ \Rightarrow \\ \exists w' \in \Pi_G(\mathit{last}(w)) : \inf(w \cdot w') \cap E \neq \emptyset \wedge w \cdot w' \in \mathit{Out}(\sigma).\end{gathered}$$

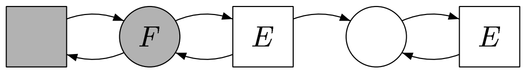

Figure 6. Every winning state is not merciful

This definition can be expressed in the *Game Logic* of [6]. The model-checking problem for this logic is decidable and relies on iterated applications of the module-checking procedure of [36]. However, this procedure requires doubly-exponential time. Here, we exhibit a direct solution to the problem of solving merciful games which causes only a linear blow-up in the size of the game graph. For the particular problem of one Streett pair merciful games, the solution is polynomial, as the number of colors of the underlying game graph is fixed to four. We also give an upper bound on the amount of memory needed to win merciful games, viz. two bits. We did not want to build a new routine for synthesizing strategies. Instead, we reduce the problem of synthesizing merciful strategies to the problem of solving standard parity games. This will make it possible to investigate the relationships between these two types of games.

A state is mercifully winning if there is a merciful winning strategy starting from that state. The merciful winning region is the set of states from which player 0 can use a merciful strategy and win, with respect to the Streett condition. We will say that we solve a merciful game if

1. We decide membership of states to merciful winning regions,

2. We construct a merciful strategy winning from each of these states.

By definition, the merciful winning region is included in the winning region and one could wonder whether the two regions do coincide. This would solve trivially the first part of the problem but as shown in Fig. 6, winning regions and merciful winning regions do not coincide. Player 0 vertices are displayed as circles. The winning states for player 0 are the gray states. However, there are no merciful winning vertices in this game: as soon as player 0 chooses to move right and let his opponent meet an E state, player 1 can force the game to enter the rightmost vertex, and win. Any winning strategy is thus merciless.

Furthermore, merciful strategies are more complex than simple winning strategies. In order to solve merciful games, some memory is needed. Consider the game of Fig. 7. Player 0 needs some memory, to remember to alternate between E and F states. Otherwise, his strategy would be either merciful and losing or merciless and winning. Using a graph-based technique to solve these games allows us to quantify precisely the amount of memory needed to be merciful: two bits.

Given a game graph G, we define an extended game graph G'. It is a standard parity game graph, "simulating" G. A vertex in G is mercifully winning iff it is winning in G', according to the standard parity condition.

To simplify the construction of G' we will assume that from each vertex $v \in V$ there exists a path in $\Pi_G(v)$ that visits E infinitely often. Vertices that do not satisfy this condition obviously are in the

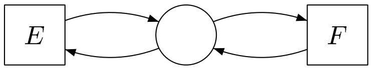

Figure 7. Mercifulness requires memory

merciful winning region of player 0 and can easily be detected by computing the strongly connected components of the game graph. Hence, a vertex is mercifully winning if we can find a strategy which (1) is winning and (2) always contains at least one path visiting E infinitely often.

G' is constructed by introducing two copies of G that correspond to two different games

the *play* **game:** this is the usual game. Player 0 tries to fulfill the Streett condition.

the *show* **game** is entered by player 1. Player 1 challenges player 0 to find a path, starting at the current vertex and visiting E infinitely often. Of course, this path must visit F infinitely often, too. To prove that such a path exists, player 0 will play twice in a row: first, he plays on behalf of player 1 and then plays his own move. Player 1 can then decide to switch back to the original game. This accounts for the fact that merciful winning strategies must also be winning in the original game.

In order to prove that this reduction enables us to find merciful winning states, we will need to show that (1) whenever player 0 has a winning merciful strategy from v on G, there is a winning strategy from (play, v) on G' and (2) whenever player 0 has a winning strategy from (play, v), he also has a winning merciful strategy from v. Both proofs are constructive: a mercifully winning strategy σ on G will be translated to a strategy σ', winning on G', and vice-versa. The main difficulty in the proof of (1) is to ensure that E will be visited infinitely often when the play remains almost forever in *show* vertices. From a winning strategy σ' on G', we will need to extract a winning merciful strategy σ on G. Since every vertex in G is duplicated in G', at a vertex v, player 0 has to decide whether σ must behave as σ' defined on *play* vertices or on *show* vertices. These two cases depend on the behavior of player 1. If player 1 makes a move in G from vertex v which corresponds to the move of σ' from vertex (show, v), then from now on, σ will simulate σ' as if the play was in the *show* part of G'. If player 1 does not follow σ', then σ simulates σ' on the *play* part of G. Now that the intuition of the construction has been presented, we will formalize it and prove its correctness.

If we construct G' as described above, then the required winning condition cannot be expressed as a parity condition. We need to introduce one additional bit that "memorizes" visits of the sets E and F. Formally, $G' = \langle V', V'_0, \Delta', \Omega' \rangle$ is defined as follows.

- $V' = V \times \{\mathit{play}, \mathit{show}\} \times \{0, 1\}$.
- $V'_0 = (V_0 \times \{\mathit{play}\} \times \{0, 1\}) \cup (V \times \{\mathit{show}\} \times \{0, 1\})$.
- For all $q, q' \in V'$, $\Delta'(q, q')$ if one of the following statements holds.

1. $q = (u, \mathit{play}, i)$, $q' = (v, \mathit{play}, \kappa(i, u))$, and $\Delta(u, v)$,
2. $q = (u, \mathit{play}, i)$, $u \in V_1$, and $q' = (u, \mathit{show}, i)$,
3. $q = (u, \mathit{show}, i)$, $u \in V_1$, $q' = (v, \mathit{show}, \kappa(i, u))$, and $\Delta(u, v)$, or
4. $q = (u, \mathit{show}, i)$, $u \in V_0$, $q' = (v, \mathit{play}, \kappa(i, u))$ and $\Delta(u, v)$,

where

$$\kappa(i, u) = \begin{cases} 0 & \text{if } u \in F \text{ and } i = 1, \\ 1 & \text{if } u \in E \text{ and } i = 0, \\ i & \text{otherwise}. \end{cases}$$

- $\Omega : V' \to \{1, 2, 3, 4\}$ is defined as follows

  1. $\Omega((u, x, 0)) = \begin{cases} 2 & \text{if } x = \mathit{play} \wedge u \in V_0 \\ 1 & \text{otherwise} \end{cases}$
  2. $\Omega((u, x, 1)) = \begin{cases} 4 & \text{if } u \in F \\ 3 & \text{if } u \in E \setminus F \\ 2 & \text{if } x = \mathit{play} \wedge u \in V_0 \setminus (E \cup F) \\ 1 & \text{otherwise} \end{cases}$

If we project a sequence of vertices from V' to its first component we get a sequence of vertices from V. Whenever we write that such a sequence of V' vertices satisfies some condition related to the sets E and F, then we refer to this projection.

The following lemma relates the winning condition of G to the winning condition of G' and can be shown by a simple case distinction using the definition of G'.

Lemma 5.1. A path w in G' is winning for player 0 iff one of these conditions holds:

1. w visits infinitely many states in $V_0 \times \{\mathit{play}\} \times \{0, 1\}$ and satisfies the Streett condition (E, F).
2. w visits only finitely many times $V_0 \times \{\mathit{play}\} \times \{0, 1\}$ and goes infinitely often through both E and F.

Now we can show the main theorem of this section.

Theorem 5.1. For all states u in G, u is mercifully winning for player 0 iff $(u, \mathit{play}, 0)$ is winning for player 0 on G'. Furthermore, one can construct a merciful winning strategy for player 0 in G' using four states of memory from a memoryless winning strategy for player 0 in G.

Proof:
Assume that we are given a merciful winning strategy σ for player 0 in G. From σ we build a winning strategy for player 0 on G'. As mentioned above, we assume that from each vertex there is a path in G that visits E infinitely often. Therefore, since σ is a merciful strategy, we can choose for every σ-prefix an extension that visits E infinitely often and is played according to σ. We will use these extensions of σ-prefixes to define the strategy in G' whenever player 1 chooses to move to a *show* vertex.

Formally, for every σ-prefix $w \in V^*$ let $\gamma_w \in V^\omega$ be such that

1. $w \cdot \gamma_w \in Out(\sigma)$,
2. $\inf(w \cdot \gamma_w) \cap E \neq \emptyset$,
3. $\forall w' \in V^* : (w \sqsubseteq w' \wedge w' \sqsubseteq w \cdot \gamma_w) \implies w \cdot \gamma_{w'} = w \cdot \gamma_w$.

The third condition is a consistency condition that is needed to ensure that the strategy we construct will produce a run of the form $w \cdot \gamma_w$ if eventually player 1 decides to always move to *show* vertices.

Now, note that every path w' (finite or infinite) through G' can be turned into a path w in G by removing all vertices from $V_1 \times \{\mathit{show}\} \times \{0,1\}$ and projecting the remaining vertices to the first component.

For any $w' \in (V')^+$, letting $w \in V^+$ be w' transformed as explained above, we define the strategy σ' for player 0 in G' as follows, where *first*(γ_w) denotes the first vertex of the sequence γ_w.

- If $\mathit{last}(w') = (u, x, i)$ with $u \in V_0$, then $\sigma'(w') = (\sigma(w), \mathit{play}, \kappa(u,i))$.
- If $\mathit{last}(w') = (u, \mathit{show}, i)$ with $u \in V_1$, then $\sigma'(w') = (\mathit{first}(\gamma_w), \mathit{show}, \kappa(u,i))$.

To show that this strategy is winning we prove that every $w' \in Out(\sigma')$ is winning by performing a case split according to Lemma 5.1.

1. There are infinitely many $V_0 \times \{\mathit{play}\} \times \{0,1\}$ vertices occurring in w'. Since we follow σ in the first component and σ is winning, the outcome fulfills the Streett condition, which results in w' being winning for player 0.
2. From some point onwards, only "*show*" states occur in the second component, when the first part is at V_0. Then w' is of the form $w' = w'_1 w'_2$ such that w'_2 does not contain any vertices from $V_0 \times \{\mathit{play}\} \times \{0,1\}$. Let w_1 be the finite path through G corresponding to w'_1 (as explained above). By the definition of σ' one can see that the first component of w'_2 follows γ_{w_1}. Hence, w' corresponds to $w_1 \cdot \gamma_{w_1}$ in G and therefore visits E infinitely often (by the choice of γ_{w_1}) and also visits F infinitely often since $w_1 \cdot \gamma_{w_1} \in Out(\sigma)$. Thus, w' is winning for player 0 by Lemma 5.1.

This finishes the first part of the proof.

For the second part of the proof, given a memoryless strategy σ' for player 0 on G', we show how to construct a winning merciful strategy with finite memory $M = \{\mathit{play}, \mathit{show}\} \times \{0,1\}$, acting on G. The memory update function $\tau : \Delta \times M \to M$ is responsible for updating the memory on every transition followed in G and is defined according to the given strategy σ'. One might think that it is possible just to use memory $\{0,1\}$ and the strategy σ' as defined for vertices from $V_0 \times \{\mathit{show}\} \times \{0,1\}$ but there are examples where this approach leads to a strategy in G that is not winning. To obtain a merciful winning strategy we have to take into account the strategy σ' on both *show* vertices and *play* vertices.

Note that the main difference between G and G' is that in the *show* vertices player 0 plays on behalf of player 1. The basic idea for the strategy σ is to always use the strategy as defined for *play* vertices except if player 1 makes a move that also would have been chosen by σ' in the corresponding *show* vertex where player 0 moves instead of player 1. This idea is reflected in the definition of the memory update function τ:

$$\tau((u,v),(x,i)) = \begin{cases} (\mathit{play}, \kappa(u,i)) & \text{if} \quad u \in V_0 \\ (\mathit{play}, \kappa(u,i)) & \text{if} \quad u \in V_1 \text{ and } (v, \mathit{show}, \kappa(u,i)) \neq \sigma'((u, \mathit{show}, i)) \\ (\mathit{show}, \kappa(u,i)) & \text{if} \quad u \in V_1 \text{ and } (v, \mathit{show}, \kappa(u,i)) = \sigma'((u, \mathit{show}, i)) \end{cases}$$

The game with memory is played on $G_\tau = \langle V \times M, V_0 \times M, \Delta_\tau, \Omega_\tau \rangle$, where

- $\Delta_\tau((u,m),(v,m'))$ iff $\Delta(u,v)$ and $m' = \tau((u,v),m)$.
- Ω_τ is Streett$(\{(E \times M, F \times M)\})$.

For $u \in V_0$ we let $\sigma(u,(x,i)) = (v, \tau((u,v),(x,i)))$ with $\sigma'(u,x,i) = (v,x',i')$. Thus, from a vertex u in V, if the environment imitates the moves of σ' in "*show*" positions, the play will be equivalent to a "*show*" play in the extended game graph G'. Therefore, since σ' is winning on G', the resulting play on G_τ will be merciful (visit infinitely often both E and F).

Clearly, if σ is mercifully winning on G_τ, one obtains that σ is an automaton strategy that is winning and merciful on G. This can easily be shown by using the notion of game reduction (cf. [52]) and adapting the proof that game reduction preserves winning strategies to merciful winning strategies.

Now, we show that σ is mercifully winning on G_τ. Note that every play w in G that is played according to σ corresponds to a play w' in G' played according to σ' except that the vertices of the form (v, show, i) with $v \in V_1$ are missing in w. We formally define a function f that inserts these vertices at the appropriate places to transfer every σ-play to its corresponding σ'-play. For this purpose we first define an auxiliary function g operating on pairs of G_τ vertices as follows:

$$\begin{aligned} g((u,(x,i)),(v,(\mathit{play},i'))) &= (v,\mathit{play},i'), \\ g((u,(x,i)),(v,(\mathit{show},i'))) &= (u,\mathit{show},i)\cdot(v,\mathit{show},i'). \end{aligned}$$

Note that the second case only occurs for $u \in V_1$ and $x = \mathit{play}$ (by definition of τ).

Given a path (finite or infinite) $w = u_1u_2u_3u_4\ldots$ in G_τ, we let

$$f(w) = u_1 \cdot g(u_1,u_2)g(u_2,u_3)\ldots g(u_{i-1},u_i)g(u_i,u_{i+1})\ldots$$

Using the definition of g, one can show that

1. f is injective.
2. If $w \in Out(\sigma)$, then $f(w) \in Out(\sigma')$. (At every V'_0 position $f(w)$ follows σ' for the following reason. If the game is in a $(v,(\mathit{play},i))$ position with $v \in V_0$, then σ chooses the same move as σ'. If there is a move from (u,show,i) to (v,show,i') in $f(w)$, then this move is produced by an application of g of the form $g((u,(play,i)),(v,(\mathit{show},i')))$ and then $(v,\mathit{show},i') = \sigma'(u,\mathit{show},i)$ by definition of τ.)
3. f is invertible if we restrict the range to the set of all G' runs w' that do not contain vertices from $V_0 \times \{\mathit{play}\} \times \{0,1\}$. (From a run w' in G' that always follows "*show*" we get a run w in G with $f(w) = w'$ by removing all vertices from $V_1 \times \{\mathit{show}\} \times \{0,1\}$ from w'.)

We want to prove the following: Assuming that σ' is a winning strategy on G', then σ is winning and merciful (on G_τ).

σ is winning on G_τ: Assume that σ is losing. Then there is some path $w \in Out(\sigma)$ such that $\inf(w) \cap (E \times M) \neq \emptyset$ but $\inf(w) \cap (F \times M) = \emptyset$. By definition of f, we get that along $f(w)$ infinitely many E states occur while only finitely many F states are visited. Hence, by Lemma 5.1, $f(w)$ is not winning for player 0. Since $f(w) \in Out(\sigma')$ as shown above and σ' is winning we get a contradiction.

σ is merciful on G_τ: Let w be a σ-prefix. Since $f(w)$ is a σ'-prefix and since in G' player 1 can always choose to move to *show* vertices, there is $\gamma_{show} \in (V')^\omega$ such that $f(w) \cdot \gamma_{show} \in Out(\sigma')$ and no vertices from $V_0 \times \{play\} \times \{0, 1\}$ occur in γ_{show}. Since σ' is a winning strategy, $f(w) \cdot \gamma_{show}$ is winning, which, in turn, implies that E and F are encountered infinitely many times in the first component (Lemma 5.1). Furthermore, by the form of γ_{show}, f is invertible on it, delivering a σ-play $f^{-1}(\gamma_{show})$ such that $w \cdot f^{-1}(\gamma_{show})$ visits infinitely often $E \times M$ and $F \times M$. Thus, σ is a merciful strategy.

□

6. Conclusion

We have argued that Live Sequence Charts is a suitable and expressive scenario-based specification language, because it makes a clear distinction between mandatory and provisional scenarios. It also declares explicitly which messages are abstracted away in each scenario. This makes it possible to easily combine and refine scenarios. By the fact that

language is a variant of MSCs, it shows explicitly which instance controls which events. Then, once the frontier of the system has been drawn and actors have been assigned either to the system or the environment, one can determine which event is controlled by the system or the environment.

We then proved that a universal LSC was equivalent to a series of liveness and safety conditions, one for each event in the system. The former condition states that, whenever an event is required, it eventually occurs. The latter tells us in which situations events may not occur. This simple decomposition, on a *per event basis*, makes it possible to share responsibilities out.

Some specifications are then inconsistent, in the sense that it would be impossible to implement a program behaving as prescribed in the specification, against every well-behaving environment. Such inconsistent specifications have to be detected before one turns to design and implementation. This problem found a natural expression in the framework of turn-based games. Hence, the problem of LSCs consistency has been reduced to the problem of solving parity games, for which practically efficient algorithms exist [57]. They synthesize a strategy for the system, if the specification is consistent, or for its opponent (a sabotage plan), otherwise. Thus, meaningful feedback on specification flaws can be provided.

We have raised that some consistent specifications are problematic, because they can be implemented by non-*merciful* programs. A merciful strategy always leaves the opportunity to its opponent to fulfill her liveness conditions, provided she has been observing her safety conditions. We also gave an algorithmic method to synthesize merciful strategies. Again, feedback on ill-formed specifications can be given.

References

[1] Abadi, M., Lamport, L.: Composing Specifications, *ACM Transactions on Programming Languages and Systems*, **15**(1), January 1993, 73–132.

[2] Abadi, M., Lamport, L., Wolper, P.: Realizable and Unrealizable Specifications of Reactive Systems, *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings* (G. Ausiello, M. Dezani-Ciancaglini, S. R. D. Rocca, Eds.), 372, Springer, 1989, ISBN 3-540-51371-X.

[3] de Alfaro, L., Henzinger, T. A., Stoelinga, M.: Timed Interfaces, *Proc. of EMSOFT 2002, Second International Workshop on Embedded Software* (A. Sangiovanni-Vincentelli, J. Sifakis, Eds.), 2491, Springer, Grenoble, France, October 2002.

[4] Alpern, B., Schneider, F. B.: Defining Liveness, *Information Processing Letters*, **21**(4), October 1985, 181–185, ISSN 0020-0190.

[5] Alur, R., Etessami, K., Yannakakis, M.: Inference of Message Sequence Charts, *Proceedings of 22nd International Conference on Software Engineering*, 2000.

[6] Alur, R., Henzinger, T. A., Kupferman, O.: Alternating-time temporal logic, *Journal of the ACM (JACM)*, **49**(5), 2002, 672–713, ISSN 0004-5411.

[7] Amyot, D., Eberlein, A.: An Evaluation of Scenario Notations for Telecommunication Systems Development, *Proc. of 9th Int. Conference on Telecommunication Systems (9ICTS)*, Dallas, USA, March 2001.

[8] Antoniotti, M.: *Synthesis and verification of discrete controllers for robotics and manufacturing devices with temporal logic and the Control-D system*, Ph.D. Thesis, New York University, New York, 1995.

[9] Berry, G.: The Foundations of ESTEREL, in: *Proof, Language and Interaction: Essays in Honour of Robin Milner* (G. Plotkin, C. Stirling, M. Tofte, Eds.), MIT Press, 1998.

[10] Biermann, A. W., Krishnaswamy, R.: Constructing Programs from Example Computations, *IEEE Transactions on Software Engineering (TSE)*, **SE-2**(3), September 1976, 141–153.

[11] Bontemps, Y.: *Automated Verification of State-based Specifications Against Scenarios (A Step towards Relating Inter-Object to Intra-Object Specifications)*, Master Thesis, University of Namur, rue Grandgagnage, 21 - 5000 Namur(Belgium), June 2001.

[12] Bontemps, Y.: *Realizability of Scenario-based Specifications*, Diplôme d'études approfondies, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (University of Namur, Computer Science Dept), rue Grandgagnage, 21, B5000 - Namur (Belgium), September 2003.

[13] Büchi, J., Landweber, L. H.: Solving sequential conditions finite-state strategies, *Trans. Ameri. Math. Soc.*, **138**, 1969, 295–311.

[14] Church, A.: Logic, arithmetic and automata, *Proc. of Intern. Cong. Math*, 1963.

[15] Cobben, J., Engels, A., Mauw, S., Reniers, A., M.: *Formal Semantics of Message Sequence Charts (ITU-T Recommendation Z.120 Annex B)*, International Telecommunication Union, Eindhoven, The Netherlands, April 1998, `http://www.itu.int`.

[16] Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts, *Formal Methods in System Design*, **19**(1), 2001, 45–80.

[17] Desforges, P.: Industrialisation of the B method, *La lettre B*, **1**(1), Nov 1996.

[18] Desharnais, J., Frappier, M., Khédri, R., Mili, A.: Integration of Sequential Scenarios, *IEEE Transactions on Software Engineering*, **24**(9), September 1998, 695–708.

[19] Diethelm, I., Geiger, L., Maier, T., Zündorf, A.: Turning Collaboration Diagram Strips into Storycharts, *Proc. of "Scenarios and State-Machines: models, algorithms and tools" (SCESM) workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*, ACM, Orlando, FL, May 2002, `http://www.cs.tut.fi/~tsysta/ICSE/papers/`.

[20] Dwyer, M. B., Avrunin, G. S., Corbett, J. C.: Patterns in Property Specifications for Finite-State Verification, *Proceedings of the 21st International Conference on Software Engineering*, ACM, May 1999.

[21] Emerson, E. A., Clarke, E. M.: Using branching time temporal logic to synthesize synchronization skeletons, *Science of Computer Programming*, **2**(3), December 1982, 241–266, ISSN 0167-6423.

[22] Finkbeiner, B., Krüger, I. H.: Using Message Sequence Charts for Component-based Formal Verification, *Proc. of OOPSLA 2001 Workshop on Specification and Verification of Component-Based Systems*, Tampa Bay, FL, USA, October 2001.

[23] Grädel, E., Thomas, W., Wilke, T., Eds.: *Automata Logics, and Infinite Games: A Guide to Current Research*, vol. 2500 of *Lect. Notes in Comp. Sci.*, Springer, November 2002, ISBN 3-540-00388-6.

[24] Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications, *International Journal of Foundations of Computer Science*, **13**(1), February 2002, 5–51, (Preliminary version appeared in, *Proc. Fifth Int. Conf. on Implementation and Application of Automata (CIAA 2000)*, LNCS 2088, July 2000.).

[25] Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-Out of Behavioral Requirements, *Proc.* 4^{th} *Intl. Conference on Formal Methods in Computer-Aided Design (FMCAD'02), Portland, Oregon*, 2002, To appear. Also available as Tech. Report MCS02-08, The Weizmann Institute of Science.

[26] Harel, D., Marelly, R.: *Come, let's play! Scenario-based programming using LSCs and the Play-engine*, Springer, 2003, ISBN 3-540-00787-3.

[27] Harel, D., Politi, M.: *Modeling Reactive Systems with Statecharts: the STATEMATE Approach*, McGraw-Hill, 1998, ISBN 0-070-26205-5.

[28] Hsia, P., Samuel, J., Gao, J., Kund, D., Toyoshima, Y., Chen, C.: Formal Approach to Scenario Analysis, *IEEE journal*, March 1994, 33–41.

[29] Huber, F., Schätz, B., Einert, G.: Consistent Graphical Specification of Distributed Systems, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods, 4th International Symposium of Formal Methods Europe, Graz, Austria, September 15-19, 1997, Proceedings* (J. S. Fitzgerald, C. B. Jones, P. Lucas, Eds.), 1313, Springer, 1997, ISBN 3-540-63533-5.

[30] MSC-2000: ITU-T Recommendation Z.120 : Message Sequence Chart (MSC), 2000, `http://www.itu.int/`.

[31] Jacobson, I.: *Object Oriented Software Engineering: a Use-Case Driven Approach*, ACM Press/Addison-Wesley, 1992.

[32] Klose, J., Wittke, H.: An Automata Based Interpretation of Live Sequence Charts, *Proc. of TACAS (Tools and Algorithms for the Construction and Analysis of Systems) 2001* (T. Margaria, W. Yi, Eds.), 2031, Springer-Verlag, Genova, Italy, April 2001.

[33] Koskimies, K., Männistö, T., Systä, T., Tuomi, J.: *SCED: A Tool for Dynamic Modelling of Object Systems*, Technical Report Report A-1996-4, Department of Computer Science, University of Tampere, University of Tampere, Department of Computer Science, P.O. Box 607, FIN-33101 Tampere, Finland, July 1996, ISBN 951-44-4003-X, ISSN 0783-6910.

[34] Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts, Kluwer Academic Publishers, 1999, Franz J. Rammig (ed.).

[35] Kupferman, O., Vardi, M. Y.: Synthesizing distributed Systems, *Proc. 16th IEEE Symp. on Logic in Computer Science*, July 2001.

[36] Kupferman, O., Vardi, M. Y., Wolper, P.: Module Checking, *Information and Computation*, **164**, 2001, 322–344.

[37] Lamouchi, H. M., Thistle, J. G.: Effective control synthesis for DES under partial observations, *Proc. 39th IEEE Conference on Decision and Control (Session on Discrete Event Systems)*, IEEE, Sidney, Australia, December 2000.

[38] Li, Y., Wonham, W.: On supervisory control of real-time discrete-event systems, *Information Sciences*, **46**(3), 1988, 159–183.

[39] Madhusudhan, P., Thiagarajan, P.: A Decidable Class of Asynchronous Distributed Controllers, *Proc. of CONCUR'02*, 2421, Springer, Brno, Czech Republic, 2002.

[40] Manna, Z., Wolper, P.: Synthesis of Communicating Processes from Temporal Logic Specifications, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, **6**(1), 1984, 68–93, ISSN 0164-0925.

[41] Martin, D. A.: Borel Determinacy, *Annals of Mathematics*, **102**, 1975, 363–371.

[42] Object Management Group (UML Revision Task Force): *OMG UML Specification (2.0)*, September 2003, `http://www.omg.org/uml`.

[43] Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module, *Proceedings of the sixteenth annual ACM symposium on Principles of programming languages*, 1989, ISBN 0-89791-294-2.

[44] Puri, A., Tripakis, S., Varaiya, P.: Problems and Examples of Decentralized Observation and Control for Discrete Event Systems, in: *Synthesis and Control of Discrete Event Systems* (B. Caillaud, P. Darondeau, L. Lavagno, X. Xie, Eds.), Kluwer Academic Publisher, January 2002, ISBN 0-7923-7639-0.

[45] Ramadge, P. J. G., Wonham, W. M.: The Control of Discrete Event Systems, *Proceedings of the IEEE; Special issue on Dynamics of Discrete Event Systems*, **77, 1**, 1989, 81–98.

[46] Rosner, R.: *Modular Synthesis of Reactive Systems*, Ph.D. Thesis, The Weizmann Institute of Science, Rehovot, Israel, April 1992.

[47] Safra, S.: On the complexity of ω-automata, *29th annual Symposium on Foundations of Computer Science, October 24–26, 1988, White Plains, New York* (IEEE, Ed.), IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1988, ISBN 0-8186-0877-3 (paperback), 0-8186-4877-5 (microfiche), 0-8186-8877-7 (hard).

[48] Thistle, J. G.: Supervisory Control of Discrete Event Systems, *Mathl. Comput. Modelling*, **23**(11/12), 1996, 25–53.

[49] Thistle, J. G., Malhamé, R. P.: Control of ω-automata under state fairness assumptions, *Systems and Controls Letters*, **33**, 1998, 265–274.

[50] Thistle, J. G., Wonham, W. M.: Control of infinite behavior of finite automata, *SIAM J. Control and Optimization*, **32**(4), July 1994, 1075–1097.

[51] Thistle, J. G., Wonham, W. M.: Supervision of infinite behavior of discrete-event systems, *SIAM J. Control and Optimization*, **32**(4), July 1994, 1098–1113.

[52] Thomas, W.: Infinite games and verification, *Proceedings of the International Conference on Computer Aided Verification (CAV'02)*, 2404, Springer, 2002.

[53] Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages, *Information Processing Letters*, **90**, 2004, 21–28.

[54] Uchitel, S., Kramer, J.: A Workbench for Synthesizing Behaviour Models from Scenarios, *Proc. of the 23rd IEEE International Conference on Software Engineering (ICSE'01)*, ACM, 2001.

[55] Uchitel, S., Kramer, J., Magee, J.: Detecting Implied Scenarios in Message Sequence Chart Specifications, *Proceedings of the Joint 8th European Software Engeneering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engeneering (ESEC/FSE-01)* (V. Gruhn, Ed.), 26, 5, ACM Press, New York, September 10–14 2001, ISSN 0163-5948.

[56] Vardi, M. Y.: An automata-theoretic approach to fair realizability and synthesis, *Proceedings of the 7th International Conference On Computer Aided Verification* (P. Wolper, Ed.), 939, Springer Verlag, Liege, Belgium, 1995.

[57] Vöge, J., Jurdziński, M.: A Discrete Strategy Improvement Algorithm for Solving Parity Games (Extended Abstract), *Computer Aided Verification, 12th International Conference, CAV 2000, Proceedings* (E. A. Emerson, A. P. Sistla, Eds.), 1855, Springer, Chicago, IL, USA, July 2000.

[58] Weidenhaupt, K., Pohl, K., Jarke, M., Haumer, P.: Scenario Usage in System Development: A Report on Current Practice, *IEEE Software*, **15**(2), March 1998, 34–45.

[59] Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios, *22nd International Conference on Software Engineering (ICSE 2000)*, ACM, Limerick, Ireland, June 2000.

[60] Wong-Toi, H., Dill, D. L.: Synthesizing Processes and Schedulers from Temporal Specifications, *Computer-Aided Verification '90: Proceedings of a DIMACS Workshop* (E. M. Clarke, R. P. Kurshan, Eds.), 3, American Mathematical Society, 1991.