

From Live Sequence Charts to State Machines and Back: A Guided Tour

Yves Bontemps, Patrick Heymans, and Pierre-Yves Schobbens, *Member, IEEE Computer Society*

Abstract—The problem of relating state-based intraagent (or intraobject) behavioral descriptions with scenario-based interagent (interobject) descriptions has recently focused much interest among the software engineering community. This paper compiles the results of our investigation of this problem. As interagent formalism, we adopt a simple variant of Live Sequence Charts. For the intraagent perspective, we consider a game-theoretic foundation, looking at agents as “strategies,” which encompasses the popular “state-based” paradigm. Three classes of relationships between models are studied: scenario checking (called eLSC checking), synthesis, and verification. We set a formally defined theoretical stage that allows us to express these three problems very simply, to discuss their complexity, and to describe optimal solutions. Our study reveals the intrinsic high computational difficulty of these tasks. Consequently, many related problems and solutions are surveyed, some of which can be the basis for practical solutions. In this, we also offer a panorama of current research and directions for the future.

Index Terms—Requirements engineering, life cycle, program verification.

1 INTRODUCTION

IN [1], Harel proposed a novel way to engineer reactive distributed systems, starting from user-oriented scenarios and progressively transforming them to the level of executable code. Our investigation concerns the theoretical feasibility of this approach.

We focus on reactive distributed systems. A *distributed* computer-based system is made of several components, executing concurrently and interacting with each other to solve some problem [2]. A *reactive* system, as opposed to a *transformational* system, is one that “keeps an ongoing relationship with its environment” [3]. While the purpose of a transformational system is essentially to transform input data into output data, reactive systems interact continuously with their environment. In response to environmental stimuli, reactive systems adapt their *state*, perform some action, and get ready to react to the next stimulus. Such systems may respond differently to the first or the second occurrence of the same stimulus. Transformational systems, on the other hand, are stateless: They always deliver the same data when fed with the same input.

We focus on reactive distributed systems because 1) they are pervasive in our society, 2) their number and complexity increase rapidly, and 3) humans are more and more dependent on them. For example, the automotive industry is embedding ever more software into modern cars. Automotive software includes many highly

critical, real-time systems, e.g., braking-assistance systems, door-locking systems, cruise-control systems, and so forth. In this sector, distribution is a matter of fact, conceptually and physically: Several processors execute code in parallel, and components that are conceptually seen as separate can end up being mapped onto the same processor, thus sharing a common resource. Conversely, the same component can be deployed to different physical architectures, e.g., in another car model, thus executing in parallel. Correcting flaws in reactive distributed systems is often very expensive, sometimes prohibitively, in the case of hardware-implemented systems. Additionally, some systems might not tolerate bugs at all if human lives are at stake.

In a way, the problem of engineering reactive distributed systems is a simple task. It amounts to “write executable code fulfilling the requirements,” as shown in Fig. 1. Problems arise when there are many such requirements, when they change over time, when they are conflicting, when they are not clearly stated, and so forth. But, this is true of any realistic development endeavor. And, in the end, the engineer is left with the question: “Does my code behave as it is intended to?” In the case of distributed reactive systems, answering the question is even more difficult. Understanding and assessing the global emergent behavior of all those concurrent pieces of software and hardware is a tremendously difficult task.

Moreover, executable code is, by definition, polluted by implementation details. Those details are not necessary to understand the logics behind it. Code is also component-centric: because components are distributed, every component needs to have its own executable code that fully describes its behavior. Different parts of the system may also be written in different languages or even be implemented as hardware. This is a further impediment to understanding. Finally, the same “logical” system can be

- Y. Bontemps is with SmalS-MvM/eGov, 102 Rue du Prince Royal, B-1050 Bruxelles, Belgium. E-mail: yves.bontemps@smals-mvm.be.
- P. Heymans and P.-Y. Schobbens are with the Institut d’Informatique, University of Namur, 21 rue Grandgagnage, B-5000 Namur, Belgium. E-mail: {phe,pys}@info.fundp.ac.be.

Manuscript received 4 May 2005; revised 24 Nov. 2005; accepted 29 Nov. 2005; published online 22 Dec. 2005.

Recommended for acceptance by S. Uchitel, M. Broy, I.H. Krüger, and J. Whittle.

For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0139-0505.

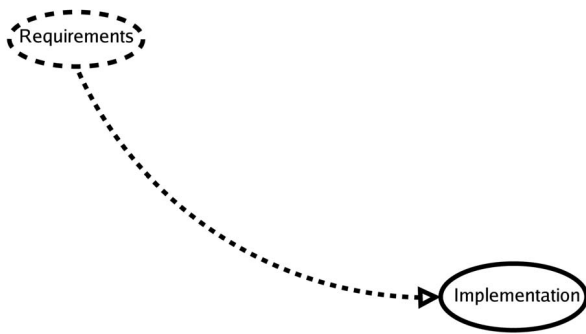


Fig. 1. One big leap (from [1]).

mapped onto different physical architectures, as already mentioned above.

In order to understand reactive distributed systems, instead of using executable code, engineers often resort to an abstraction of it. OMG's Model-Driven Architecture (MDA) initiative calls such an abstraction a "platform-independent model" (PIM) [4]. This abstraction is a *design model* (see Fig. 2), tailored for human understanding. It ignores implementation details and only presents the control logic of every component, preferably in an architecture-independent way. When useless details have been removed, the system appears in a much simpler form that one can read and, hopefully, better understand.

Being an abstraction of the actual code, it is no wonder that a design model inherits many of its properties. Implementation and architecture-related parts have been forgotten, but a design model is still component-centric and complete. For every component of the system, it has to describe its entire behavior: how it should act in every possible situation. We will thus say that this model is *intraagent*, as it considers every component separately. We use the term "agent" here in its broad, Latin meaning, i.e., "an acting entity." In this document, we consider it a generalization of, and therefore interchangeable with "component," "subsystem," "process," or "object."

Relying on this intraagent abstract model, can we be sure that the actual system fulfills the user requirements? Not really. There is still a huge gap between requirements and design models. Users do not spontaneously think in terms of intraagent specifications. They are usually unable to express, even in natural language,

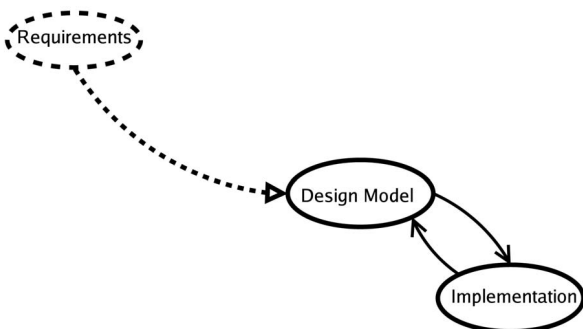


Fig. 2. Introducing design models (from [1]).

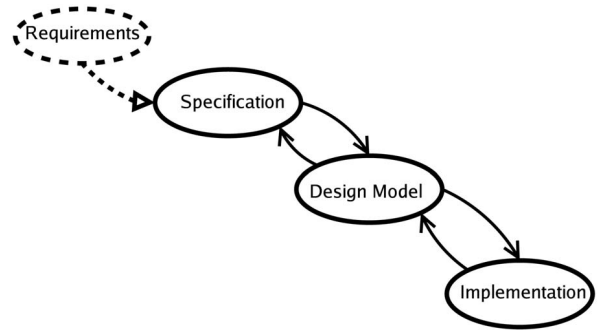


Fig. 3. Interagent specification narrows the gap (from [1]).

what the complete behavior of the future system should be. Studies have shown that users feel more comfortable using *scenarios*, i.e., partial and concrete example stories of the future system [5].

Those executions are not focused on a single agent. When dealing with distributed systems, a scenario will involve several agents and cross their borders. It is thus said to be an *interagent* representation. For instance, the following user statement is, in our terminology, a scenario: "I engage cruise control by pressing the SET button. A green light flashes on the dashboard for five seconds and remains lit. Then, when I push on the brake pedal, cruise control is disengaged, the car brakes, and the green light turns off." This scenario involves several agents (SET button, cruise control, green light, brake pedal, and so forth) and is partial as it does not tell about all the other situations that may happen when cruise control is engaged. It also overlaps with other scenarios, e.g., scenarios talking about ABS or automatic door-locking system, although those scenarios also share brake-pedal interactions.

To summarize, we observe that the gap to bridge between requirements and design models is caused by *two* paradigm shifts. First, engineers have to move from a partial and overlapping description of the system to a complete behavioral specification. Second, a transition from the interagent to the intraagent perspective must be made. When these two changes take place at once, mistakes are very likely to be introduced: Some examples will be wrongly generalized and/or not correctly distributed among agents.

In order to progressively narrow this gap, we follow the well-known "divide and conquer" strategy (see Fig. 3). First, we build a complete behavioral specification, using an interagent specification language. Second, this interagent specification is transformed into an intraagent design model. Of course, this does not remove all risks of building the wrong system (or building the system wrong) for no technique could ever fully remove this risk. But, it is meant to make the development more reliable by helping engineers master a well-known source of errors.

If one provides an *automated* translation from interagent specifications to intraagent specifications, risks will be even more decreased. No mistake can be introduced by the transition from specification to design model, provided that the translator software has been proven correct. This proof

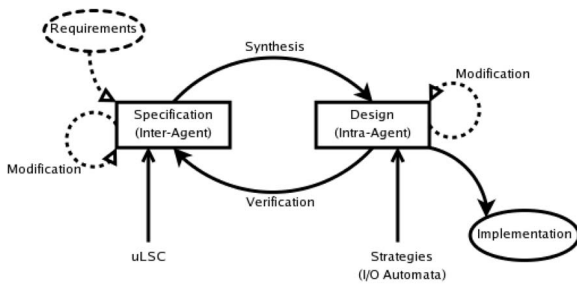


Fig. 4. Full development (from [1]).

can be done once and for all by experts, thus removing the burden of verification from engineers.

There are several links that need to be automated. First, the interagent specification should be tested in order to check that it fulfills the users' intent. Second, intraagent design models should be derived from the interagent specifications. This implies that interagent specifications should be tested for implementability: If the specification is inconsistent, i.e., imposes conflicting requirements on the system, those should be detected and no implementation must be synthesized. Third, both models are likely to evolve on their own. New requirements will be added to the specification and, in parallel, the design model will become more detailed. After some time, it is interesting to check whether the design model still complies with the specification, i.e., that the requirements are still met. This approach has been suggested by Harel [1] and is illustrated in Fig. 4.¹

In order to support this approach, the following issues need to be addressed:

- A scenario-based interagent formal specification language needs to be defined. We propose using Live Sequence Charts (LSCs), which were introduced in [6] and later substantially extended [7]. We focus on a simple part of the original version, named Constant LSCs in [7].
- The various links between specification and design models need to be formalized. In particular, what it means for an intraagent to "correctly implement" an interagent specification needs to be clearly defined.
- The feasibility of these links must be assessed. When feasible, algorithms supporting the links must be developed and implemented. A link is "feasible" if it can be implemented by some algorithm, taking into account real-world efficiency constraints.

This is basically what this paper has to offer. It compiles the results of an extensive investigation of these issues, which took place over the last five years. Our object of study is a fully formalized scenario-based synthesis and verification framework, proposed to address the challenges discussed above. The framework is an attempt to incarnate Harel's initial vision of an ideal scenario-based software development approach [1]. This vision is the one that we have depicted throughout the introduction. In his paper, Harel gives intuitive arguments toward the achievability of

his "dream," as we have so far. The results that will be presented in the remainder of this paper are mathematical evidence that some ways to make the dream real are simply impracticable. This, as we will explain, is due to the high computational complexity or the undecidability of the problems to be solved. Indeed, a series of synthesis and verification problems have been studied, under various hypotheses, but the verdict, as we will show, has always turned out to be tough. Far from being a negative result, or questioning Harel's vision, our contributions more accurately indicate what kinds of solutions do not work and what kinds are more likely to contribute to making the dream a reality. To emphasize this, the paper contains a comprehensive discussion of related work, insisting on approaches that are judged promising. In that, this paper is also a survey of formal scenario-based techniques.

The rest of the document is structured as follows: In Section 3, we briefly establish some notational conventions used throughout the paper. Section 3 introduces LSCs, a scenario-based language for specifying reactive distributed systems that adds more expressiveness with regard to the widespread Message Sequence Charts notation. In Section 4, the various models involved in the engineering process of Fig. 4 are formally defined: the interagent specification (Section 4.2), the intraagent specification (Section 4.3), and the structural specification (Section 4.1) that makes the link between them. We also determine what it means for an intraagent specification to be a correct implementation of an interagent specification (Section 4.4). This sets the stage for our investigation of Harel's vision. Relying on these notions, in Section 5, we analyze the various problems involved in the process of Fig. 4. For each problem, we assess its complexity, provide an asymptotically optimal algorithm, and survey the literature in relation to it. We conclude in Section 6 by summing up our contributions and proposing possible research directions.

2 NOTATIONS

In this section, we briefly introduce our notations. A word over an alphabet Σ is a sequence of symbols $e_1 \dots e_n$, with $e_i \in \Sigma$. The empty word is denoted ϵ . The set of all finite words over Σ is denoted by Σ^* . A language over Σ is a set $L \subseteq \Sigma^*$. The concatenation of n words v_1 to v_n will be denoted $v_1 v_2 v_3 \dots v_n$. Every concatenation of words $v_1 \dots v_n$ that forms a given word v ($v_1 \dots v_n = v$) is called a decomposition of v . We say that u is a prefix of w ($u \sqsubseteq w$) if there is some word v such that $uv = w$. If v is nonempty, u is a strict prefix of w and we write it $u \sqsubset w$. Projecting a word $v \in \Sigma^*$ over an alphabet $A \subseteq \Sigma$, written $v|_A$, discards from v all symbols not in A . Formally, it is the word obtained by applying the following recursive rule: $\epsilon|_A = \epsilon$ and, for $e \in A$, $ue|_A = u|_A e$ and, for $e \notin A$, $ue|_A = u|_A$. An infinite word over Σ is an infinite sequence of symbols from Σ . We denote by Σ^ω the set of all infinite words over Σ .

3 SCENARIO-BASED SPECIFICATION

In this section, we look for an adequate interagent, formal, scenario-based specification language. We first consider message sequence charts (MSCs) [8] on the grounds of their

1. The terms "uLSC" and "strategies (I/O automata)" appearing in the figure will be explained later in Sections 3 and 4.

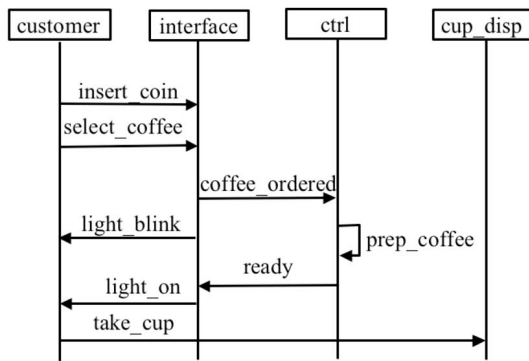


Fig. 5. A Message Sequence Chart for a coffee dispensing machine.

popularity. Yet, we underline major shortcomings that make them unsuitable for our purpose. LSCs are then introduced as an adequate solution.

3.1 Message Sequence Charts

MSCs are a popular formalism for describing scenarios. They have been standardized by the ITU [8]. A variant, called Interaction Diagrams, is also found in UML 2.0 [9]. An MSC displays vertical “lifelines,” each representing the contribution of an involved agent whose name is indicated at the top. Time goes from top to bottom. For instance, the MSC of Fig. 5, for a coffee dispensing machine, describes a scenario involving four agents: a customer, a user interface, a controller, and a cup dispenser. In MSCs, agents interact by sending messages (represented as arrows) to each other.² The MSC of Fig. 5 represents the natural-language scenario “when the customer inserts a coin and selects coffee, the interface tells the controller that coffee has been ordered and makes a light blink. The controller prepares coffee and notifies the interface when it is ready. The light is turned on and the customer takes the cup from the dispenser.”

An MSC describes a partial order between events, according to the following rule: An event may occur if all events higher up on the same lifeline already happened. This introduces concurrency between events. For instance, in Fig. 5, events “light_blink” and “prep_coffee” are not ordered and may occur in any order.

MSCs feature composition operators that make it possible to describe complex scenarios from basic ones. Typical control flow operators, such as concatenation, choice, unbounded loop (Kleene’s star), and interleaving, are provided. There are two ways of using these operators: inline expressions and High-Level MSCs (hMSCs). In the former, boxes containing MSCs are included in MSCs themselves. In the latter, a graph whose nodes contain MSCs indicates all possible sequences of MSCs.

Despite its graphical appeal, confirmed by a wide popularity, the language has, in our opinion, five major shortcomings that make it unsuitable for our purpose:

- The *semantics* of MSCs is weak and aims at describing execution samples only. This is very different from *specifying* (meaning “describing completely and precisely”) the expected behavior of the future system.
- *Composition* of MSCs causes enormous overhead for the engineer. This is because the semantics of MSCs is silent about the meaning of a set of scenarios, i.e., how the requirements they impose on the future system shall be composed.
- MSCs lack syntactic constructs to express the *scope* of a scenario, i.e., whether events not appearing in this scenario can occur at will or are forbidden by their mere absence.
- MSCs do not distinguish between events that *trigger* the scenario and events that occur *in response* to this activation, even though this is a usual implicit distinction in scenarios.
- MSCs offer no syntactic means to distinguish between *universal rules* and *examples*.

For a more detailed discussion of these shortcomings, including examples, the interested reader is referred to [6] or [10].

In an attempt to tackle these problems, the OMG has substantially extended the language of MSCs in the UML 2.0 [9]. The semantic domain of the so-called interaction diagrams is a couple of sets of executions. In a couple (G, B) , G is the set of “good executions,” i.e., positive examples, and B is the set of “bad executions,” i.e., counterexamples. G and B must be disjoint, but not necessarily cover the universe of executions. Therefore, there are some executions about which it is not known whether they are good or bad. Those are called *inconclusive*. Interaction diagrams introduce new composition operators, named “restrict,” “ignore,” “assert,” and “neg.” The first two operators specify the scope of a scenario by adding/removing events from its scope. The “assert” operator means that only the traces described in the diagram are good; all other traces are bad. The “neg” operator means that the described traces are counterexamples. However, as demonstrated in [11], [12], the UML standard is ambiguous about the meaning of these operators. Their semantics is only defined with regard to good scenarios. The interpretation of negation is ambiguous. Furthermore, we showed that, under some reasonable assumptions, it is undecidable whether an Interaction Diagram is consistent, i.e., whether good and bad executions are disjoint [12].

Given all the above reasons, we conclude that MSCs (including UML interaction diagrams) are not suitable for the specification of reactive distributed systems. However, MSCs are nowadays the most popular scenario language. They are at the root of a whole family tree of languages, namely, sequence chart languages. To provide a realistic alternative to MSCs, we need to maintain the appeal of the language, hence, the goal is to be as conservative as possible concerning syntax. Semantics, on the other hand, needs to be more thoroughly reshaped. The adopted solution is now presented.

2. The message construct is used not only to represent instantaneous events exchanged between agents but also shared states or shared noninstantaneous activities. In these latter cases, messages indicate the start (respectively, the entering) or the end (respectively, the leaving) of an activity (respectively, the state). For example, “insert_coin” is the end of an activity performed by the user; “light_blink” is the start of the state of the interface in which the light is blinking.

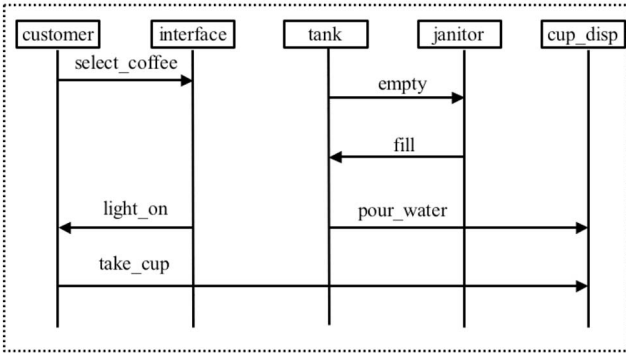


Fig. 6. Sample existential LSC of the coffee dispensing machine.

3.2 Live Sequence Charts

Damm and Harel introduced LSCs in [6]. The language was later extended in [7]. In this paper, we focus on a simple version of LSCs, named *Constant LSCs* [7].

3.2.1 Basic Charts

LSCs are built from *basic charts*. MSCs, such as the one presented in Fig. 5, are basic charts. Like MSCs, basic charts can also be composed using *inline expressions*. As control flow operators, we allow strong concatenation (called strict sequencing in [9]), nondeterministic choice, and interleaving, but we forbid unbounded loops. The semantics of a basic chart B is a set of executions $\mathcal{L}(B)$. Executions are generated by successively marking locations (arrows) in B . An arrow a involving lifelines l_1 and l_2 and labeled by e may be marked when all arrows touching l_1 and all arrows touching l_2 that are drawn higher up than a have already been marked. The fact of marking a produces its labeling event e . An execution is a sequence of events produced by successively marking all locations in B from the empty marking. The set of all executions of a basic chart B is called its “language” and is denoted $\mathcal{L}(B) \subseteq \Sigma^*$.

3.2.2 Existential and Universal Charts

Basic charts do not solve the problems of MSCs. In order to address these issues, we follow Damm and Harel [6] and add syntactic constructs.

First, LSCs are given a *scope*, which consists of all events occurring in the chart, plus additional events that are listed in a special “restricts” clause. Events that are not in the scope of the LSC may occur at will.

Second, LSCs are decorated with a modality. An LSC can be *existential* or *universal*. An existential LSC (eLSC) is a basic chart, surrounded by a dashed-line rectangle, that describes a possible execution of the future system (see Fig. 6). Abstractly, we represent an eLSC as $\Diamond(B, \Sigma_R)$, where B is a basic chart and Σ_R is the scope of the eLSC. A universal LSC (uLSC), as presented in Fig. 7, is made of two superimposed basic charts. The upper one is surrounded by a dashed-line hexagon and named *prechart*. The lower one is called *main chart* and is surrounded by a solid-line rectangle. This makes it possible to distinguish between activation events, i.e., the prechart, and response events, i.e., the main chart. The semantics of a uLSC is “whenever the prechart occurs in an execution, the main chart shall

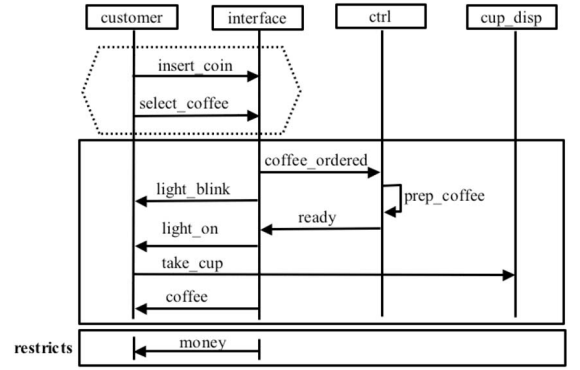


Fig. 7. Sample universal LSC of the coffee dispensing machine.

follow.” We will write uLSCs as $\Box(P, M, \Sigma_R)$ with P and M being basic charts. P is the prechart, M is the main chart, and Σ_R is the scope of the LSC, i.e., all events appearing in P or M , plus additional events specified by the “restricts” clause.

We are interested in reactive systems, which are not supposed to stop. Abstractly, we consider an execution of a reactive system as a sequence of observed events. Like linear temporal logics (LTL), LSCs are thus interpreted against *infinite words*. We write $\gamma \models L$ to denote that γ is a legal execution with regard to L and define this relation as follows:

Definition 1 ($\models \subseteq \Sigma^\omega \times \text{LSC}$). An infinite word $\gamma \in \Sigma^\omega$ satisfies an LSC L , written $\gamma \models L$, iff one of the following two constraints holds:

- $L = \Box(P, M, \Sigma_R)$ and, for every decomposition $uv\gamma'$ of γ , if $p|_{\Sigma_R} \in \mathcal{L}(P)$, then there is a decomposition $m\gamma''$ of γ' such that $m|_{\Sigma_R} \in \mathcal{L}(M)$.
- $L = \Diamond(B, \Sigma_R)$ and there is a decomposition $uv\gamma'$ of γ such that $v|_{\Sigma_R} \in \mathcal{L}(B)$.

An LSC L defines the language

$$\mathcal{L}(L) = \{\gamma \in \Sigma^\omega \mid \gamma \models L\}.$$

A set of executions $\Gamma \subseteq \Sigma^\omega$ satisfies an LSC L , which we denote $\Gamma \models L$, iff

- $L = \Box(P, M, \Sigma_R)$ and $\Gamma \subseteq \mathcal{L}(L)$.
- or, $L = \Diamond(B, \Sigma_R)$ and $\Gamma \cap \mathcal{L}(L) \neq \emptyset$.

uLSCs are a real specification formalism in the sense that they draw a clear line between legal and illegal behaviors. An execution is illegal if, at some point in it, the prechart is matched, but is not followed by the main chart. The set of all legal executions with regard to a uLSC L is $\mathcal{L}(L)$, the language defined by L .

In the scenario-based software engineering approach that we adopted, we are interested in gathering scenarios from several stakeholders and “throwing” them together in a “big bag of requirements.” This set is deemed a specification and the semantics of the language will take care of how these bits of requirements shall be composed. Hence, the following definition:

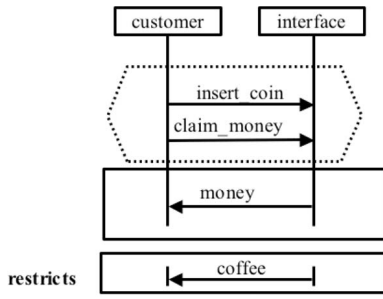


Fig. 8. uLSC “return money” of the coffee dispensing machine.

Definition 2 (uLSC-Spec). A uLSC specification is a finite set of uLSCs.

Conjunction is our composition operator, hence providing uLSC-Spec with a semantics. A uLSC-Spec is a set of uLSCs. Intuitively, every uLSC corresponds to one requirement about the future system. Every uLSC constrains the behavior of the future system by obliging the main chart to be observed after each occurrence of the prechart. A specification obliges all executions of the future system to match the constraints set by each uLSC of which it consists:

$$\mathcal{L}(\{L_1, \dots, L_n\}) = \bigcap_{i=1}^n \mathcal{L}(L_i).$$

Composition is *semantic*. It is in general impossible to conjunct a uLSC-Spec into a single uLSC [10].

4 MODELS

In this section, we present the abstract models that are used in Fig. 4. We use three kinds of models. The first model presents the structure of the system to be built. It shows the different agents and displays the communication channels. It constrains the interactions by stating precisely who can communicate with whom and which messages may be used. This model is *static*. It states what every agent can do, but does not provide information on when agents must or should not perform an action.

The next two models are needed to incorporate dynamic information on the behavior of agents. On the one hand, there is an “interagent” language. It uses uLSCs to describe interaction scenarios. Every scenario is made of an activation condition and a precise description of how the agents must interact after this activation. On the other hand, there is an “intraagent” language. For every single agent, it shows what actions the agent must perform in every reasonably conceivable situation.

The former is closer to the user’s requirements. Hence, it acts as a specification. The latter is closer to code and, therefore, we consider it a “design artifact.” Of course, the design model should correctly implement the specification in order for the system under development to meet user requirements. A precise definition of the concept of “correct implementation” is needed and will also be provided in this section.

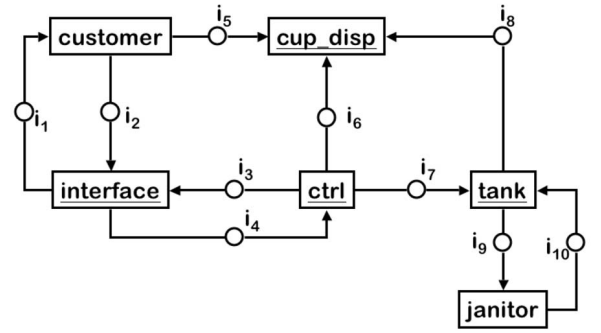


Fig. 9. Structure diagram for the coffee dispensing machine.

4.1 Structure

The structural view of the system provides static information about the possible interactions between agents. It lists the various agents and shows what events they trigger (send) or sense (receive).

Concretely, a system structure is described in a diagram such as the one shown in Fig. 9. The notation we use here could be replaced by another equivalent notation, such as context diagrams [13] or UML object or component diagrams [9]. We use our own notation for the sake of simplicity. Agents are drawn as boxes in which their name is written.

A distinction is made between agents of the future system and agents that (will) exist independently of the future system. The former are “system agents,” and the latter are “environment agents.” System agents have their names underlined.

An interface is drawn as a circle with an identifier. In a separate table, this identifier is expanded as a list of event names (see Table 1). In Fig. 9, an arrow from agent a_1 to agent a_2 , going through interface I , means that a_1 may send all events in I to a_2 .

Formally, we will use the following definition system structures.

Definition 3 (System Structure). A system structure (in abstract syntax) is a tuple

TABLE 1
Interface Definitions for the Coffee Dispensing Machine

Identifier	Events
i_1	light_blink, light_on, light_off, money
i_2	insert_coin, select_coffee, claim_money
i_3	ready
i_4	coffee_ordered
i_5	take_cup
i_6	prepare_cup, add_coffee
i_7	get_water
i_8	pour_water
i_9	empty
i_{10}	fill

$$\langle Ag, (\Sigma_a^s)_{a \in Ag}, (\Sigma_a^r)_{a \in Ag}, Sys \rangle,$$

where

- Ag is a finite set of agent names.
- Σ_a^s gives, for every agent a , the events that it sends. These sets are disjoint, as two agents may not control the same event.³
- Σ_a^r is the set of events received by agent a . Again, these sets must be disjoint.
- $Sys \subseteq Ag$ is the set of system agents. All agents not in Sys are called environment agents: $Env \triangleq Ag \setminus Sys$.

We let the set of all events be Σ and require that $\bigcup_{a \in Ag} \Sigma_a^r = \bigcup_{a \in Ag} \Sigma_a^s = \Sigma$. We let $\Sigma_a = \Sigma_a^r \cup \Sigma_a^s$.

Mapping this formal definition back to our concrete syntax, we have Ag as the set of all names appearing in boxes and Σ represented by interfaces.

In the rest of this section, we will refer to this structure and assume that it is fixed.

4.2 Interagent Behavior

Interagent behavior is specified with uLSCs. Every behavioral requirement is translated and generalized into some uLSC. All uLSCs are then gathered in a uLSC-Spec. Since uLSCs come with a built-in notion of scenario integration, through conjunction, the uLSC-Spec describes all the desirable behaviors of the future system.

Definition 4 (Interagent Specification). An interagent specification is a couple

$$\langle S, \mathcal{U} \rangle,$$

with S being a system structure and $\mathcal{U}$ a uLSC-Spec, over alphabet Σ . It is required that only agents from Ag take part in $\mathcal{U}$ and that they respect their interfaces, i.e., a (uLSC) arrow from agent a_1 to a_2 may only be labeled by events from $\Sigma_{a_1}^r \cap \Sigma_{a_2}^s$.

4.3 Intraagent Behavior

The system structure specifies which agents belong to the system (Sys) and which agents are part of their environment (Env). Engineers only have to implement Sys agents. These will later be deployed among, and interact with, Env agents. The system to be built is *open*: engineers have no control over what Env agent may do.

We consider agents abstractly as “reactive (sub)systems”: they continually react to environment inputs (stimuli) by producing outputs (responses). This reaction may be history-dependent: An agent may provide different responses to the same stimulus, because different events have been observed in the past.

Definition 5 (Strategy). Our abstract view of an agent a is a (nondeterministic) strategy $f_a : \Sigma^* \rightarrow 2^{\Sigma_a^s}$.

A strategy f_a represents the behavior of agent a . For every possible history $w \in \Sigma^*$, $f_a(w)$ is the set of all possible

responses that a can deliver. Conversely, after w , a “decides” to *disable* all events in $\Sigma_a^s \setminus f_a(w)$.

This abstraction highlights that each agent has powers: It can make some events happen, because they are under its control. Other events, on the contrary, are beyond its control; they cannot be delivered or constrained by this agent. Thus, a single agent *cannot* force its environment or prevent it from performing an event. A strategy for a hence *generates* a set of runs: all the executions in which events controlled by a only appear when they are enabled by f_a .

Definition 6 (Outcome). The outcome of a strategy f_a is the set of all runs in which Σ_a^s events appear only according to the strategy $Out(f_a) = \{u_0 e_0 u_1 e_1 \dots \mid \forall i \geq 0 : u_i \in (\Sigma \setminus \Sigma_a^s)^* \text{ and } e_i \in f_a(u_0 e_0 \dots u_i)\}$.

Agents can be organized in societies, e.g. Sys and Env . We have thus to explain how their behavior can be composed. A society is simply a set of agents $A \subseteq Ag$. Its sent (respectively, received) events are the union of all events sent (respectively, received) by its composing agents: $\Sigma_A^s = \bigcup_{a \in A} \Sigma_a^s$ and $\Sigma_A^r = \bigcup_{a \in A} \Sigma_a^r$. The strategy of A is also the union of the strategies of its agents: $f_A(w) = \bigcup_{a \in A} f_a(w)$.

Definition 7 (Intraagent Specification). An intraagent specification is a couple

$$\langle S, f_{Sys} \rangle,$$

where S is a system structure and f_{Sys} is a strategy for society Sys .

Although this model may be appealing from a mathematical point of view, sometimes we will have to focus only on strategies that are representable within computers. We use the notion of input/output (I/O) automaton for this purpose [14]. This is our intraagent specification formalism. It encompasses the popular state machine formalism.

Definition 8 (I/O Automaton). An I/O automaton for agent $a \in Ag$ is a deterministic finite automaton, the alphabet of which is Σ_a . A distinction is made between input events (Σ_a^r) and output events (Σ_a^s). Syntactically, an I/O automaton for agent a must be input-enabled: In every state q , agent a should have one transition labeled by every input event. In other words, a may never block incoming events.

This definition fulfills our goal to model agent powers, as explained above.

An I/O automaton represents a finite-memory strategy for agent a . Formally, a strategy f_a for agent a is of finite memory if there is an equivalence relation $\simeq$ on Σ^* such that 1) $\forall w \simeq w' : \forall e \in \Sigma : we \simeq w'e$, 2) $\simeq$ is of finite index, and 3) $\forall w \simeq w' : f_a(w) = f_a(w')$. The size of the memory is the index of the smallest such equivalence relation. Every finite-memory strategy can be translated to an I/O automaton. Conversely, every I/O automaton can be turned into a finite-memory strategy.

4.4 Relating Interagent and Intraagent Specifications

In order to define the automated links between our artifacts, we must tell when an intraagent specification is *correct* with

3. Note that disjointness can always be satisfied by considering the sender and receiver as part of the event identity: It is just a convenience.

regard to an interagent specification. As a first attempt, one could resort to *trace inclusion*, i.e., require that all the executions generated by the intraagent specification be legal behaviors with regard to the interagent specification. Formally, given $\langle S, \mathcal{U} \rangle$ and $\langle S, f_{Sys} \rangle$, $Out(f_{Sys}) \subseteq \mathcal{L}(\mathcal{U})$.

However, this definition causes a mismatch between powers and responsibilities. Since system agents only control some events, they shall not be responsible for errors due to the occurrence of events they do not control. We illustrate this situation with an example. Consider the uLSC of Fig. 7 and the partial execution “insert_coin, select_coffee, coffee_ordered, light_blink.” This execution matches the prechart and partially matches the main chart. What happens if, at this point, “money” occurs? The specification is violated, since “money” is restricted but is not one of the next allowed events. This error is attributable to the system, as it controls “money” and thus *decided* to give money back when it was not allowed. Now, in the same configuration, what happens if the customer takes his cup (“take_cup”)? Again, the specification is violated, as “take_cup” is not one of the next allowed events. An intraagent specification allowing such an execution would therefore be classified as incorrect, according to trace inclusion. However, this sounds rather unfair as, in the latter case, violation was not due to the system but to an inadequate action taken by the customer. Therefore, the system should not be held responsible for this situation. A similar example is the following: Still in the same configuration, “prep_coffee, ready, light_on” occur. The new configuration of the uLSC now requires “take_cup” to occur in order to complete the main chart. What if the customer walks away and never takes his cup? The specification is violated, as the main chart will never be matched, but the system is not responsible for this situation either.

Actually, the two kinds of violations of which we just gave examples suffice to capture the semantics of uLSCs: If, along a run γ , every agent ensures that its controlled events do not occur when they are forbidden and that they eventually occur when they are required to, then γ will satisfy the uLSC [15]. This is good news, as it makes it possible to share responsibilities along with powers. For some set of events $V \subseteq \Sigma$, we will say that an execution is *V-safe* if no event in V occurs when it is forbidden. It is *V-live* if every event in V eventually occurs when it is required. We formalize these notions.

Definition 9 (Forbids-Safe). *Given some uLSC $L = \square(P, M, \Sigma_R)$, we say that a word $w \in \Sigma^*$ forbids some event $e \in \Sigma_R$ iff there is a decomposition upv of $w|_{\Sigma_R}$ such that*

1. $p \in \mathcal{L}(P)$ (the uLSC has been activated),
2. $\forall m \sqsubseteq v : m \notin \mathcal{L}(M)$ (it is not yet terminated),
3. $\nexists v' \in \Sigma_R^* : vev' \in \mathcal{L}(M)$ (doing e would prevent termination).

An infinite run γ is e-safe iff, for every $uv \sqsubset \gamma$, if v forbids e , then $uve \not\sqsubset \gamma$.

Definition 10 (Requires-Live). *Given some uLSC $L = \square(P, M, \Sigma_R)$, a word $w \in \Sigma^*$ requires an event $e \in \Sigma_R$ iff there is a decomposition upv of $w|_{\Sigma_R}$ such that*

1. $p \in \mathcal{L}(P)$ (the uLSC has been activated),
2. $\forall m \sqsubseteq v : m \notin \mathcal{L}(M)$ (it is not yet terminated),
3. $\exists v' \in \Sigma_R^* : vev' \in \mathcal{L}(M)$ (termination cannot take place without e).

An infinite run γ is e-live iff, for every $uv \sqsubset \gamma$, if v requires e , then there is $w \in \Sigma^*$ and $e' \in \Sigma_R$ such that $uvwe' \sqsubset \gamma$.

Generalizations of these two definitions from events (*e-safe/e-live*) to sets of events (*V-safe/V-live*) is straightforward.

These constraints, which are given event by event, are equivalent to the semantics of uLSCs.

Theorem 1 (uLSC = safe + live). *For every $\gamma \in \Sigma^\omega$ and every uLSC $S = \square(P, M, \Sigma_R)$,*

$$\gamma \models S \iff \forall e \in \Sigma_R : \gamma \text{ is } e\text{-safe and } e\text{-live}.$$

In the spirit of Gunter et al. [16], we can state that an intraagent specification is correct if, for every *well-behaving* environment, the system also behaves correctly.

Definition 11 (Correct Implementation). *An intraagent specification (S, f_{Sys}) associated to a society of agents Sys is a correct implementation of an inter-agent specification $(S, \mathcal{U})$ iff $\forall \gamma \in Out(f_{Sys})$, these two constraints hold:*

- $\gamma \text{ is } \Sigma_{Env}^s\text{-live} \implies \gamma \text{ is } \Sigma_{Sys}^s\text{-live}.$
- $\gamma \text{ is } \Sigma_{Env}^s\text{-safe} \implies \gamma \text{ is } \Sigma_{Sys}^s\text{-safe}.$

4.5 Discussion

In the last two sections, we have defined settings in which we assume an “uncontrollable environment.” This means that, at any point in time, an environment is likely to generate any event as an input to the system. Our agents are further not allowed to know the relative speed of the other agents: They have to accept any sequence before they can react.

The first part of our assumption is due to the “input-enabledness” of I/O automata and our definition of “correct implementation.” Relaxing input-enabledness would allow system agents to disable incoming events, that is, to inhibit their environment, e.g., through handshake communication. In this case, we would actually be dealing with closed systems.⁴ Input-enabledness has been recognized as a primary characteristic of reactive systems for years (see [17], [18]). We mitigated it with the constraints imposed by the definition of “correct implementation.” However, the reader should pay attention to the fact that these constraints do *not* mean that environment agents are prevented from acting as they want. They are free to do so, but, in case they violate these basic constraints, the system may stop working according to the specification.

Concerning the second part of our assumption, we could instead have chosen the “perfect synchrony” hypothesis, where the environment would generate a single event, and then the system can go infinitely fast to perform any sequence of events. The perfect synchrony assumption is a useful simplifying assumption in reactive-systems engineering as it abstracts away all the problems related to

4. In closed systems, synthesis boils down to satisfiability. This will be discussed in Section 5.

environment speed [19]. However, composing intraagent specifications under this assumption is much harder than in our settings since all system components assume that they are going infinitely faster than their neighbor. We chose our model for the sake of simplicity. Furthermore, imposing the perfect synchrony hypothesis would not change the complexity of the problems presented in Section 5. The proof is based on the idea that our settings can be encoded in synchronous settings by letting environment agents choose when the system may respond to their inputs. This constraint can be written as a synchronous uLSC.

5 ANALYSIS PROBLEMS

Having precisely set the stage for a round-trip scenario-based software engineering approach (recall Fig. 4), we can now move to the last step: precisely describing what the supporting tools should do and how they could do it.

Such tools should offer three functionalities: 1) eLSC checking, 2) synthesis, and 3) verification. In this section, we will study each of these functionalities in turn. For each of them, we first precisely describe the *problem*, then provide an algorithm (*solution*) implementing it; we will also study the computational *complexity* of each problem and *review existing work* on LSCs in relation to it. The latter is intended to provide a structured panorama of current research on LSCs.

5.1 eLSC Checking

5.1.1 Problem

Imagine the following situation: Users provide examples of expected system behavior. These informal scenarios are translated to eLSCs. From these eLSCs, the engineer derives an interagent specification. As already explained, every new uLSC added to this specification constrains the desirable behavior of the future system a little bit more. At some point, it might make sense for the engineer to check that the specification is not too restrictive, i.e., that the original eLSCs from which the specification was derived are still allowed by this specification. If they are not, then the engineer needs to relax the behavior.

The eLSC checking problem is to determine whether an interagent specification satisfies an eLSC. Formally, it can be defined as:

Definition 12 (eLSC Checking). *The eLSC Checking decision problem is specified as*

- *Input:*
 - An interagent specification $\langle S, U \rangle$
 - An eLSC $\Diamond(M, \Sigma_R)$
- *Output:*
 - yes, if $\mathcal{L}(U) \models \Diamond(M, \Sigma_R)$
 - no, otherwise

5.1.2 Solution

There is an automata-theoretic algorithm solving this problem. The uLSC-Spec can be translated to a deterministic Büchi automaton $\mathcal{A}_U$ such that $\mathcal{L}(\mathcal{A}_U) = \mathcal{L}(U)$ [10]. The

eLSC can be turned into an LTL formula $\varphi_{\neg\Diamond(M, \Sigma_R)}$ which asserts that M is *never matched* [20]. Then, a model checker is used to verify that $\varphi_{\neg M}$ does not hold in $\mathcal{A}_U$. If a counterexample is found, it means that there is a run satisfying the eLSC and we are given such a witness. This algorithm can be implemented to perform an on-the-fly search, hence avoiding building $\mathcal{A}_U$ completely before searching for a satisfying path.

5.1.3 Complexity

This algorithm is in PSPACE. The problem is also PSPACE-hard [21].

Theorem 2. *eLSC checking is PSPACE-complete.*

In principle, eLSC checking is thus intractable.

5.1.4 Related Work

This problem is actually closely related to the “smart play-out” approach, by Kugler et al. [22] Play-out is an interactive technique for executing LSC specifications. It proceeds in turns: The user performs some event through a graphical front-end; this event may complete the prechart of some uLSCs, hence activating them; the play-out engine computes a superstep, i.e., a sequence of system events leading to a state in which system agents are not required to perform any more events. Due to nondeterminism in the uLSC specification, there may be several possible supersteps, but some of them can lead to deadlock or divergence (infinite supersteps). The basic play-out engine arbitrarily explores a single superstep, hence possibly running into problems. The *smart* play-out approach uses model checking to compute a valid superstep, if it exists. Computing valid supersteps is reducible to eLSC checking of uLSC specifications. Hence, smart play-out is also a difficult problem. Practically, this process seems valuable and applicable and preliminary results indicate that it scales up quite well.

5.2 Synthesis

5.2.1 Problem

When engineers are satisfied with their requirements specification, they have to design an intraagent specification of the system. Currently, this is a manual operation: Analysts have to consider the whole requirements specification, focus on a single agent, and write down a state-machine implementation for it. This tedious and error-prone process can be automated thanks to *synthesis*. Synthesis is *not* a simple compilation problem; technically, it is a *constructive* proof that the specification is implementable, that it does not contain any conflicting requirements.

Definition 13 (Synthesis). *The synthesis problem is*

- *Input:*
 - an interagent specification $\langle S, U \rangle$,
- *Output:*
 - some intraagent specification $\langle S, f_{Sys} \rangle$ which is a correct implementation of $\langle S, U \rangle$,
 - no, if $\langle S, U \rangle$ has no correct implementation.

This problem is also different from *satisfiability*. Indeed, satisfiability simply asks whether $\mathcal{L}(\mathcal{U})$ is nonempty or, in other words, whether there is a benevolent environment in which some implementation can be deployed in order to fulfill the specification. Synthesis asks whether there is an implementation that is correct, be it deployed in *any* malevolent environment. As an example, consider a uLSC-Spec made of the uLSCs of Fig. 7 and Fig. 8. This specification is satisfiable. To see this, consider an implementation that always returns money to the customer *and* a customer that repeatedly inserts a coin and claims his money back. When put together, they clearly satisfy the specification as they never activate the uLSC of Fig. 7, but only activate and fulfill the uLSC of Fig. 8.

However, there is no correct implementation of this particular uLSC-Spec. It suffices to consider a customer that inserts a coin, asks for coffee, *and* claims his money back. We notice that both precharts are matched. This sequence of events is unavoidable as they are environment-controlled and thus cannot be prevented by an implementation. The main chart of Fig. 7 obliges any implementation to serve coffee and ultimately perform a “coffee” event, but without returning money. The main chart of Fig. 8 does the reverse: It obliges any implementation to return money and to not serve coffee. It is therefore impossible to implement this uLSC-Spec because there exists an environment that can drive *any* implementation to a state in which it will have to violate the specification.

We somehow contributed to the confusion between these two problems in [23]. What we called “consistency” in [23] was actually “satisfiability” and the problem we named “synthesis” corresponds to “extracting a witness of satisfiability.” In comparison, in [24], Harel and Kugler solve the problem of “synthesis,” which they define as verifying that an LSC specification is “consistent.” Their definitions of “consistency” and “synthesis” correspond to the more complicated and really interesting problem we are considering in this section. We will come back to their solution later on.

Notice that the synthesis problem is very lenient regarding outputs. It does not put any constraint on the form of the intraagent specification that can be output. It is simply a strategy, i.e., a function $f_{Sys} : \Sigma^* \rightarrow 2^{\Sigma_{Sys}}$. This function could even be noncomputable. However, there is a nice result in the field of infinite games that applies here: In infinite games with an ω -regular winning condition, if there is a winning strategy, there is a finite-memory winning strategy. Furthermore, ω -regular games are *determined*, i.e., in every game, one of the two players has a winning strategy. Since, as we will explain shortly, a correct implementation is a winning strategy in a two-player game, we can rephrase these two results as: “If there is a correct implementation of any kind, there is also a correct implementation representable as an I/O automaton and vice versa.” This ensures the tractability of the overall since we only need to look at the finite-state case.

5.2.2 Algorithm

In [15], we present a game-theoretic algorithm solving the synthesis problem. It reduces synthesis to solving a two-player game [25].

5.2.3 Complexity

This algorithm runs in time exponential in the size of the specification. The automata $\mathcal{A}_{Env}$ and $\mathcal{A}_{Sys}$ are both exponential in the size of the specification. So is their product. Solving the game is done in time quadratic in the size of these automata. We showed that this algorithm was asymptotically optimal [21].

Theorem 3. *Synthesis is EXPTIME-complete.*

5.2.4 Related Work

Another solution to this problem is provided by 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 superstep approach of [26]: 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 into a strategy automaton. Our solution is an extension of their work. First, 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. Second, 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.

Currently, experimental results are negative. We are not able to synthesize systems from large interagent specifications. This is probably due to the fact that the prototype implementation is poor. We could improve internal data structures and resort to techniques such as partial order reduction [27] or symbolic synthesis [28], [29]. This is still an important direction for future research and synthesis could “like verification, turn out to be theoretically difficult, but practically feasible,” to paraphrase [1].

5.2.5 Distributed Synthesis

We remark that the synthesis problem as we have just defined it is actually too general to exactly match the functionality named “synthesis” in Fig. 4. The development process suggested in Section 1 suggests automatically building a *distributed* intraagent specification from a given interagent specification. The “synthesis” problem that we considered in the previous section asked whether a *global* implementation existed with perfect information about its environment. We are actually interested in *distributed* intraagent specifications, i.e., every system agent must have

its own strategy determining its behavior and this strategy may only look at events visible by this agent, as prescribed by the structural specification.

Definition 14 (Distributed Intraagent Specification). *An intraagent specification $\langle S, f_{Sys} \rangle$ is distributed if f_{Sys} is the composition of a set of strategies $\{f_a\}_{a \in Sys}$, one for each system agent, such that, for every f_a ,*

1. $f_a : \Sigma^* \rightarrow 2^{\Sigma_a}$;
2. $\forall w, w' \in \Sigma^* : w|_{\Sigma_a} = w'|_{\Sigma_a} \implies f_a(w) = f_a(w')$, i.e., if w and w' are the same from a 's point of view, then a shall behave the same way after w or w' .

Distributed synthesis is defined like synthesis, except that its outputs are restricted to distributed intraagent specifications.

Definition 15 (Distributed Synthesis). *The Distributed Synthesis problem is*

- *Input:*
 - *an interagent specification $\langle S, \mathcal{U} \rangle$,*
- *Output:*
 - *some distributed intraagent specification $\langle S, f_{Sys} \rangle$ which is a correct implementation of $\langle S, \mathcal{U} \rangle$,*
 - *no, if $\langle S, \mathcal{U} \rangle$ has no correct implementation.*

This problem is at the core of the process depicted in Fig. 4. However, it turns out to be undecidable. There is thus no algorithm building a distributed implementation of an LSC specification if it exists or refusing to do so if there is no such implementation.

Theorem 4. *Distributed Synthesis is undecidable [21].*

Distributed synthesis is still liberal about its outputs, as it allows strategies with infinite memory, therefore not representable as finite-state I/O automata. This fact was used in the proof of undecidability presented in [21]. However, restricting the problem to strategies representable as I/O automata does not make it decidable. The finite-state case is undecidable, but requires a reduction from Turing machine halting. The infinite state case is highly undecidable [30].

So, how can one cope with undecidability? Several approaches can be followed.

- *Syntactical restrictions* can be applied to the *structural specification*. This obliges the analyst to use some architectural communication patterns. For instance, Pnueli and Rosner showed this for pipeline architectures and special cases of connected architectures [31]. Kupferman and Vardi considered other styles of architectures, such as cycles, and also obtained decidability results [32]. Gastin et al. considered asynchronous games played over traces and obtained decidability results for recognizable trace languages when allowing players to use causal memory [33]. However, these decision procedures are nonelementary. Madhusudan and Thiagarajan gave sufficient conditions to obtain decidability of distributed synthesis [34]. In general, it seems that

trace-closure of the specification is a crucial point, i.e., the interagent specification may not force two events performed independently at two different points across the network to be ordered. uLSC-Specs are not trace-closed because matching the prechart imposes a strong synchronization point between all instances.

- *Incomplete algorithms* can be developed, i.e., algorithms that can fail to find a correct implementation even though it exists. For instance, the “compile-verify” scheme has been followed by some researchers. The essential idea is to apply a predefined implementation scheme to the interagent specification, i.e., to compile it to a distributed intraagent specification and, then, to use model checking techniques to verify that this implementation is correct. Harel et al. proposed a “compile-verify” algorithm in [35]. They generate a state-chart design from a uLSC specification and then model check it to ensure its correctness. The main advantage of this approach resides in the use of orthogonal regions of state charts to encode the concurrent executions of the various uLSCs. This makes the generated state charts more readable as it avoids an exponential blowup in the number of states. We also instantiated the “compile-verify” technique in [36], [37] in which we added an additional “refinement” step. The algorithm we proposed iteratively builds a local strategy for every system agent. First, uLSCs are projected onto the agent a that we want to synthesize. Then, a strategy is built according to the following rule: It proposes an event e if e is required by some scenario, but not forbidden by any other scenario (according to the knowledge of a). The resulting implementation is safe, but can fail to be live. A model checker is used to verify its liveness, which then guarantees that the distributed implementation is correct. If it is not live, a game-based algorithm looks for a live refinement of the implementation.
- *Distribute a centralized implementation* on a set of agents. In [24], Harel and Kugler propose three strategies to distribute intraagent specifications:
 1. build a central controller, driving the execution of the individual agents;
 2. duplicate the central controller in every agent;
 3. remove states that are not relevant to the object in question.
- *Allow extra communication* between system agents. This makes it possible for them to exchange information about their current state and to deduce the actual global state (which precharts are completed and so forth). Using this technique, Sun and Dong build a CSP process from an LSC specification and use algebraic group laws to obtain a distributed implementation [38]. They deal with closed systems and do not allow overlapping executions of the same chart. They are thus studying the problem of “satisfiability” instead of “synthesis.”

- *Syntactical restrictions* can be applied to the *interagent specification* to ensure that every interagent specification is implementable and can be translated to a distributed intraagent specification in a straightforward way. This eliminates the need for a complex distributed synthesis procedure. Homme and Ramsland survey several techniques for implementing uLSCs: play-out, state-chart synthesis following [24], and direct code generation [39]. They propose an algorithm for deriving Java code directly from uLSCs, but this algorithm works only under very strong assumptions on the given uLSC (no user input, messages invoking methods must refer precisely to the called instance, all messages in the main chart must originate from the instance that received the prechart message, there may not be two scenarios with the same prechart). Under these assumptions, the uLSC-Spec is always implementable and the generated intraagent specification is correct.

5.2.6 Synthesis versus Induction versus Compilation

In the literature, the term “synthesis” has been broadly used to denote different kinds of problems. In the field of formal languages and temporal logics, the “synthesis” problem has been well defined since the late ‘80s. It is the same problem that we are considering here: building an implementation that will preserve a specification against any malevolent environment [17], [18], [32], [40], [41], [25], [42], [43]. Before this, some authors treated the problem of synthesis of closed systems, which is synonymous with satisfiability [44], [45].

In scenario-based software engineering, the term “synthesis” is often used with different meanings. To summarize, scenario-based “synthesis” extended in two directions: induction and compilation. These approaches receive their justification depending on the purpose of the scenarios and the phase of the software life cycle in which they are used [46], [5].

1. *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 $E \subseteq W$. The problem is then to build an automaton $\mathcal{A}$ recognizing W . Koskimies et al. [47] have used an algorithm, due to Biermann and Krishnaswamy [48], 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 to the CASE tool FUJABA and validated on a case study from the manufacturing industry [49].

Minimal Adequate Synthesizer (MAS) is an interactive approach to machine learning from a synthesizer based on the Minimal Adequate Teacher

paradigm. MAS finds missing scenarios in the set of scenarios and asks its operator whether such scenarios should be integrated in the specification or not. It has been implemented by Mäkinen and Systä [50] and validated on the “Paderborn New-Rail Technology” Shuttle case study [51].

Hsia et al. use example scenarios, given as an execution tree, to build a grammar and analyze the specification with regard to consistency criteria using automata analysis techniques [52]. 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.

2. *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 [53], [54], [55], [56], [57]. 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” [56]. Techniques have been developed to detect such implied scenarios and report on them [58], [56]. In [59], the authors give syntactical restrictions, namely *causality*, to ensure 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. [60]. The major difference is that their setting is state-oriented, whereas most researchers on scenarios, including us, focus on an event-based setting. In [60], 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. 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. As in our work, the integration of two scenarios relative to the same input obliges the system to answer as specified by both scenarios.

5.3 Verification

5.3.1 Problem

The design model, be it built automatically or manually, will often be modified by engineers. The specification is

likely to be updated after the design model has been built too. This phenomenon is known in Requirements Engineering as “requirements creep”: During the life of a software project, new requirements are discovered and must be integrated to the software product [61]. As a matter of fact, the two artifacts, specification and design, will grow independently. Nevertheless, ultimately, the design model will *have to* comply with the requirements. Therefore, engineers need tools that will help them *verify* this compliance.

Definition 16 (Verification). *The verification problem is*

- *Input:*
 - *an interagent specification $\langle S, \mathcal{U} \rangle$,*
 - *an intraagent specification $\langle S, f_{Sys} \rangle$, with f_{Sys} represented as I/O automata.*
- *Output:*
 - *yes, if $\langle S, f_{Sys} \rangle$ is a correct implementation of $\langle S, \mathcal{U} \rangle$.*
 - *no, otherwise.*

The verification problem is to check that an intraagent specification is a correct implementation of an interagent specification. We will consider several variants of this problem. The most general case considers that the society Sys consists of at least one agent, and that there might be agents out of Sys interacting with them. We will investigate “degenerated” versions, along the following axes:

1. Sys consists of a single agent or several agents (centralized versus distributed agent verification);
2. Env is empty or not (closed versus open agent verification).

5.3.2 Solution

All versions of the verification problem can be solved using the classical automata-theoretic approach to model checking [62]. I/O automata representing f_{Sys} are conjoined into a single I/O automaton $\mathcal{A}$. An automaton $\mathcal{A}_U$ recognizing all counterexamples of the interagent specification (i.e., runs that are Σ_{Env}^s -safe but not Σ_{Sys}^s -safe, or Σ_{Env}^s -live but not Σ_{Sys}^s -live) can also be extracted [10]. This automaton is nondeterministic. It suffices to check that $\mathcal{A} \times \mathcal{A}_U$ has some accepting run to decide whether the intraagent specification is correct or not. Notice that searching accepting runs can be done on-the-fly [63].

In the case of closed verification, the algorithm can be simplified. Since there is no environment agent, the correct implementation condition boils down to checking that all runs are Σ -safe and Σ -live. By Theorem 1, this is equivalent to verifying that the intraagent specification respects the constraints of all scenarios. The algorithm can thus verify *iteratively* that the intraagent automaton $\mathcal{A}$ satisfies each scenario. So, instead of building a big $\mathcal{A}_U$ automaton, a series of small automata can be built and verified. This approach benefits from the simplicity of such automata, which is reflected in Table 2 by the “lower” complexity of this problem. An automaton waits for some arbitrary time in its initial state and nondeterministically guesses to

TABLE 2
Verification Complexity

Distributed	Open	Complete for
No	No	coNP
Yes	No	PSPACE
No	Yes	PSPACE
Yes	Yes	PSPACE

recognize a counterexample. Then, it checks that the prechart is matched and is not followed by the main chart. Such counterexamples are short: They contain as many states as there are events in the uLSC. Besides the blowup occurring when computing the product of I/O automata, the only problem in verification of closed systems comes from the *width* of automata: Because uLSCs are based on partial orders, there can be exponentially many paths to explore, even though each of them is short. In practice, scenario-based specifications tend to be linearly ordered and, hence, have relatively small automata, which makes them amenable to automated verification.

In closed verification, the analyst must come up with an intraagent model of *each* agent, including environment agents. In open verification, some agents belong to the environment, while others are system agents. Environment agents are not explicitly modeled. We then have to deal with an intraagent specification model of system agents only and the question becomes: “Whenever environment agents do behave correctly, does this implementation behave appropriately?”

This problem is “harder” than closed model checking because we cannot check the specification against each uLSC individually. Verification must be restricted to runs that are Σ_{Env} -live and Σ_{Env} -safe. This property is not local to a single scenario, but involves all scenarios simultaneously.

5.3.3 Complexity

The complexity of the verification problem is summarized in Table 2. The proofs and details can be found in the first author’s PhD thesis [10].

5.3.4 Related Work

The problem of LSC-based verification of closed systems has been considered in the literature. Klose et al. follow the approach outlined above and adapt it to LSCs with time information: they turn each uLSC into a timed automaton recognizing all counterexamples [64], [65], [66]. Kugler et al. propose translating uLSCs to temporal logic formulae that can be readily used as input to model checkers. Since uLSCs can be translated to both CTL and LTL, a large number of existing model checkers can be reused [20].

Klose studies open verification in his PhD thesis [67]. He remarks, as we do, that uLSCs can embed hypotheses about the environment as well as commitments about future system behavior. He gives an algorithm for extracting assumptions and dealing with them explicitly. Brill et al. use this technique and explain how LSCs and verification can be integrated in V-software development cycles [68]. The problem of model checking open systems has been

considered by Kupferman et al. They name it *module checking* [69]. They show that, in the case of linear-time specification, the problem is not more difficult than model checking.

Verification can be considered impractical. Success stories are nevertheless accumulating in automated verification. Usage has shown that model checking, despite its high theoretical complexity, could be applied to real-world cases [70]. This situation also applies to the more restricted field of LSC-based verification. Klose et al. report on the successful use of model-checkers to verify Statemate models against LSCs [66], [65], [67]. In general, Damm et al. are accumulating positive results in LSC-based verification. Bunker and Gopalakrishnan have verified hardware bus protocols models [71]. Bontemps checked a model of Ricart-Agrawala's algorithm for mutual exclusion against LSCs [72]. Those preliminary results seem to indicate that, although intraagents are computationally hard to verify, modern verification techniques do scale up and there is some reasonable hope that this problem could be practically addressed.

6 CONCLUSION

6.1 Summary

In this paper, we presented the result of our investigation of a novel way to engineer reactive distributed systems proposed by Harel [1]. This approach relies heavily on automated processes to synthesize code all the way from user scenarios. It is based on two essential artifacts: interagent specifications to express scenarios, because they are close to user requirements, and intraagent specifications to describe the complete behavior of every agent/object of the future system. Intraagent specifications are closer to code and existing code generators can use them as input. We formalized the framework in which this process can take place. On the one hand, interagent specifications are expressed with uLSCs, an expressive variant of MSCs. On the other hand, intraagent specifications consist of strategies, an abstract model that encompasses the popular formalism of state machines. In this framework, we defined three problems of interest:

1. eLSC checking: verifying that the interagent specification allows some user-given example scenario (or eLSC);
2. synthesis: checking that an interagent specification is implementable and automatically building a correct intraagent specification from it, if it exists;
3. verification: checking that a given intraagent specification is a correct implementation of a given interagent specification.

We provided algorithms for each of these problems, except for synthesis, which we proved to be undecidable.

As the reader might notice, we expended serious effort *defining* and *formalizing* modeling languages and relationships between models, hence clearly stating what it means for an intraagent specification (a design model) to *correctly implement* an interagent specification (a set of requirements). This effort paid off: defining problems became straightforward. Essentially, synthesis amounts to asking whether a correct implementation of an interagent specification exists and verification whether an intraagent model is a correct implementation of an interagent specification. Then, the solution of these problems could be presented and *proven*

correct. The main benefit for the reader is that he can actually understand the essence of synthesis and verification algorithms without having to delve into the details of automata and game-theoretic constructions. More simply stated, he can grasp *what* the tools do without having to know *how* they do it.

The formal framework that we proposed made it possible to study the complexity of these problems and explore some variants. All these problems turned out to be intractable, with complexities ranging from coNP-complete to EXPTIME-complete. We also showed that the algorithms presented in this paper are optimal. We put this intractability in perspective by thoroughly reviewing the literature on LSCs. We highlight the fact that there are many positive results in verification and some synthesis algorithms trying to cope with undecidability are being developed.

By precisely defining the problems at play in the development process suggested by Harel and explaining how current research relates to these problems, this paper acts as a research roadmap. Providing such a panorama of research on LSCs is a major contribution of this paper.

Two important issues that we have also studied but that are not discussed in this paper are 1) what could an LSC-based CASE tool look like and 2) how is the complexity of LSC-related problems affected by their various extensions? Both questions are addressed in [10], where we present a tool prototype called REMoRDS and discuss the consequences of three extensions of the LSC language, namely, conditions, real-time, and symbolic instances.

6.2 Directions for Research

We wish that our negative results regarding complexity might act as a guide for future research. There are several ways to live with intractability. First, problems may be relaxed in order to work on subsets of LSCs for which efficient algorithms exist. Second, algorithms can be made efficient but imprecise. Third, precise algorithms can be devised with improvements that allow them to scale up very well, but that, in the worst case, still have a very high complexity. This trend is being witnessed, with success, in the field of model checking, with approaches such as symbolic model checking, SAT-based model checking, predicate abstraction, or Craig interpolation. Theoretically, these techniques do not bring any improvement, but, practically, they make ever larger systems amenable to automated verification.

Future research should also consider the problem of inserting LSCs in existing computer systems. Our approach is "greenfield": We start from scratch and generate a complete system. However, in practice, an existing implementation may already exist. We also drew a clear line between inter- and intraagent specifications. However, in practice, both aspects will be mixed, as some agents, for instance, are easier to specify using state machines, even during requirements engineering. LSCs and existing implementations could be related through the generation of test suites, too.

We also explained that existing scenario-based synthesis algorithms belong to two different categories: induction and compilation. It would be interesting to understand the interplay between induction, verification, compilation, and synthesis. For instance, induction can be used in combination with verification to guess an intraagent specification from

examples and ensure that the created system preserves safety-critical properties.

Finally, a major appeal of scenario-based requirements engineering is the idea of starting from unrelated requirements chunks and composing them automatically. It is also the greatest obstacle to its practical use: Our research has shown that the high complexity of LSCs comes precisely from this unstructured concurrent composition of scenarios. Efficiency could be improved by finding some trade-off between an easy-to-analyze structured specification and an easy-to-manipulate loose specification.

REFERENCES

- [1] D. Harel, "From Play-In Scenarios to Code: An Achievable Dream," *Computer*, vol. 34, no. 1, pp. 53-60, Jan. 2001.
- [2] L. Lamport and N. Lynch, *Handbook of Theoretical Computer Science*, vol. B, chapter 18, "Distributed Computing: Models and Methods," pp. 1157-1199. Amsterdam and Cambridge, Mass.: Elsevier-MIT Press, 1990.
- [3] D. Harel and A. Pnueli, "On the Development of Reactive Systems," *NATO ASI Series, K. Apt, ed.*, vol. F-13, pp. 477-498. New York: Springer, Jan. 1985.
- [4] J. Mukerji and J. Miller, "MDA Guide v 1.0.1 (OMG)," Mar. 2003, <http://www.omg.org/mda/>.
- [5] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer, "Scenario Usage in System Development: A Report on Current Practice," *IEEE Software*, vol. 15, no. 2, pp. 34-45, Mar./Apr. 1998.
- [6] W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts," *Formal Methods in System Design*, vol. 19, no. 1, pp. 45-80, 2001.
- [7] D. Harel and R. Marelly, *Come, Let's Play! Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.
- [8] "MSC-2004: ITU-T Recommendation Z.120: Message Sequence Chart (MSC)," Int'l Telecomm. Union (prev. CCITT), 2004. Draft version, still unapproved.
- [9] *OMG UML Specification (2.0)*, Object Management Group (UML Revision Task Force), Sept. 2003, <http://www.omg.org/uml>.
- [10] Y. Bontemps, "Relating Inter-Agent and Intra-Agent Specifications (the Case of Live Sequence Charts)," PhD dissertation, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique (Univ. of Namur, Computer Science Dept.), Apr. 2005.
- [11] M.V. e Cengarle and A. Knapp, "UML 2.0 Interactions: Semantics and Refinement," *Proc. Third Int'l Workshop Critical Systems Development with UML (CSDUML '04)*, J. Jürjens, E.B. Fernandez, R. France, and B. Rumpe, eds., pp. 85-99, 2004.
- [12] Y. Bontemps, "On the Semantics of UML 2.0 Interaction Diagram," technical report, Univ. of Namur, Institut d'Informatique, 2004.
- [13] M. Jackson, *Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices*. Addison Wesley, 1995.
- [14] N.A. Lynch and M.R. Tuttle, "An Introduction to Input/Output Automata," *CWI Quarterly*, vol. 2, no. 3, pp. 219-246, 1989.
- [15] Y. Bontemps, P.-Y. Schobbens, and C. Löding, "Synthesis of Open Reactive Systems from Scenario-Based Specifications," *Fundamenta Informaticae*, vol. 62, no. 2, pp. 139-169, July 2004.
- [16] C. Gunter, A.E. Gunter, L.M. Jackson, and P. Zave, "A Reference Model for Requirements and Specification," *IEEE Software*, vol. 17, no. 3, pp. 37-43, May/June 2000.
- [17] M. Abadi, L. Lamport, and P. Wolper, "Realizable and Unrealizable Specifications of Reactive Systems," *Automata, Languages, and Programming, Proc. 16th Int'l Colloquium (ICALP '89)*, G. Ausiello, M. Dezani-Ciancaglini, and S.R.D. Rocca, eds., July 1989.
- [18] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proc. 16th Ann. ACM Symp. Principles of Programming Languages*, pp. 179-190, 1989.
- [19] G. Berry, "The Foundations of Esterel," *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, eds. MIT Press, 1998, <ftp://ftp-sop.inria.fr/meije/esterel/papers/foundations.pdf>.
- [20] H. Kugler, D. Harel, A. Pnueli, L. Yuan, and Y. Bontemps, "Temporal Logic for Live Sequence Charts," *Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS '05)*, Apr. 2005.
- [21] Y. Bontemps and P.-Y. Schobbens, "The Complexity of Live Sequence Charts," *Proc. Foundations of Software Science and Computation Structure (FoSSACS '05)*, V. Sassone, ed., pp. 364-378, Apr. 2005.
- [22] D. Harel, H. Kugler, R. Marelly, and A. Pnueli, "Smart Play-Out of Behavioral Requirements," *Proc. Fourth Int'l Conf. Formal Methods in Computer-Aided Design (FMCAD '02)*, 2002. Also available as Technical Report MCS02-08, The Weizmann Inst. of Science.
- [23] Y. Bontemps and P. Heymans, "Turning High-Level Live Sequence Charts into Automata," *Proc. Scenarios and State-Machines: Models, Algorithms, and Tools (SCESM) Workshop, 24th Int'l Conf. Software Eng. (ICSE 2002)*, May 2002, <http://www.cs.tut.fi/tsysta/ICSE/papers/>.
- [24] D. Harel and H. Kugler, "Synthesizing State-Based Object Systems from LSC Specifications," *Int'l J. Foundations of Computer Science*, vol. 13, no. 1, pp. 5-51, Feb. 2002. Preliminary version *Proc. Fifth Int'l Conf. Implementation and Application of Automata (CIAA 2000)*, July 2000.
- [25] *Automata Logics, and Infinite Games: A Guide to Current Research*, E. Grdel, W. Thomas, and T. Wilke, eds. Springer, Nov. 2002, <http://link.springer.de/link/service/series/0558/tocs/t2500.htm#toc2500>.
- [26] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The STATEMATE Approach*. McGraw-Hill, 1998.
- [27] P. Wolper and P. Godefroid, "Partial-Order Methods for Temporal Verification," *Proc. Int'l Conf. Concurrency Theory (CONCUR '93)*, pp. 233-246, 1993.
- [28] K. McMillan, *Symbolic Model Checking*. Kluwer Academic, 1993.
- [29] N. Wallmeier, P. Hütten, and W. Thomas, "Symbolic Synthesis of Finite-State Controllers for Request-Response Specifications," *Proc. Eighth Int'l Conf. Implementation and Application of Automata (CIAA 2003)*, O.H. Ibarra, and Z. Dang eds., pp. 11-22, 2003.
- [30] R. Rosner, "Modular Synthesis of Reactive Systems," PhD dissertation, The Weizmann Inst. Science, Rehovot, Israel, Apr. 1992.
- [31] A. Pnueli and R. Rosner, "On the Synthesis of an Asynchronous Reactive Module," *Automata, Languages, and Programming, Proc. 16th Int'l Colloquium (ICALP)*, G. Ausiello, M. Dezani-Ciancaglini, and S.R.D. Rocca eds., pp. 652-671, July 1989.
- [32] O. Kupferman and M.Y. Vardi, "Synthesizing Distributed Systems," *Proc. 16th IEEE Symp. Logic in Computer Science*, July 2001.
- [33] P. Gastin, B. Lerman, and M. Zeitoun, "Distributed Games and Distributed Control for Asynchronous Systems," *Proc. Latin Am. Theoretical Informatics (LATIN '04)*, Apr. 2004, <http://www.liafa.jussieu.fr/~versydis/data/publications/GastinLermanZeitoun-LATIN04.pdf>.
- [34] P. Madhusudhan and P. Thiagarajan, "A Decidable Class of Asynchronous Distributed Controllers," *Proc. CONCUR '02*, 2002, <http://www.cis.upenn.edu/madhusud/concur02.ps.gz>.
- [35] D. Harel, H. Kugler, and A. Pnueli, "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements," *Formal Methods in Software and Systems Modeling*, H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, eds., pp. 309-324, 2005.
- [36] Y. Bontemps and P. Heymans, "As Fast as Sound (Lightweight Formal Scenario Synthesis and Verification)," *Proc. Third Int'l Workshop Scenarios and State Machines: Models, Algorithms, and Tools (SCESM '04)*, H. Giese, and I. Krüger, eds., pp. 27-34, May 2004. <http://www.info.fundp.ac.be/~ybo>.
- [37] Y. Bontemps, P. Heymans, and P.-Y. Schobbens, "Lightweight Formal Methods for Scenario-Based Software Engineering," *Scenarios, T. Systä, and S. Leue, eds.*, pp. 174-192. Springer, 2005.
- [38] J. Sun and J.S. Dong, "Synthesis of Distributed Processes from Scenario-Based Specifications," *Formal Methods (FM 2005)*, 2005.
- [39] T. Homme and J.E. Ramsland, "From Live Sequence Charts to Implementation (a Study of the LSC Specification, the Execution of Behavioral Requirements and Exploring the Possibilities to Use an LSC Model to Generate Java Code)," master's thesis, Agder Univ. College, Agder, Norway, 2003.
- [40] O. Kupferman and M.Y. Vardi, "Synthesis with Incomplete Information," *Proc. Second Int'l Conf. Temporal Logic*, pp. 91-106, July 1997.
- [41] M.Y. Vardi, "An Automata-Theoretic Approach to Fair Realizability and Synthesis," *Proc. Seventh Int'l Conf. Computer Aided Verification*, P. Wolper, ed., pp. 267-278. Springer Verlag, 1995, <http://citeseer.nj.nec.com/article/vardi95automatatheoretic.html>.

- [42] P.J.G. Ramadge and W.M. Wonham, "The Control of Discrete Event Systems," *Proc. IEEE*, special issue on dynamics of discrete event systems, vol. 77, no. 1, pp. 81-98, 1989.
- [43] H. Wong-Toi and D.L. Dill, "Synthesizing Processes and Schedulers from Temporal Specifications," *Computer-Aided Verification '90: Proc. DIMACS Workshop*, E.M. Clarke and R.P. Kurshan, eds., pp. 177-186, 1991.
- [44] Z. Manna and P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications," *ACM Trans. Programming Languages and Systems (TOPLAS)*, vol. 6, no. 1, pp. 68-93, 1984.
- [45] E.A. Emerson and E.M. Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons," *Science of Computer Programming*, vol. 2, no. 3, pp. 241-266, Dec. 1982.
- [46] D. Amyot and A. Eberlein, "An Evaluation of Scenario Notations for Telecommunication Systems Development," *Proc. Ninth Int'l Conf. Telecomm. Systems (9ICTS)*, Mar. 2001.
- [47] K. Koskimies, T. Männistö, T. Systä, and J. Tuomi, "SCED: A Tool for Dynamic Modelling of Object Systems," Technical Report A-1996-4, Dept. of Computer Science, Univ. of Tampere, Finland, July 1996.
- [48] A.W. Biermann and R. Krishnaswamy, "Constructing Programs from Example Computations," *IEEE Trans. Software Eng. (TSE)*, vol. 2, no. 3, pp. 141-153, Sept. 1976.
- [49] I. Diethelm, L. Geiger, T. Maier, and A. Zündorf, "Turning Collaboration Diagram Strips into Storycharts," *Proc. Scenarios and State-Machines: Models, Algorithms, and Tools (SCESM) Workshop 24th Int'l Conf. Software Eng. (ICSE 2002)*, May 2002, <http://www.cs.tut.fi/~tsysta/ICSE/papers/>.
- [50] E. Mäkinen and T. Systä, "Minimally Adequate Teacher Synthesizes Statechart Diagrams," *Acta Informatica*, vol. 38, pp. 235-259, 2002.
- [51] J. Koskinen, E. Mäkinen, and T. Systä, "Minimally Adequate Teacher Synthesizes Shuttles, Too," *Proc. Third Int'l Workshop Scenarios and State Machines: Models, Algorithms, and Tools (SCESM '04)*, H. Giese, and I. Krüger, eds., May 2004.
- [52] P. Hsia, J. Samuel, J. Gao, D. Kund, Y. Toyoshima, and C. Chen, "Formal Approach to Scenario Analysis," *IEEE J.*, pp. 33-41, Mar. 1994.
- [53] I. Krüger, R. Grosu, P. Scholz, and M. Broy, *From MSCs to Statecharts*, F.J. Rammig, ed. Kluwer Academic, 1999.
- [54] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios," *Proc. 22nd Int'l Conf. Software Eng. (ICSE 2000)*, pp. 314-323, June 2000.
- [55] S. Uchitel and J. Kramer, "A Workbench for Synthesizing Behaviour Models from Scenarios," *Proc. 23rd IEEE Int'l Conf. Software Eng. (ICSE '01)*, 2001.
- [56] R. Alur, K. Etessami, and M. Yannakakis, "Inference of Message Sequence Charts," *Proc. 22nd Int'l Conf. Software Eng.*, pp. 304-313, 2000.
- [57] T. Ziadi, L. Hélouët, and J.-M. Jézéquel, "Revisiting Statechart Synthesis with an Algebraic Approach," *Proc. 26th Int'l Conf. Software Eng. (ICSE)*, pp. 242-251, May 2004.
- [58] S. Uchitel, J. Kramer, and J. Magee, "Detecting Implied Scenarios in Message Sequence Chart Specifications," *Proc. Joint Eighth European Software Eng. Conf. and Ninth ACM SIGSOFT Symp. Foundation of Software Eng. (ESEC/FSE-01)*, V. Gruhn, ed., pp. 74-82, Sept. 2001.
- [59] B. Finkbeiner and I.H. Krüger, "Using Message Sequence Charts for Component-Based Formal Verification," *Proc. OOPSLA 2001 Workshop Specification and Verification of Component-Based Systems*, Oct. 2001.
- [60] J. Desharnais, M. Frappier, R. Khédri, and A. Mili, "Integration of Sequential Scenarios," *IEEE Trans. Software Eng.*, vol. 24, no. 9, pp. 695-708, Sept. 1998.
- [61] K.E. Wiegers, *Software Requirements*, second ed. Microsoft Press, Feb. 2003.
- [62] M. Vardi and P. Wolper, "Automata-Theoretic Techniques for Modal Logics of Programs," *J. Computer and System Science*, vol. 32, no. 2, pp. 183-221, Apr. 1986.
- [63] S. Schwoon and J. Esparza, "A Note on On-the-Fly Verification Algorithms," *Tools and Algorithms for Analysis and Construction of Systems (TACAS 2005)*, N. Halbwachs, and L.D. Zuck, eds., pp. 174-190, 2005.
- [64] J. Klose and H. Wittke, "An Automata Based Interpretation of Live Sequence Charts," *Proc. TACAS (Tools and Algorithms for the Construction and Analysis of Systems)*, T. Margaria, and W. Yi, eds., pp. 512, Apr. 2001.
- [65] J. Klose, T. Kropf, and J. Ruf, "A Visual Approach to Validating System Level Designs," *ISSS '02: Proc. 15th Int'l Symp. System Synthesis*, pp. 186-191, 2002.
- [66] J. Bohn, W. Damm, J. Klose, A. Moik, and H. Wittke, "Modeling and Validating Train System Applications Using Statemate and Live Sequence Charts," *Proc. Conf. Integrated Design and Process Technology (IDPT 2002)*, H. Ehrig, B.J. Krämer, and A. Ertas, eds., 2002, <http://ca.informatik.uni-oldenburg.de/cgi-bin/bibsearch?author=Wittke>.
- [67] J. Klose, "Live Sequence Charts: A Graphical Formalism for the Specification of Communication Behavior," PhD dissertation, Carl von Ossietzky Universität, Oldenburg, 2003.
- [68] M. Brill, R. Buschermöhle, W. Damm, J. Klose, B. Westphal, and H. Wittke, "Formal Verification of LSCs in the Verification Process," *Integration of Software Specification Techniques for Applications in Eng.*, M. Grosse-Rhode, W. Reif, E. Schnieder, and E. Westkämper, eds., 2004.
- [69] O. Kupferman, M.Y. Vardi, and P. Wolper, "Module Checking," *Information and Computation*, vol. 164, pp. 322-344, 2001.
- [70] E.M. Clarke and J.M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, no. 4, pp. 626-643, Dec. 1996.
- [71] A. Bunker and G. Gopalakrishnan, "Verifying a Virtual Component Interface-Based PCI Bus Wrapper Using an LSC-Based Specification," Technical Report UUCS-02-004, School of Computing, Univ. of Utah, 2002.
- [72] Y. Bontemps, "Automated Verification of State-Based Specifications against Scenarios (A Step towards Relating Inter-Object to Intra-Object Specifications)," master's thesis, Univ. of Namur, Namur, Belgium, June 2001.



Yves Bontemps received the MSc and PhD degrees in computer science from the University of Namur, Belgium. In 2005, he completed his PhD thesis under the supervision of Professor Pierre-Yves Schobbens. His doctoral studies were funded by a Belgian FNRS "Research Fellow" grant. He is currently a full-time consultant with SmalS-MvM/eGov, Belgium.



Patrick Heymans received the MSc degree and the PhD degree in computer science from the University of Namur. He is a full professor at the PRECISE lab, Computer Science Department, University of Namur. His research interests include semantics of modeling languages and requirements engineering. In 2001, he completed a PhD thesis on requirements animation under the supervision of Professor Pierre-Yves Schobbens and Professor Eric Dubois.



Pierre-Yves Schobbens received MSc degrees and the PhD degree from University of Louvain, Belgium. He is a professor at the PRECISE lab, Computer Science Department, University of Namur. His research interests include formal semantics, logics in computer science, model checking, UML semantics and agent-oriented specification. He is a member of the IEEE Computer Society.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.