ELSEVIER

Information and Software Technology 41 (1999) 697–713

INFORMATION AND SOFTWARE TECHNOLOGY

A service creation environment based on scenarios ☆

R. Dssouli[a,*], S. Somé[b], J. Vaucher[a], A. Salah[a]

[a]*Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, C.P.6128, succ centre ville, Montréal, Quebec, Canada H3C 3J7*
[b]*KBRE, SITE, University of Ottawa, Ottawa, Canada*

Abstract

Scenarios are often constructed for illustrating example runs through reactive system. Scenarios that describe possible interactions between a system and its environment are widely used in requirement engineering, as a means for users to communicate their functional requirements. Various software development methods use scenarios to define user requirements, but often lack tool support. Existing tools are graphical editors rather than tool support for design. This paper presents a service creation environment for elicitation, integration, verification and validation of scenarios. A semi-formal language is defined for user oriented scenario representation, and a prototype tool implementing an algorithm that integrates them for formal specification generation. This specification is then used to automatically find and report inconsistencies in the scenarios.

Keywords: Scenario; Service creation environment; Requirement engineering

1. Introduction

Requirements engineering is the first step in the software engineering life cycle. Documents that arise from this step are informal specifications. The automation of this step is recognized as a hard task as formal methods are not easily accepted and automatic computation of natural languages is not yet ready. Passing from the informal specifications to the formal ones is still an open issue. One of the proposed approaches is to offer a simple specification language that is close to the natural language to be accepted, and hides the formal methods. Timed automata are used as a target formal method because of the direct relationship between formal interpretation of scenarios, and partial runs of the described system. Requirements engineering includes elicitation, understanding and representation of the user's need for a reactive system. It is a critical task, which induces a great number of software failures [1]. An important part of these failures come from undetected inconsistencies and incompleteness of user requirements. Errors are also introduced during the conversion of requirements to specifications.

A scenario is a possible interaction sequence between a system and its environment. Each scenario describes a partial behavior arising in a restricted situation (a precondition) [2]. Scenarios are appropriate for *reactive systems* (like real-time controllers, embedded systems, communicating systems, etc.) external behavior description. These systems react to *stimuli* from their environment according to their history. We extended the scenarios with timing constraints, because the behavior of an important part of the reactive system is constrained by timing requirements.

2. Related works

During the last few years, scenarios have been used for elicitation of systems specification in several requirement specification methods. The *partial* nature of scenarios make them suitable to represent parts of a system behavior. This allows several users with different views or users of the same system, to provide different but possibly overlapping scenarios to describe its behavior. However, these scenarios may include contradictions, and the system behavior may not be completely defined by the set of scenarios provided.

In order to obtain a global behavior model, scenarios must be composed, and it must be possible to deal with their contradictions and incompleteness. The composition aims to integrate scenarios in a whole specification model. We distinguish two composition methods; the declarative one, in which the composition is made according to the manner explicitly indicated by the user [3–6], and the inductive composition that uses inductive rules to insert a scenario

☆ This work was partly funded by NSERC and the Ministry of Industry, Commerce, Science and Technology, Quebec, under the IGLOO project organized by the Centre de Recherche Informatique de Montreal.

* Corresponding author. Tel.: + 1-514-343-6111; fax: + 1-514-343-5834.

E-mail addresses: dssouli@iro.umontreal.ca (R. Dssouli); salah@iro.umontreal.ca (A. Salah)

0950-5849/99/$ - see front matter
PII: S0950-5849(99)00031-2

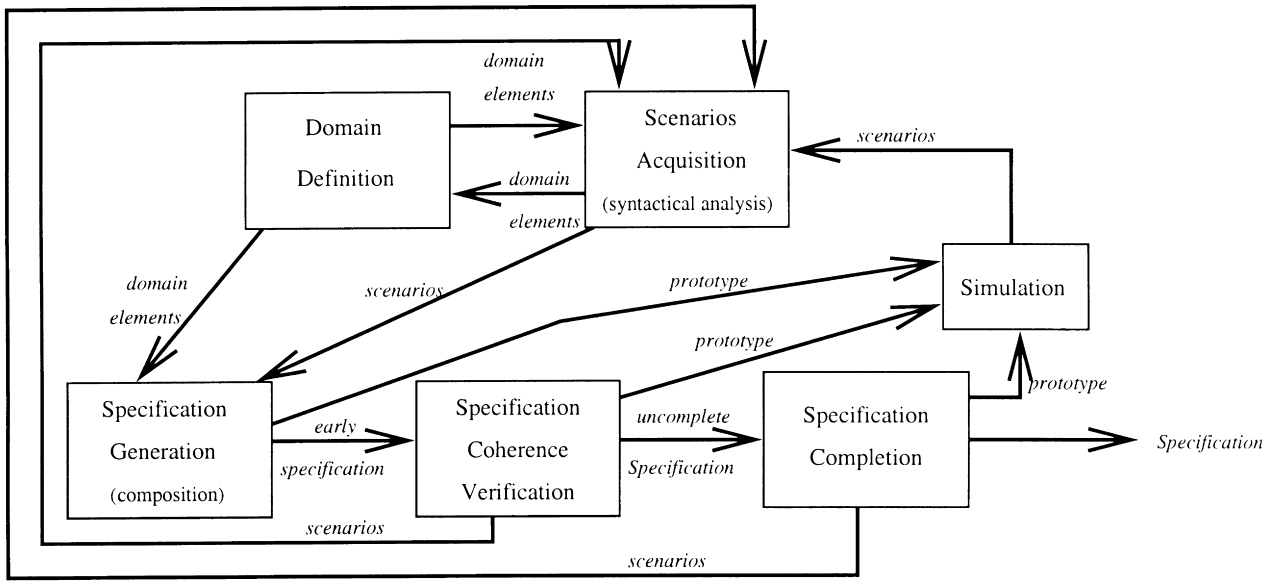

Fig. 1. Activities of a prototype tool for specification construction. Arrows show the succession of activities and the elements transmitted between them.

in an existing specification [7,8] which may be initially empty.

The automation of tasks performed on scenarios during composition, analysis, simulation and prototype generation require a formal representation of scenario. Several existing formalisms have been used. In Ref. [9], the authors use *Message Sequences Charts* (MSC) [10] as the scenarios' formal representation. These scenarios are integrated in a global *Message Flow Graph* (MFG) [11], which is checked for some syntactic properties. Glinz [4] describes his scenarios in Harel's state chart formalism [12] for which the scenarios' composition templates were defined. Overlapping scenarios are forbidden, hence they must be decomposed into disjoint ones, which can then be composed according to the defined templates. Amyot et al. [3] have chosen a *Use Cases Maps* (UCM) [13] representation for their scenarios. The UCM formalism may include architectural requirement that eases the scenarios' composition into a LOTOS specification. The latter serves to generate test sequences. Koskimies and Mäkinen [7] use scenarios to describe partial behaviors of object classes in the OMT method. Scenarios are formalized as trace diagram. Then, a Biermann [14] inductive algorithm synthesizes a minimal state machine that contains exactly the scenarios' traces. A state is defined as an abstraction of object attribute values, but actions lack of semantic that may lead to a non convenient insertion of a scenario. Hsia et al. [6] constructed scenarios in a tree structure by considering at each node all possible events, according to a user's view. Then scenarios are translated to grammar that will be inserted into a finite conceptual state machine. The advantage of this method is the completeness of its generated specification. The Z algebraic specification language is used as a target of composed scenarios in Ref. [15].

Time is an important concept in the emerging networks and applications. Hence, scenarios' representation must be able to express time constraints that reflect real time system situations. In Refs. [16,17], scenarios are expressed using MSC extended by time constraint. There is no scenario composition in Ref. [16], but the authors propose an algorithm for checking time consistency. However, composition of scenarios in Ref. [17] is based on higher MSC (HMSC), which is a connected direct graph of MSC. The timing coherence is treated in a similar fashion as Ref. [16].

Our work encompasses several activities of scenario based requirements engineering reported in the literature. These activities include scenario modeling [18], scenario formalization [6], scenario composition [4,7], verification of inconsistencies, and completion [6,19,20]. Our main contribution is an integration of all these activities in a framework for the automatic generation of valid and complete specifications from user scenarios that support temporal constraints.

This paper describes these activities and a prototype tool for scenario based requirements engineering that provides an automatic support for scenarios discovery, acquisition, analysis, validation and completion. This environment uses a *semi-formal* language for scenario description, and algorithms for scenario composition, verification and validation. The elicitation process is supported by an early execution of scenarios.

The paper is organized as follows. In Section 3, an overview of the prototype tool is provided. Sections 4–8 > detail the prototype tool, and Section 9 concludes the paper. An Appendix is added to show the applicability of the proposed work, but it is not necessary for understanding the actual paper.

System Component: ATM
- **attribute**: number of attempts
  - **values**: num
- **attribute**: cash
  - **values**: available
- **attribute**: display
  - **values**: card insert prompt, pin enter prompt, warning, service menu, amount menu
- **operation**: display card insert prompt
  - **added-conditions**: display is card insert prompt
- **operation**: display pin enter prompt
  - **added-conditions**: display is pin enter prompt
- **operation**: display amount menu
  - **added-conditions**: display is amount menu
- **operation**: display warning
  - **added-conditions**: display is warning
- **operation**: display service menu
  - **added-conditions**: display is service menu
- **operation**: issue cash
- **operation**: check id
  - **added-conditions**: id is valid **or** id is invalid
- **operation**: check amount
  - **added-conditions**: amount is valid **or** amount is invalid
- **operation**: retain card
- **operation**: eject card
- **operation**: ask amount
- **operation**: reinit
  - **added-conditions**: display is card insert prompt
  - **withd-conditions**: all

Environment Component: CUSTOMER
- **attribute**: id
  - **values**: valid, invalid
- **attribute**: amount
  - **values**: valid, invalid
- **operation**: insert card
- **operation**: enter pin
  - **added-conditions**: increment number of attempts
- **operation**: enter amount
- **operation**: select cash withdrawal

Fig. 2. ATM application domain.

3. An overview of activities supported

Fig. 1 shows the activities supported by the prototype tool and their relationships.

An application domain definition concerns the enumeration and description of the system and its environment features. It includes objects making the system and its environment, and relevant environmental factors to the application (temperature, pressure, etc.). A system component may be a physical unit (display, actuator, motor, etc.) or a piece of software performing specific functions, and may be composed of sub-components. A system component description includes its attributes and operations. At the beginning of a system requirement engineering process, an application domain definition is generally made by some elements known by users. This earlier definition is, however, generally incomplete and may be completed during a scenario acquisition.

Scenario acquisition includes obtaining them from users and their syntactical analysis. They are described in a *semi-formal* language based on *structured english* or using a graphical representation. A graphical language is based on MSCs. Our extention concerns addition of symbols for delays and conditions. We have constructed a graphical editor for scenario description, and translation to textual representation. As a scenario describes partial behaviors, and all of them may not be known at the beginning of the development process, scenarios are acquired one by one, and the whole behavior of the system is incrementally constructed. A scenario is expressed according to the constituents of the application domain. The syntactical analysis of scenarios uses elements of the application domain. Missing elements referred in scenarios might, however, be added to the application domain description during scenario analysis.

The specification generation activity analyzes a scenario and merges their partial behavior with that obtained from the previously acquired scenarios'. This activity produces an early specification that includes all the scenarios. After obtaining an early specification or a complete specification, a corresponding SDL (Specification and Description Language) specification can be generated. We have defined an operational semantic that is used to translate a timed automata to an SDL specification [21]. In spite of a high abstraction degree and lack of details, early specification can be used to show a system general behavior by prototype simulation. Prototyping is well used within requirements engineering methods based on scenarios, as

```
1a WHEN ATM display is card insert prompt
   IF CUSTOMER inserts card THEN ATM displays pin enter prompt
   IF CUSTOMER enters pin THEN ATM checks id
      TRANSITION DELAY 60 sec ON EXPIRY ATM ejects card, ATM reinits
   IF id is invalid AND numbers attempts becomes greater than 3 THEN
     ATM retains card, ATM reinits
```

Fig. 3. Scenario 1a.

it allows to reproduce all the partial behaviors included in them, and to uncover other behaviors stemming from them. Simulation may also induce users to modify their requirements, and make changes early in the development process.

The generation of specifications does not make any assumptions on the set of the scenarios used. Scenarios (as partial descriptions), possibly gathered from different users, may include contradictions. The early specification obtained reflects inconsistencies in the scenarios, and coherence verification aimed at finding them. This activity may lead to a modification of some scenarios previously acquired.

The set of scenarios provided may not completely define the behavior of the system. A specification generated might lack important behaviors and the specification completion activity aims at adding these missing definitions that may cause the need for additional scenarios.

4. The application domain description

The application domain is an enumeration of the system and its environment elements. It includes environmental factors relevant to the system behavior. Indeed, behavior of reactive systems may depend on elements of the environment such as temperature or pressure. The domain description enumerates such elements and their value types. We distinguish between continuous values (numerical attributes) and discrete ones.

A system is made up of one or more interacting components. A component may be a physical unit (sensor, display, motor, etc.) or a piece of software. Components may also be composed of sub-components, and they may include attributes, and allow operations. A component description includes its attributes (with their possible values) and

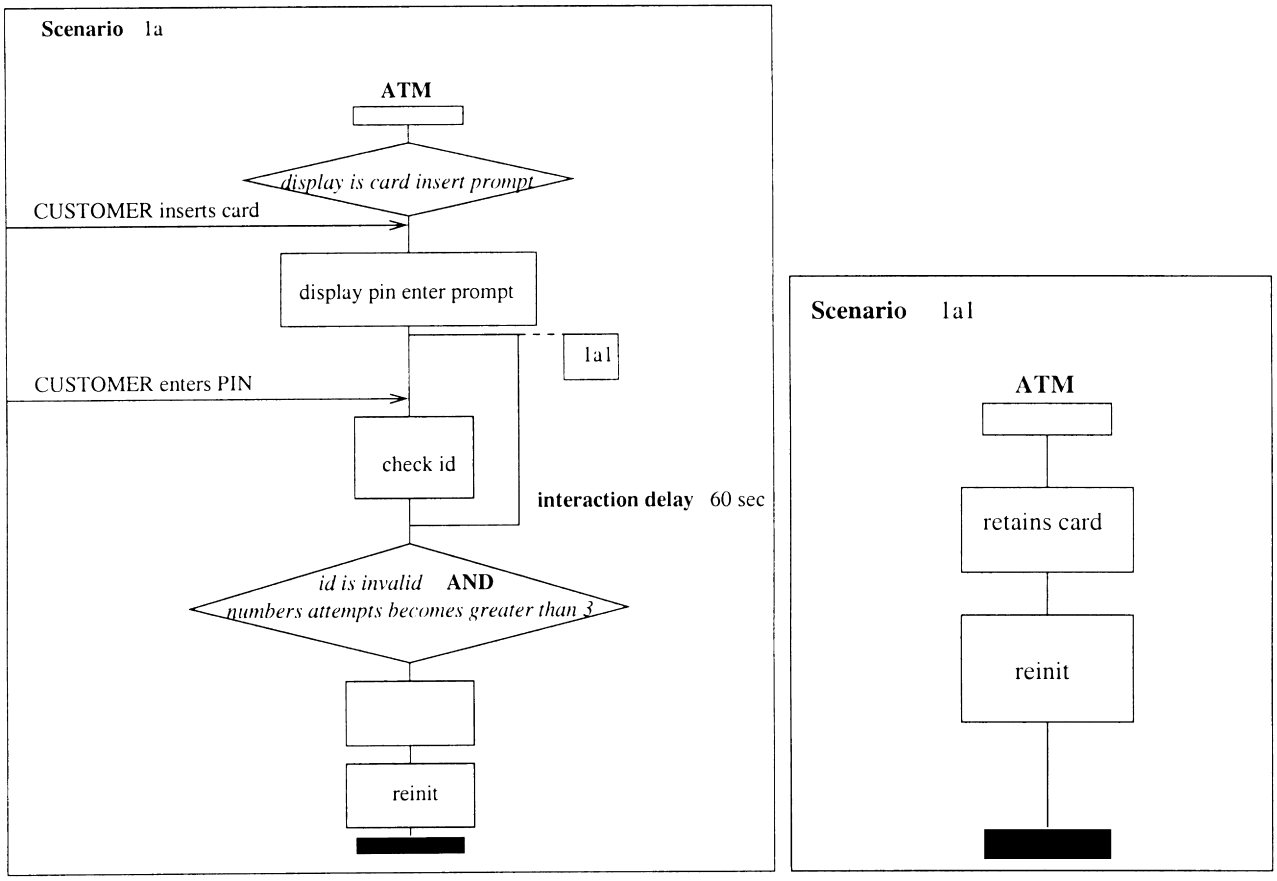

Fig. 4. Graphical representation of scenario 1a. The Customer being a part of the environment is not represented. The system described here includes a single component. Several interacting components may be described using an axis for each of them, and arrows showing their interactions.

operations. The effects of operations are included as modifications induced on the system and environment by executing them. These effects are described as two sets of *conditions*: *conditions withdrawn* and *conditions added*.

We have developed a language for the application domain description. Fig. 2 shows an example of an Automated Teller Machine (ATM) where application domain description uses this notation.

The application involves a system (ATM) and its environment (CUSTOMER). The ATM's attributes include the customer's *numbers of attempts*, the *cash* in the ATM, and the *display* of the ATM. The attributes of the customer are his identification number *id* and the *amount* specified in the transaction. The description includes possible values of each attribute. As an example, the ATM *display* may have five possible "values", and the *number of attempts* any numerical value. The description includes operation of components with their effects as *added* and *withdrawn* conditions. An operation may withdraw a specific condition, conditions on an attribute, all conditions added by an operation, or all conditions known. As an example, the operation *check amount* adds condition *id is valid* or *id is invalid*, while the operation *reinit* withdraws all the conditions and adds the condition *display is card insert prompt*.

The domain description in Fig. 2 is limited to the elements needed for the examples used in this paper. Only attributes and operations that appear in our examples are enumerated. The domain description is partial, because the level of abstraction is such that it includes only elements apparent from outside the system. Some elements may also be known at the beginning, and others added in the course of a scenario acquisition.

5. A language for scenario representation

We have defined *semi-formal* textual and graphical languages for a scenario, which uses some natural language sentences.

Fig. 3 shows an example of a scenario that involves a Customer and an ATM in the textual form, and Fig. 4 shows the same example using the graphical notation. A scenario is a series of *interactions* that follow each other in sequential order, when a pre-condition is verified. Each interaction is a couple (*stimuli*/*reactions*). A *stimulus* may be an operation at the system interface or the occurrence of a certain situation. In scenario 1a, operations performed by a CUSTOMER are *stimuli*, while those of the ATM are *reactions* to them. The third interaction is activated by a condition.

Conditions and operations are parts of a natural language sentence. Conditions describe *situations* prevailing within the system and its environment, or *changes*. A *situation* is written as an *adjectival clause*, seeking a certain quality on an entity of an application domain. As an example, in scenario 1a, the pre-condition is ATM *display is card insert prompt*. It is a natural language description of a *condition* where the ATM attribute display has the property *card insert prompt*. Changes express modifications of features of entities in a system or an environment.

"Operations" are active sentences in which a component, or "verb" performs an action. Another component may be added in the active sentence, as the one affected by the operation.

Delays are introduced for timing constraint definition in scenarios. There are two kinds of *delays*: *triggering delays* and *completion delays*. A *triggering delay* applies to an interaction. Its constraints are at the starting moment. After completion of the interaction that precedes it, a *minimal*, *exact* or *maximal* time amount that must be spent before its activation. *Completion delays* are put on interactions or scenarios, so that all their operations are completed before a given moment. Expiry operations are associated with *completion delays*, and are executed when timing terminates.

Scenario 1a includes a *completion delay* on its second interaction, and specifies that it should be completed 60 s after the end of the first interaction.

The graphical language is based on MSC [22]. We constructed an editor for the graphical scenarios' description, and translation to textual representation. This editor is added on the top of the scenario acquisition. Compared to Z12, our extension mainly concerns the addition of symbols for *delays* and *conditions*.

A scenario is formally represented as a quadruple $\langle R_{\text{num}},\ R_{\text{P}},\ R_{\text{I}},\ R_{\text{D}}\rangle$, where:

- R_{num} is a scenario number
- R_{P} is the scenario *precondition*, a set of conditions $\langle E,V\rangle$ where E is an entity and V a possible value of E
- R_{I} is a sequence of *interactions* $[I_1,\ldots,I_n]$.
  Each $I_i = \langle ind_i, D_i, R_i, ID_i\rangle$ with ind_i an initial delay, $D_i = [d_{i_1},\ldots,d_{i_n}]$ a set of *stimuli*, $R_i = [r_{i_1},\ldots,r_{i_n}]$ are reactions of the system, $ID_i = \langle dv_i, IDR_i\rangle$ is an interaction *completion delay* (dv_i is the value of the delay, and IDR_i a sequence of expiry operations).
- $R_{\text{D}} = \langle rdv_i, DR_i\rangle$ is a scenario *completion delay*.

A scenario is formally interpreted as a possible set of *timed event sequences* $(\sigma,\tau) = (\sigma_1,\tau_1),(\sigma_2,\tau_2),\ldots(\sigma_n,\tau_n)$, where each σ_i is an operation and τ_i, the instant when it occurs according to an abstract global clock. Each operation has applicability conditions and can occur only if they hold. The applicability conditions of the first operation in a scenario corresponds to the scenario *precondition*. Other operation applicability conditions are obtained from the normal processing of operations that precede them.

In Ref. [6], scenarios are formalized as *scenario trees*, with nodes to represent states, and events to represent specific stimuli that may change the state of the system or trigger other events. The difference with our approach is that we do not rely on the use of unique state names, but on the use of *conditions* which infer states. As discussed in Section 6,

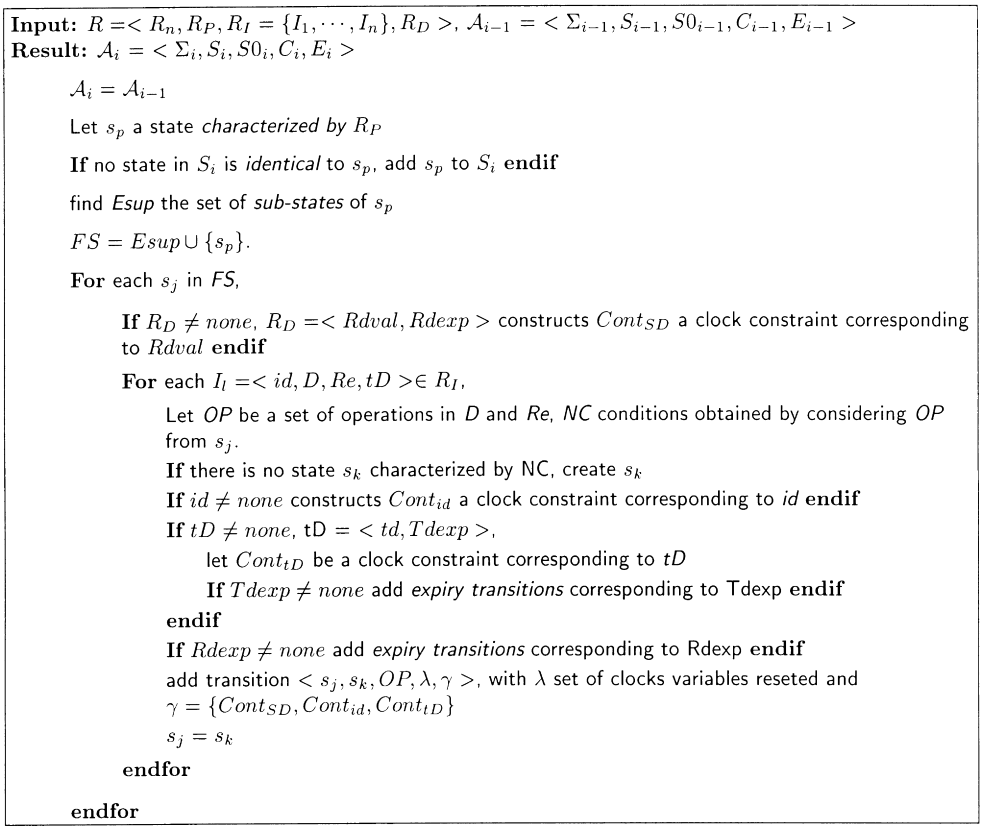

Fig. 5. Scenario composition algorithm.

conditions give us more flexibility when comparing and merging scenarios than state names, as there are no formal means to compare them. Another difference is the addition of timing constraints.

6. Timed automata generation from scenarios

We have developed a specification generator based on an algorithm that incrementally generates timed automata [23] from scenarios. The algorithm allows either the integration of scenarios to an existing timed automata or the construction of new ones. The algorithm guarantees that for each scenario used as input to produce timed automata, there exists a partial run (or partial trace) that corresponds to it. The proof is given in Ref. [8]. This section summarizes this algorithm which is fully presented in Ref. [24].

6.1. Timed automata

A timed automaton is defined as a *timed transition table* $\langle \Sigma, S, S_{ini}, C, E \rangle$ where Σ is a finite alphabet, S is a finite set of states, $S_{ini} \subseteq S$ is a set of start states, C is a finite set of clock variables and $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ is a set of transition.

A transition from state s to s' on the input symbol a is represented as a 5-tuple $\langle s, s', a, \lambda, \gamma \rangle$. $\lambda \subseteq C$ is a set of clock variables reset with the transitions, and γ is a set of clock constraints expressed using clock variables that must be satisfied in C. The clock variable values are set according to a global abstract clock, and hold at each moment the time elapsed since its last reset. The theory of Timed Automata uses a *dense-time* model in which the time domain is a set of positive real values.

A word (σ,τ) recognized by a timed automaton A consists of an event sequence $\sigma = \sigma_1, \ldots, \sigma_n$ and a temporal sequence $\tau = \tau_1, \ldots, \tau_n$, such that σ_i is consumed at the moment of τ_i.

A run $r(\bar{s}, \bar{v})$ of a timed transition table over a timed word (σ,τ) is defined as an infinite sequence $r : \langle s_0, v_0 \rangle \overset{(\sigma_1,\tau_1)}{\rightarrow} \langle s_1, v_1 \rangle \overset{(\sigma_2,\tau_2)}{\rightarrow} \langle s_2, v_2 \rangle \overset{(\sigma_3,\tau_3)}{\rightarrow} \ldots$, with $s_i \in S$ and $v_i \in [C \rightarrow R]$, for all $i \geq 0$, satisfying the following requirements:

- $s_0 \in S_{ini}$ and $v_0(x) = 0$ for all $x \in C$ and
- for all $i > 0$, there is an edge in E of the form $\langle s_{i-1}, s_i, \sigma_i, \lambda_i, \gamma_i \rangle$ such that $(v_{i-1} + \tau_i - \tau_{i-1})$ satisfies γ_i and v_i equals $[\lambda_i \rightarrow 0](v_{i-1} + \tau_i - \tau_{i-1})$.

We define a partial run $\check{r}$ of a run r, as a finite sequence $\check{r}{:}\langle s_i, v_i \rangle \overset{(\sigma_{i+1},\tau_{i+1})}{\rightarrow} \ldots \langle s_{i+n}, v_{i+n} \rangle$ included in r.

Timed automata are used as a formal method target because of the direct relationship between formal interpretation of scenarios and partial runs.

6.2. A timed-automata synthesis algorithm

The general principle of timed-automata generation is to take each input scenario and produce a timed automaton so that there exists a partial run. When a scenario is considered, such a partial run is sought, and if it does not exist, the automaton is augmented with states and transitions to include it. Correspondence is made between each scenario and parts of an automaton. More precisely, between conditions (in scenarios) and states (in the automaton), and between interactions and transitions. Clock variables and constraints are added to transitions according to delays and timeouts in scenarios.

Timed-automata construction is based on concepts similar to those of state-based planning systems such as STRIPS [25].

1. A state is defined by *characteristic conditions* that hold. Conditions thus provide a finer definition of states than abstract names, and allow us to formally compare states as follows:
   - Two states are *identical* if they have the same *characteristic conditions*.
   - A state s_b is a *sub-state* of a state s_a (its *sup-state*), if its characteristic conditions include those of s_a.
2. Each operation execution results are expressed by means of *added-conditions* and *withdrawn-conditions*. For an operation, *added-conditions* are a set of conditions that become true after its execution, while its *withdrawn-conditions* are a set of conditions that are no longer true after its execution.

The following rules must be respected in each automaton generated.

Rule 1. All transitions possible from a state must be possible from all its sub-states.

Rule 2. A non-empty sequence of transitions must exist between any state and each of its sub-states.

The composition algorithm is shown in Fig. 5. Given a set of scenarios, it generates a set of automata that execute in parallel. Each automaton generated describes the behavior of a system component, which appears in some scenarios. A scenario composition begins by selecting the automaton that corresponds to it. This automaton is then updated by addition of states and transitions to obtain a *partial run* over the scenario in it. There may be several partial runs corresponding to a single scenario. Each of these partial runs begin at a state where the *preconditions* hold (first states of the partial runs). Transitions corresponding to the scenario interactions are added from each of these states according to rule 1. The first state of a scenario partial run includes a state *characterized* by its *pre-conditions* (created if not existent), and all its *sub-states*. The interactions are added sequentially from each of the first states. After adding an interaction, a state is obtained and used as a beginning state for the next interaction. For each interaction, a transition is created to allow the execution of its *stimuli*/*reaction*. Transitions may also be created for a *completion delay* expiry.

Delays in scenarios cause the addition of clock constraints to transitions. These clock constraints are constructed with clock variables used to count the time elapsed within the states. There is at the most one clock variable for each state. It is initialized in all of the transitions that go to this state.

There is the following correspondence between delays and clock constraints. For *triggering delays*, c being a *clock variable*, and d the timed amount of the delay, a *maximal delay* produces a clock constraint $c < d$, a *minimal delay* a clock constraint $c > d$, and an *exact delay* a clock constraint $c = d$.

A *completion delay* corresponds to a clock constraint $c < d$. This clock constraint is added to all transitions generated from a scenario, and when there are *expiry operations*, a transition with a clock constraint $c = d$ is created to execute these operations.

Transition arrival states are obtained using operation effects. These begin with the *characteristic conditions* of the state from which the interaction is added.

For each operation, a new set of conditions is obtained from the original one, by removing conditions that appear in the operation *withdrawn-conditions*, and by adding conditions in its *added-conditions*. The *characteristic conditions* of the arrival state of the transition are the final set of conditions obtained after considering all the operations.

When a new state is inserted in the automaton, certain conditions are determined, and there is no state *characterized* by them in it.

According to rule 1, when new states are introduced in the automaton, all transitions that are possible from a state are possible from all its *sub-states*. Transitions with the same operations are added from every new state for all the transitions going from its *super-states*. The addition of these transitions may create new states. In rule 2 *Synthetic* transitions may also be added between states and their *sub-states*. They may also be added to transitions when interaction *stimuli* include some conditions not verified in the state from which they are added.

The timed-automata generation algorithm deals with overlapping scenarios. In fact, by using characteristic conditions, we can compare states derived from different scenarios and reuse them. The only necessity is the use of same identifiers for conditions that are eased by the application domain definition. The algorithm may also produce several independent timed-automata, when the provided scenarios are unrelated.

States, transitions, clock variables, clock constraints and events obtained from a scenario, are labeled with its *number id*, providing a traceability between scenarios and timed automata generated. Using this property, we defined an algorithm to efficiently undo a scenario composition, by removing all the behaviors added to its specification. This

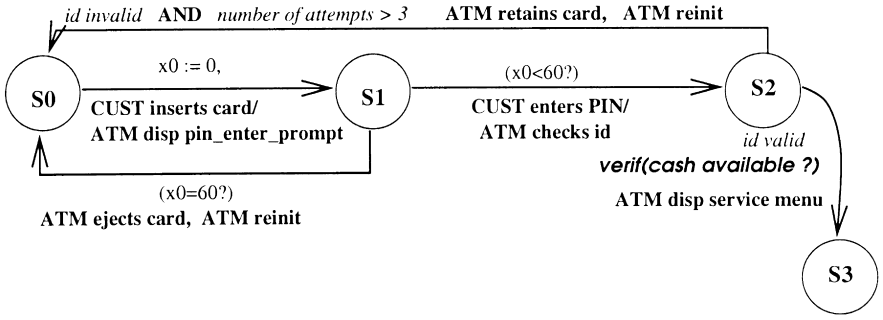

Fig. 6. Automaton generated from scenario 1a.

allows modification of a scenario without having to worry about all the other composed scenarios.

There is a possibility of an exponential explosion of the number of states when similar scenarios are composed. These problems are inherent to the use of automata for interactive systems behavior description [26]. It is not possible to avoid this exponential creation of states because each state represents a possible situation in the system according to the scenarios used, and to the description of operations provided. The algorithm shown in Fig. 5, however, is improved [27] by the use of *grouped states*, like Harel's Statecharts [12].

6.3. A scenario composition example

The following example uses the domain description in Fig. 2. Fig. 6 shows the automaton generated after composition of scenario 1a, shown in Fig. 3, with an empty specification.

S_0, the first state generated, is *characterized* by the scenario *precondition*, while the other states are determined by considering the effects of operations. As an example, state S_1 is *characterized* by the condition *display is pin enter prompt*, obtained by considering the effects of the first interaction operations. The *completion delay* of the scenario's second interaction produces clock constraints on transitions from S_1 to S_0, and S_1 to S_2. x_0, the clock variable used, is reset for transitions to state S_1.

The addition of a second scenario shown in Fig. 7, to the automaton in Fig. 6, produces the automaton in Fig. 8. Scenario 2a overlaps with the scenario 1a, and defines an alternative behavior to it.

The *precondition* of scenario 2a is the *characteristic* condition of S_1, and S_2 and obtained after the first interaction. Therefore, no new state is created until the second interaction, which creates state S_3,and a transition from S_2 to it. A *synthetic condition* verif(cash is available) is added in the transition from S_2 to S_3, because the condition *cash is available* must be satisfied in the second interaction.

```
2a WHEN display is pin enter prompt
   IF CUSTOMER enters PIN THEN ATM checks id
      TRANSITION DELAY 60 sec ON EXPIRY ATM ejects card, ATM reinits
   IF id is valid AND cash is available THEN ATM displays service menu
```

Fig. 7. Scenario 2a.

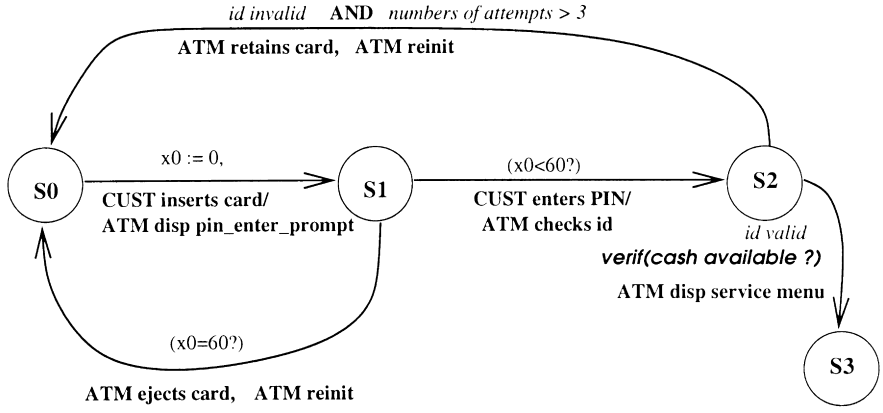

Fig. 8. Automaton obtained after composition of scenario 2a.

```
1b WHEN ATM display is card insert prompt
   IF  CUSTOMER inserts card THEN ATM displays pin enter prompt
   IF CUSTOMER enters pin THEN ATM checks id
   IF id is valid  THEN ATM displays service menu
   IF CUSTOMER selects cash withdrawal THEN ATM displays amount menu.
```

```
2b WHEN ATM display is service menu
   IF CUSTOMER selects cash withdrawal THEN ATM ask amount
   IF CUSTOMER enters amount THEN ATM checks amount
   IF amount is valid THEN  ATM gives cash, ATM displays service menu.
```

Fig. 9. Scenarios with operational inconsistency.

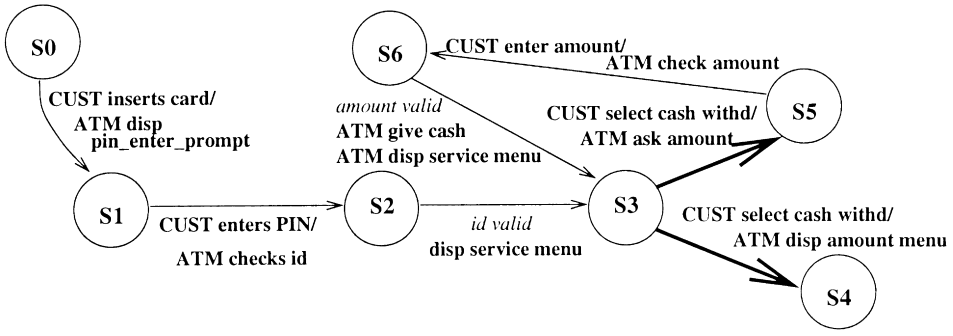

Fig. 10. Automaton obtained by composing scenarios 1b and 2b.

The states S_0, S_1, S_2 and the transitions between S_1 and S_2, and S_1 and S_0 are labeled with the numbers 1a and 2a. The state S_3 and the transition from S_2 to S_3 are labeled with the number 2a. The transitions from S_1 to S_0, and from S_2 to S_0 are labeled with 1a. Removing a scenario from a specification simply removes its number from all the labels, and when an element has no more labels, it is removed from the specification.

7. Scenario coherence verification

We distinguish three kinds of inconsistencies in a scenario: operational inconsistencies, temporal inconsistencies and inconsistencies against invariants. As observed in Ref. [20], many inconsistencies can be detected, but its correction cannot be fully automated because this implies further elicitation. Thus, we detect inconsistencies, but correction is left to analysts and users.

Detection of inconsistencies in a scenario uses the analysis of specifications resulting from their composition. The traceability relationship between specifications and scenarios, obtained by labeling specification elements with scenario numbers, allows us to point out faulty scenarios to users for modification.

7.1. Operational inconsistencies

Operations of two interactions in different scenarios may be contradictory if in a same *situation*, with the same temporal constraints, they define different system *responses* to the same *stimuli*. As in Refs. [6,19], these operational inconsistencies produce *non-deterministic* transitions that go from the same state, with the same stimuli. An algorithm, which checks all the transitions going from each state, allows us to find and report these *non deterministic* transitions.

The scenarios shown in Fig. 9 include an operational inconsistency. In scenario 1b, the operation *select cash withdrawal*, in the third interaction, causes the ATM to *display amount menu*, while in the scenario 2b, the same operation causes the ATM to *ask amount*.

Fig. 10 shows the automaton obtained when the scenario 1b is composed with the scenario 2b. The inconsistency reported produces two transitions from S_3, to S_4 and to S_5, with the same stimulus but with different reactions. Fig. 11 shows the inconsistency detected by the prototype tool.

Inconsistent scenarios are shown to users and analysts who may take appropriate actions for correction. The latter may be a modification of the existing scenarios. As an example, the inconsistency between scenarios 1b and 2b can be corrected by choosing one of the operations, either *ATM ask amount* or *ATM display amount menu* for both scenarios.

A *non-deterministic* behavior may, however, be wanted in requirements. As an example, in a game, it may be correct to randomly choose between several actions for the same stimuli. In this case, the *non-determinism* should be left in the specification.

```
OPERATIONAL INCONSISTENCY in state  s3
   Operations: ATM DISPLAYS AMOUNT MENU and  ATM ASKS AMOUNT

   Transitions:
     (s3,s5,CUST SELECTS CASH WITHDRAWAL & ATM ASKS AMOUNT) and
     (s3,s4,CUST SELECTS CASH WITHDRAWAL & ATM DISPLAYS AMOUNT MENU)

Obtained from  Scenarios: [2B] and [1B]
```

Fig. 11. Analysis result for an operational inconsistency.

```
1c  WHEN ATM display is card insert prompt
   AFTER 10 sec IF CUSTOMER inserts card THEN ATM displays pin enter prompt
   IF CUSTOMER enters pin THEN ATM checks id
   IF id is valid THEN  ATM displays service menu
  DELAY 20 sec.
```

```
2c WHEN display is card insert prompt
   BEFORE 5 sec IF CUSTOMER inserts card THEN ATM displays pin enter prompt
   AFTER 15 sec IF CUSTOMER enters pin THEN ATM checks id
   TRANSITION DELAY 35 sec
   IF id is invalid  THEN ATM reinits.
```

Fig. 12. Scenarios with temporal inconsistencies.

7.2. Temporal inconsistencies

Delays in different scenarios may be contradictory if they exclude each other, but their interactions have the same operations and can be performed in the same *situation*. The automaton obtained when scenarios with temporal inconsistencies are composed includes clock constraints, which cannot be satisfied. We have developed algorithms to check these inconsistencies.

The scenarios in Fig. 12 include two temporal inconsistencies. The two scenarios have the same *precondition*, but while the first interaction in scenario 1c needs a 10 s pause, the scenario in 2c should have started before a delay of 5 s in the first interaction.

There is another less apparent temporal inconsistency between the two scenarios. Scenario 1c must be completed in 20 s, but the conjunction with scenario 2c makes this requirement impossible. In fact, 10 s must pass before the first interaction in scenario 1c, and 15 s must pass after this interaction and before the second interaction in scenario 2c. The combination of these two delays is such that the third interaction of scenario 1c can begin after a minimal delay of 25 s, from the beginning of the scenario.

An analysis of the automaton generated by scenarios 1c and 2c, shown in Fig. 13, allows their temporal inconsistencies to be found (Fig. 14).

The first inconsistency produces the clock constraint "$x_0 < 5$ and $x_0 > 10$" on the transition from S_0 to S_1. This constraint cannot be satisfied. We automatically find this kind of unsatisfiable constraint by checking all of the transitions' temporal constraints. It is important to observe that verification of a constraint such as "$x_0 < 5$ and $x_0 > 10$" is much simpler than the *NP-complete* problem of *satisfiability*. As a matter of fact, all terms of a formula checked here are made of the same variable, include a comparator and a constant. The verification reduces to two comparisons. The kind of inconsistency described here always produces a kind of constraint with the same clock variable, because the composition algorithm uses the same clock variable for all the constraints in transitions going from the same state.

The temporal inconsistency, introduced by scenario 1c's completion delay is detected by an algorithm which determines in each state the relationship between the automaton clock variables and their minimal values. We do this by using the transition's clock constraints and initializations. This may involve an exploration of all the paths of the automaton starting at its initial state. This exploration is optimized by sorting the transitions going from each state,

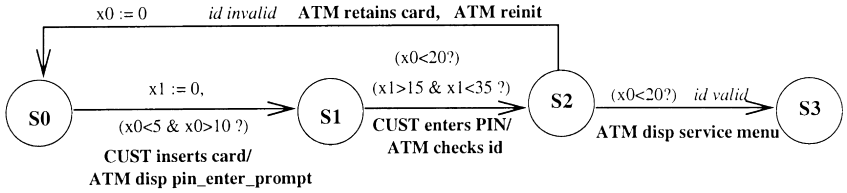

Fig. 13. Automaton obtained from scenarios 1c and 2c.

```
TEMPORAL INCONSISTENCIES
 Between scenarios: [1C,2C]
  transition:
  (s0,s1,CUST INSERTS CARD & ATM DISPLAYS PIN_ENTER_PROMPT,c1:=0,c0>10 & c0<5)
   ''c0 > 10 and c0 < 5'' is unsatisfiable.

 In scenario: [1C] ''delay 20 sec'' is impossible in transition:
   (s2,s3,ID is  VALID & ATM DISPLAYS SERVICE_MENU,,c0 < 20)
```

Fig. 14. Analysis result for temporal inconsistencies.

```
1I WHEN numbers of attempts is greater than 3 INHIBIT ATM displays service menu
```

Fig. 15. An invariant.

considering initializations and smaller constraints first, looking only for the clocks minimal values. The application of this algorithm to the automaton in Fig. 13, allows us to find that in state S_2, x_0 has 25 as a minimal value, and consequently the clock constraint "$x_0 < 20$" in the transition from S_2 to S_3 cannot be respected.

7.3. Invariants

Invariants are requirements defined to inhibit execution of some operations in particular situations.

Fig. 15 shows an example of an invariant, which prevents the operation *display service menu* by the ATM; this happens anytime the condition *numbers of attempts is greater than* 3 is satisfied. Invariants may be verified when scenarios are composed, or against an already constructed specification.

An invariant verification during composition proceeds as follows: Whenever a transition is constructed from a state, we check if the conditions of the invariant are included in its *characteristic conditions*, and then if the transition includes some of the operations inhibited. Invariants not respected are reported immediately upon detection.

Verification of an invariant against a specification looks for the operations that are inhibited in all transitions going from all states in which *characteristic conditions* include the conditions of the invariant. Transitions that do not respect the invariant are reported for a possible modification, with the number of scenarios from which they are derived.

As an example, verification of the invariant 1I against the specification in Fig. 8 produces the analysis result in Fig. 16. The invariant is not respected because the *number of attempts* may have any value in the transition from S_2 to S_3. A correction can be provided by adding the condition *number of attempts* > 3, as an additional stimulus of the second interaction of the scenario 2a.

8. Specification completion

A complete specification may be defined as one that contains all the facts about the described system, even those that are not defined in the user requirements [28]. Completeness cannot be ensured in a specification obtained from users' scenarios because it includes only the requirements given. However, we provide some guidance in specification completion.

Some of the incompleteness in specifications result from missing operations to link scenarios. This incompleteness produces *synthetic transitions* between states and sub-states, and *synthetic conditions* in transitions. *Synthetic* transitions and conditions may be removed by adding operations or interactions to scenarios. As an example, the automaton in Fig. 8 obtained from scenarios 1a and 2a includes a *synthetic condition*. This *synthetic condition* comes from the second interaction of scenario 2a's, because after the first interaction there is no way to know if one of its stimuli, the condition *cash is available*, is verified in state S_2. The specification may be complete by replacing scenario 2a with scenario 2a$'$, shown in Fig. 17.

Incompleteness that can be detected in specifications are classified as temporal constraint incompleteness and behavior incompleteness.

There exists a temporal constraint incompleteness when from a given state some time ranges do not correspond to any transition. This kind of incompleteness may result on the system being indefinitely blocked in a state. As an example in Fig. 13, from state S_2, when *id is valid*, no behavior is

```
INVARIANT VIOLATION: invariant 1I
 Transitions:
   (s2,s3,id is valid AND verif(cash is available) & ATM DISPLAYS SERVICE MENU)
 Obtained from  Scenarios: [2A]
```

Fig. 16. Analysis result for an invariant.

```
2a' WHEN display is pin enter prompt
   IF CUSTOMER enters PIN THEN ATM checks id
      TRANSITION DELAY 60 sec ON EXPIRY ATM ejects card, ATM reinits
   IF id is valid  THEN ATM checks cash availability
   IF cash is available THEN ATM displays service menu
```

Fig. 17. Scenario 2a′. Operation *check cash availability* must be defined. One of its possible effects must be the assertion of the condition *cash is available*.

defined if x_0 is greater or equal to 20. The system may, therefore, be indefinitely blocked in S_2 if no behavior is defined in this case. Incompleteness is found and reported by checking all the clock constraints from the transitions going from state to state.

Behavior incompleteness includes conditions that are not taken into account and unspecified responses to possible stimuli. Conditions are not taken into account, for instance, when we know that they may occur from a state, but no transition considers them. As an example, the automaton in Fig. 10, we know that in state S_2, the Customer's *id* may be *valid* or *invalid* according to the operation *check id* definition. There exists an incompleteness in this state, because only the condition *id is valid* produces a transition from it. Such a kind of incompleteness is reported by the prototype tool and may be corrected by providing other scenarios.

Unspecified responses to stimuli originate from missing interactions that may, e.g. be errors at the system interface. As we know from the set of *stimuli* used, some of the *error transitions* may be added by asking *What/If* questions. For the timed-automaton shown in Fig. 10, such a question is:

> What happens if the CUSTOMER selects cash withdrawal in situation "display is pin enter prompt", holding in state S_1?

The operation may be prevented by the system interface. But scenarios are supposed to be used for a system's preliminary design, and this kind of question may aid precisely the designing of the interface, or may call for the addition of missing scenarios.

9. Conclusion

We have presented an environment for scenario-based requirements engineering. This environment aims in assisting users and analysts in scenario acquisition, and production of complete and valid specifications from them. We seek the user's greater involvement in the requirement engineering process by using *semi-formal* languages for scenario representation and prototype simulation.

We also integrated time to a scenario because of its importance in real-time systems. The paper is illustrated with ATM scenarios. Although these examples are sufficient to demonstrate our algorithms, more industrial examples are needed to assess the applicability and usefulness of the approach. We used the prototype tool for telephone service definition and verification of feature interaction problems [29,30]. Telephone services are well described using scenarios and feature interaction problems that occur when several services are combined. An experiment was conducted in order to answer the following questions: How difficult is the modeling of communication services with scenarios? Are we able to detect the well-known feature interactions? And finally, how efficient is the composition algorithm? The experiment was carried out by a student with little knowledge of the system. All the scenarios have been obtained from informal descriptions. The Appendix describes two typical feature interactions. For more examples see Ref. [31].

Future work concerns the extension of the scenario language that allows the use of explicit composition operators and modularity. We have to adapt accordingly, all the algorithms that are used for inconsistency detection in the prototype tool. Future work will be undertaken with the financial support of FranceTelecom.

Appendix

Note to the reader, this appendix is not necessary for understanding this paper.

Fig. 18 shows the system architecture assumed in this appendix. The application involves a system CONTROLLER and three users. Example 1 shows the application domain description and scenarios for Three-Way Conference and Call-Waiting services. Scenarios are given in both textual and graphical forms. The tool REST is dedicated to service creation and validation from the first step. We have specified 15 telephone services and detected all corresponding feature interactions. More details are given in Ref. [31].

A1. Example 1

Application domain description

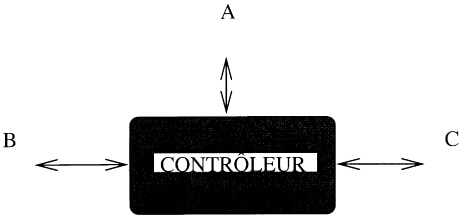

Fig. 18. An abstract view of the telephone system.

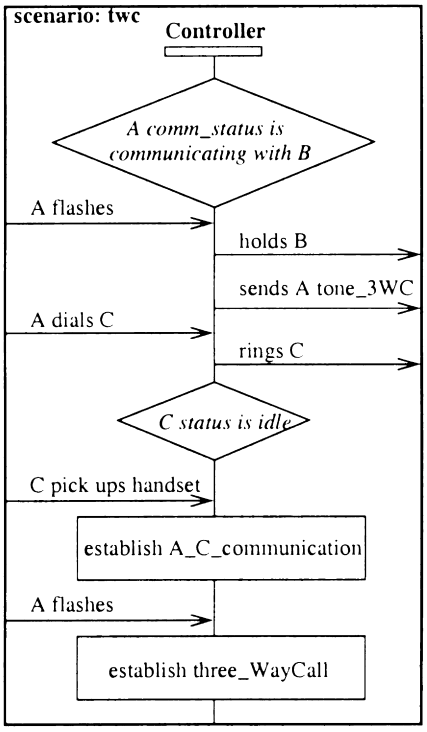

Fig. 19. Three-way conference scenario.

System Component: CONTROLLER

operation: send busy_tone
operation: send tone
operation: send tone_3WC
operation: check USER_B status

postconditions: USER_B status is idle OR USER_B status is busy

operation: ring USER_B
operation: establish A_B_comm

postconditions: USER_A communication_status is communicating_with_B

operation: establish A_C_comm
operation: establish three_WayCall
operation: stop_A
withconditions: ends USER_A dial USER_B AND

ends CONTROLLER establish A_B_comm

operation: hold USER_B

postconditions: USER_B communication_status is holding

operation: hold USER_C

postconditions: USER_C communication_status is holding

System Component: USER_A

attribute: handset values: down, up
attribute: communication_status

values: communicating_with_B, communicating_-with_C

operation: pick_up handset

postconditions: USER_A handset is up

operation: dial USER_B
operation: dial USER_C
operation: flash

System Component: USER_B

attribute: status values: busy, idle
attribute: communication_status

values: communicating_with_A, communicating_-with_C

attribute: handset values: down, up
operation: pick_up handset

System Component: USER_C

attribute: status values: busy, idle
attribute: communication_status

values: communicating_with_A, communicating_-with_C, holding

Three-Way Conference Service Scenario (Fig. 19)

1twc WHEN USER_A communication_status is communicating_with_B
IF USER_A flashes THEN CONTROLLER holds USER_B AND

CONTROLLER sends USER_A tone_3WC

IF USER_A dials USER_C THEN CONTROLLER rings USER_C
IF USER_C status is idle AND USER_C pick_ups handset

THEN CONTROLLER establishs A_C_comm

IF USER_A flashes THEN CONTROLLER establishes three_WayCall.

Call-Waiting Service Scenarios (Fig. 20)

1cw WHEN USER_A communication_status is communicating_with_B

IF USER_C status is idle AND USER_C pick_ups handset

THEN CONTROLLER sends USER_C tone

IF USER_C dials USER_A THEN CONTROLLER beeps USER_A

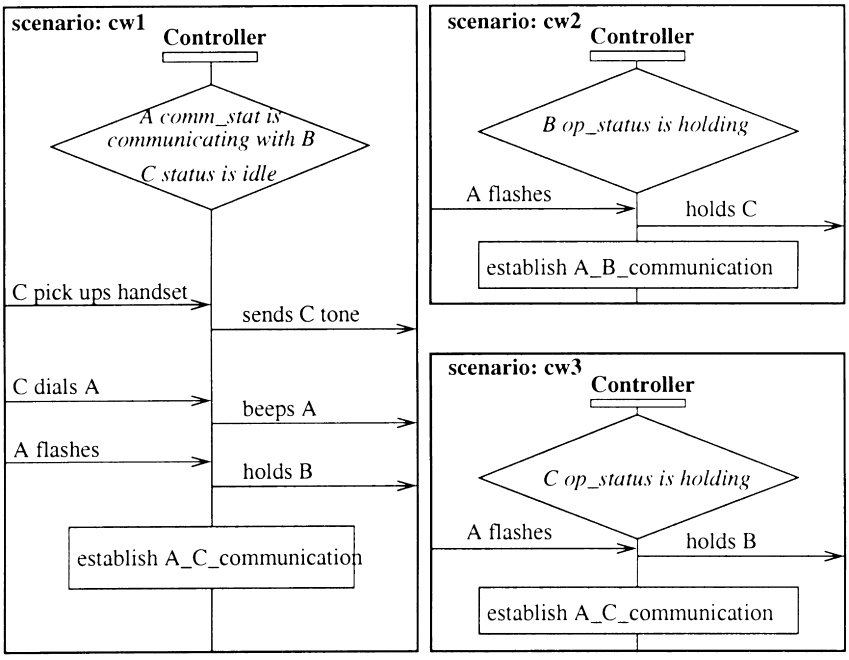

Fig. 20. Call-Waiting scenarios.

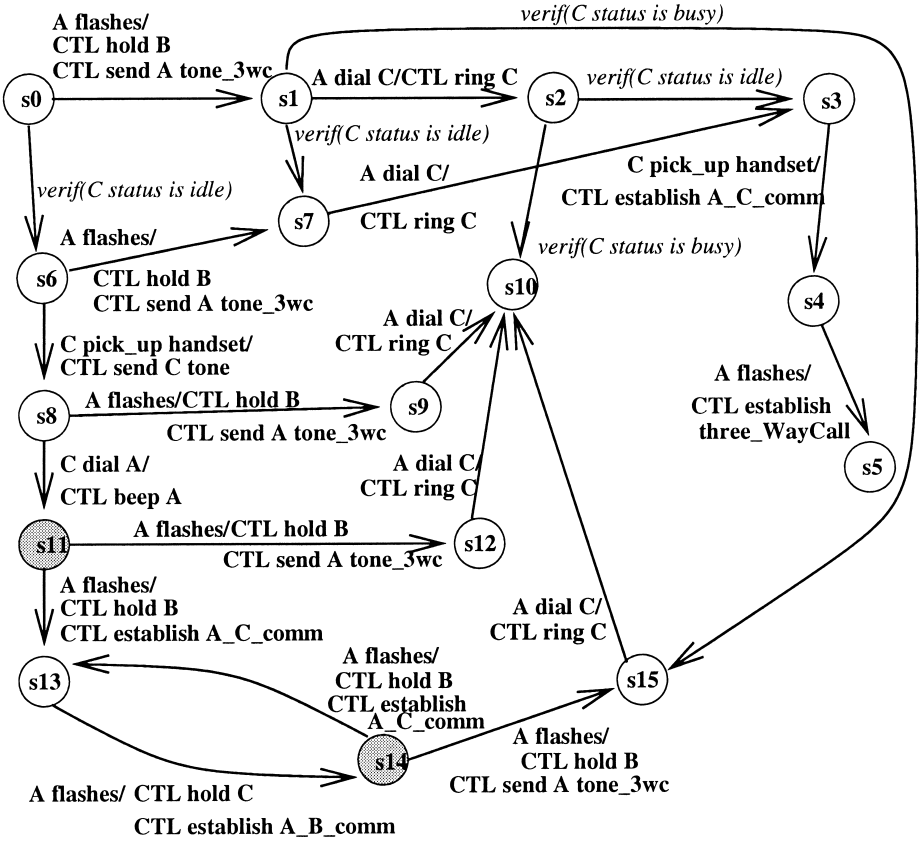

Fig. 21. Automata obtained by composition of three-way conference and call-waiting.

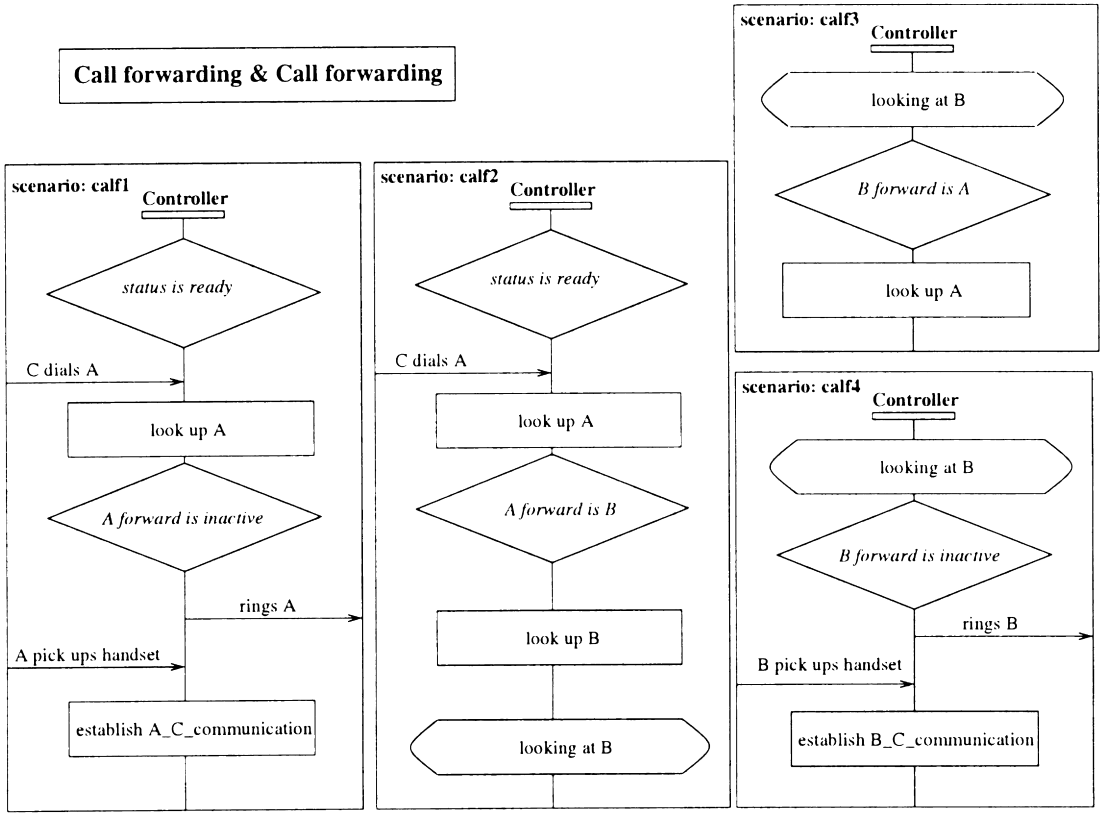

Fig. 22. Call-forwarding versus call-forwarding scenarios.

IF USER_A flashes THEN CONTROLLER holds USER_B AND

CONTROLLER establishs A_C_comm

2cw WHEN USER_B communication_status is holding

IF USER_A flashes THEN CONTROLLER holds USER_C AND
CONTROLLER establishs A_B_comm

3cw WHEN USER_C communication_status is holding

IF USER_A flashes THEN CONTROLLER holds USER_B AND
CONTROLLER establishs A_C_comm

A2. Call-Waiting versus three-way conference

We have already shown the scenarios for these services. Call-Waiting is described with scenarios 1cw, 2cw and 3cw, and Three-Way Conference is described with scenario 1twc. Fig. 21 shows the automaton obtained by the composition of

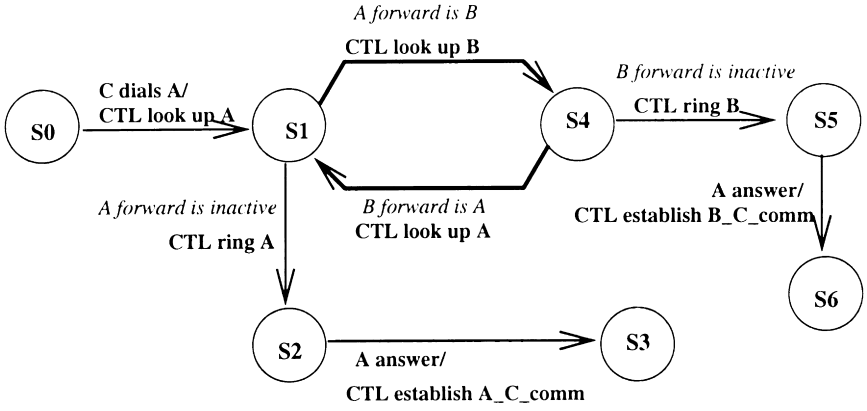

Fig. 23. Automaton obtained by composition of scenarios calf1, calf2, calf3 and calf4.

the two service scenarios using the application domain of example 1. It is already known that the combination of these two services produces a feature interaction problem [32]. This interaction is detected without tool because the resulting automaton includes non-deterministic transitions. Indeed in states s11 and s14 of the automaton in Fig. 21, the same stimulus (A flashes) produces two different system reactions: hold B followed by establish A_C_comm and hold B followed by send A tone_3wc. The first reaction corresponds to Call-Waiting, while the second reaction is defined by a Three-Way Conference. Several other interactions produce non-deterministic transitions and are detected in the same way. These interactions include Credit-Card Calling versus Voice-Mail Service and Call-Waiting versus Voice Mail Service.

A3. Call forwarding versus call forwarding

Call Forwarding allows a user to forward all his incoming calls to another user. When A forwards its calls to B, the system redirects any call for A to B. This feature may interact with itself and produce an infinite loop [32]. The problem occurs when the call forwarding feature is being used repetitively by a chain of users. Suppose that user A decides to forward his calls to user B's location and user B decides to forward his calls to user A's location. If a third user attempts do dial either phone number, an infinite loop will be generated. Scenarios calf1, calf2, calf3 and calf4 in Fig. 22 describe an example where user A may forward his calls to B, B may forward his calls to A, and C calls A. In scenario calf1, C calls A. The system verifies if A forward is activated (look up A). If A forward is inactive the system then rings A and a communication is established when he picks up. If A forward is B (scenario calf2), the system must look up B. We use a situation descriptor here to show that this scenario may be followed by either scenario calf3 or calf4. Scenario calf3 describes the case where B forward is inactive. The system then rings B and establishes a communication between B and C when B picks up. In scenario calf4, as B is forwarded to A the system looks up A. We do not show the application domain description used for this example because of space restriction. Fig. 23 shows the automaton resulting from the composition of scenarios calf1, calf2, calf3 and calf4 in our environment. From the experimentation we get the following results: scenarios are appropriate for the description of communication services. They are easy to use and understand for most designers. Our environment shows that it is possible to encapsulate formal methods within a general tool.

References

[1] B.W. Boehm, Software Engineering, IEEE Transaction on Computers C-25 (12) (1976) 1226–1241.

[2] K.M. Benner, M.S. Feather, W.L. Johnson, L.A. Zorman, Utilizing scenarios in the software development process, in: N. Prakash, C. Rolland, B. Pernici (Eds.), Information System Development Process, Elsevier, North-Holland, 1993, pp. 117–134.

[3] D. Amyot, L. Logrippo, R.J.A. Buhr, Spécification et conception de systèmes communicants: une approche rigoureuse basèe sur des scénarios d'usage. in: CFIP97, Liège, Belgique, September 1997.

[4] M. Glinz, An integrated formal model of scenarios based on statecharts, in: Software Engineering—ESEC'95, Proceedings of the 5th European Software Engineering Conference, Springler LNCS 989,1995, pp. 254–271.

[5] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modelling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.

[6] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen, Formal approach to scenario analysis, IEEE Software March (1994) 33–41.

[7] K. Koskimies, E. Mäkinen, Automatic synthesis of state machines from trace diagrams, Software-Practice and Experience 24 (7) (1994) 643–658.

[8] S.S. Somé, Dérivation de Spécification à partir de Scénarios d'Interaction, PhD thesis, Université de Montréal, 1997.

[9] H. BenAbdallah, S. Leue, Syntactic detection of process divergence and non-local choice inmessage sequence charts, TACAS (1997) 259–274.

[10] ITU, Recommendation Z.120: Message Sequence Chart (MSC), ITU, Geneva, 1996.

[11] P.B. Ladkin, S. Leue, Interpreting message flow graphs, Formal Aspects of Computing 7 (5) (1995) 473–509.

[12] D. Harel, STATECHARTS: A visual formalism for complex systems, Science of Computer Programming 8 (1987) 231–274.

[13] R.J.A. Buhr, R.S. Casselman, Use Case Maps for Object-Oriented System, Prentice Hall, Englewood Cliffs, NJ, 1995 302 pp.

[14] A.W. Biermann, R. Krishnaswamy, Constructing programs from example computations, IEEE Trans. Software Engineering SE-2 (9) (1976) 141–153.

[15] J. Desharnais, R. Khedri, M. Frappier, A. Mili, Integration of sequential scenarios, Lecture Notes in Computer Science 1301 (1997) 310.

[16] R. Alur, G. Holzmanin, D. Peled, An analyzer for message sequence charts, Software: Concepts and Tools 17 (1996) 70–77 also appeared in TACAS'96, Tools and Algorithms for the Construction and Analysis of Systems, Passau, Germany, LNCS 1055, Springer, Berlin, 1996, pp. 35–48.

[17] H. BenAbdallah, S. Leue, Proceedings of the Tenth Conference on Formal Description Techniques FORTE/PSTV'97, Osaka, Japan, Timing constraints in message sequence chart specification, Chapman and Hall, London, 1997.

[18] B. Regnell, M. Andersson, J. Bergstrand, A hierarchical use case model with graphical representation, in: Proceedings of ECBS'96, IEEE Second International Symposium and Workshop on Engineering of Computer-Based Systems, IEEE, March 1996.

[19] M.P.E. Heimdahl, N.G. Leveson, Completeness and consistency analysis of state-based requirements, in: Proceedings of the 17th International Conference on Software Engineering, 1995, pp. 3–14.

[20] S. Easterbrook, B. Nuseibeh, Using viewpoints for inconsistency management, IEE Software Engineering Journal 11 (1) (1996) 31–43.

[21] A. En-Nouaary, R. Dssouli, F. Khendek, Timed scenarios to sdl: specification,implementation and testing of real-systems, in: Submitted for SDL Forum'99, 1999.

[22] ITU-T, Message Sequence Chart (MSC), Recommendation Z.120, 1993.

[23] R. Alur, D.L. Dill, A theory of timed automata, Theoretical Computer Science 126 (2) (1994) 183–235.

[24] S. Somé, R. Dssouli, J. Vaucher, Scenarios to timed automata: building specifications from users requirements, in: Proceedings of the 2nd Asia Pacific Software Engineering Conference (APSEC'95), IEEE, December 1995.

[25] R. Fikes, N. Nilson, STRIPS: A new approach to the application of theorem proving to problem solving, Artificial Intelligence 2 (3/4) (1971) 189–208.

[26] D.L. Parnas, On the use of transition diagrams in the design of a user interface for an interactive computer system, Proceedings of the ACM Annual Conference 00 (1969) 379–385.
[27] S. Somé, R. Dssouli, An enhancement of timed automata generation from timed scenarios using grouped states, Technical Report 1029, DIRO-Université de Montréal, 1996.
[28] V.S. Alagar, D. Kourkopoulos, (In)completeness in specifications, Information and Software Technology 36 (6) (1994) 331–342.
[29] P. Zave, Feature interactions and formal specifications in telecommunications, Computer 26 (8) (1993) 20–30.
[30] R. Dssouli, S. Some, J.W. Guillery, N. Rico, Detection of feature interactions with REST, in: P. Dini, R. Boutaba, L. Logrippo (Eds.), Proceedings of the Feature Interactions in Telecommunications Networks, IOS Press, Amsterdam, 1997.
[31] J.F. Guillery, Detection of telephone service interactions. Technical report, Département IRO, Université de Montréal, 1996.
[32] E.J. Cameron, N.D. Griffeth, Y.J. Lin, M. Nilson, W.K. Schnure, H. Velthuijsen, A feature Interaction Benchmark for IN and beyond, in: L.G. Bouma, H. Velthuijsen (Eds.), Feature Interactions in Telecommunications Systems, IOS Press, Amsterdam, 1994.