Eliciting Requirements and Scenarios using the SCTL-MUS Methodology. The Shuttle System Case Study

José J. Pazos-Arias
Telematics Engineering Dept.
University of Vigo (Spain)
jose@det.uvigo.es

Jorge García-Duque
Telematics Engineering Dept.
University of Vigo (Spain)
jgd@det.uvigo.es

Martín López-Nores
Telematics Engineering Dept.
University of Vigo (Spain)
mlnores@det.uvigo.es

ABSTRACT

The development of complex systems demands methodologies that conveniently support the stakeholders in the creative tasks. In this paper, we present a methodology for the incremental elicitation of requirements and scenarios, driven by the integration checks performed over a state machine that represents the global behavior of the desired system.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Elicitation methods, methodologies*

Keywords

Incremental development, scenarios, requirements, state machines

1. INTRODUCTION

Scenario-oriented solutions are among the most successful attempts to produce quality software requirements [6]. In such approaches, the initial development of a system is based on the knowledge about certain interactions among the agents involved in it. From those known interactions, the stakeholders create a set of scenarios that constitute the starting point for eliciting the requirements of the system.

However, scenarios do not suffice by themselves to reach a requirements specification for a system, because they necessarily represent partial views of its global behavior. It remains an issue to build a global model of the system to perform *integration analysis*, that is, to check that the different parts fit well together and do result in a system that satisfies the needs and expectations of the stakeholders.

In this paper, we present an approach that facilitates the incremental elicitation of scenarios and requirements, beginning with a set of basic scenarios. For this purpose, the scenarios and requirements initially elicited are subject to a process of successive analysis-revision cycles, driven by the verification of desirable integration properties. Each cycle is done over a model of the system that is automatically synthesized from the requirements elicited so far, and produces modifications of those requirements and the scenarios being handled. Since it is not possible to predict the evolutions that the stakeholders may want to apply, these modifications are always presented as suggestions to revise their work. It is precisely in reasoning about whether to accept or reject these suggestions that the approach supports the creative task of building the desired system.

The context for our work lies in the SCTL-MUS methodology, a formal approach to the specification of distributed reactive systems that was introduced in [12]. This methodology supports the incremental development of requirements specifications, taking into account the incomplete and often imprecise knowledge available at the early stages of development. Our vision in this paper is that the same support is suitable for the establishment of a *Scenario-Based Software Engineering* (SBSE) process, with its main steps as described in Section 2. Section 3 illustrates this approach over the Shuttle System case study described in [13], and the results are finally discussed in Section 4.

2. THE APPROACH

Figure 1 depicts our proposal for an SBSE process over SCTL-MUS, which has been inspired by the work presented in [8]. Our goal is to provide adequate support to carry out integration analysis in an incremental specification process.

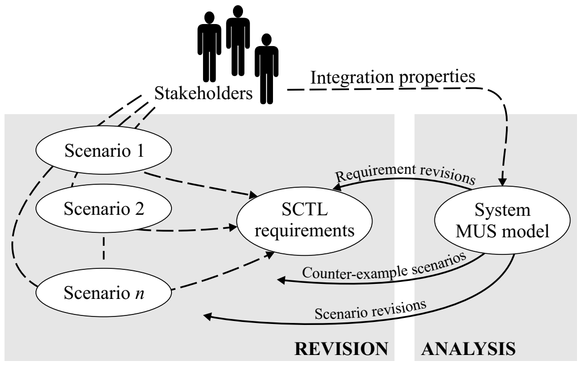


Figure 1: An SBSE process over SCTL-MUS.

From Scenarios to Requirements

The ultimate objective of our development process is to build a model of the global behavior of a system, which allows automating to a great deal its implementation. In this context, scenarios come up as an accessible means for the stakeholders to acquire knowledge about the desired functionality. However, whereas it has been argued that a direct translation is possible from scenarios to models [15], we believe that leaning intermediately on a requirements specification is a more suitable approach. The reason is that scenarios and system models serve different purposes by nature: each scenario captures a particular vision of the global system, involving only a part of the system agents and some of the possible interactions between them. The stakeholders do not conceive the system as a whole, but as a bunch of separate behaviors; thereby, it is not feasible for them to move directly from a set of scenarios to a model that represents the global behavior of a system.

In contrast, scenarios are an intuitive way for the stakeholders to capture requirements, thus enabling a property-oriented way of thinking that is highly suitable for the early stages of software development [16]. Eliciting requirements is an eminently creative task, because an expressive effort is needed to find the requirements that represent more accurately the behavior implied by the scenarios. Again, note that an automated translation between requirements and scenarios is purposeless, because it would amount to rewriting the scenarios into another language, not to eliciting system requirements.

In order to facilitate the elicitation of requirements from the scenarios, it is necessary that the language employed to specify requirements be simple and understandable to the stakeholders, so that it allows them to easily express the interactions implied by the scenarios. For this purpose, we propose using the *Simple and Causal Temporal Logic* (SCTL), whose semantics allows expressing cause-effect relationships between different events in a way that is close to natural language. Such a causal semantics has proved to be beneficial in requirements specification tasks [11].

From Requirements into a System Model

Having a formal specification language, once a set of SCTL requirements have been elicited from the scenarios, we can apply the synthesis algorithm presented in [12] to obtain a state machine for the system. This is expressed in the *Model of Unspecified States* (MUS) formalism, a variation of classical *Labeled Transition Systems* devised to capture the incompleteness of a model obtained from a partial specification. It is over this state machine that integration analysis is performed.

Eliciting Requirements and Scenarios

Due to the characteristic impreciseness of the early stages of development, the requirements elicited from the scenarios may not be representative of a system with the desired functionality. When this happens, the model obtained for the global system fails to pass the integration tests.

We consider three possible sources of incorrectness:

1. The requirements may be incorrect, because the stakeholders can make errors when expressing the scenarios through requirements.
2. The requirements may correspond accurately to the initial scenarios, but be incorrect with respect to others —unknown or not represented in the current stage of development.
3. The considered scenarios may be erroneous.

The first kind of errors can be detected by validating the system MUS model (i.e., by checking whether the scenarios are materialized over the model), but the other kinds demand alternative mechanisms. In response to that, we perform the integration tests according to the paradigm of the *analysis-revision cycle*, which was introduced in [4] as a suitable approach to support evolving requirements specifications.

The main idea behind the analysis-revision approach is to gather *diagnostic information* about the problems found during the analysis, and then use that information to generate revisions of the artifacts provided by the stakeholders. In [5], we applied this paradigm to handle certain evolutions of the requirements, with special emphasis on making it easy for the stakeholders to decide on adopting the suggested revisions. Now, we extend it for the SBSE process, introducing support for the elicitation of scenarios as well. The motivation for doing this is twofold: i) to facilitate even further the decision of whether the requirements revisions should be accepted or rejected, and ii) to facilitate the elicitation of more complex scenarios from a set of basic ones, which is hard work if no convenient assistance is provided.

2.1 Background

The Formalisms

The SCTL-MUS approach is totally formalized. As we already explained, it combines property-oriented and model-oriented formal description techniques: the logic SCTL to express the system's functional requirements, and the graph formalism MUS to model systems for validation, formal verification and implementation purposes.

SCTL statements express under what circumstances during the operation of a system shall a given condition be satisfied. They have the generic form $Premise \bigoplus Consequence$, with $\bigoplus$ ranging over the set of temporal operators $\{\Rightarrow, \Rightarrow \bigcirc, \Rightarrow \odot\}$ and the following semantics:

If $Premise$ is satisfied, then [simultaneously ($\Rightarrow$) | next ($\Rightarrow \bigcirc$) | previously ($\Rightarrow \odot$)] $Consequence$ must be satisfied.

Given a set of requirements expressed in SCTL, the synthesis algorithm of SCTL-MUS generates a MUS graph that adheres to them. MUS labels the transitions between states with three different truth values: *true* (1) for possible transitions, *false* (0) for non-possible ones, and *unspecified* ($\frac{1}{2}$) for those about which there is nothing in the current specification saying whether they must be possible or not.

This notion of *unspecification* was introduced to explicitly deal with the incompleteness inherent to the intermediate stages of an incremental process: a partial specification represents all the systems into which it can evolve by adding new requirements. MUS graphs include an *unspecified* state —not relevant for the contents of this paper and, therefore, not drawn in the figures— that represents all the states that have not been specified so far (see [12] for the details). With this definition, adding new requirements to a specification always results in *losses of unspecification* in the MUS model that implements it (i.e., some *unspecified* events are turned into possible or non-possible ones), which allows making the synthesis an incremental process. This is the main difference with respect to other formalisms intended to support partial specifications, like KPSs [3], MTSs [7] or PLTSs [14].

The Analysis-Revision cycle

The analysis phase of our analysis-revision cycle consists of verifying the system MUS model against a number of desirable properties, specified by the stakeholders as SCTL formulae. This is done by a many-valued model-checker that, leaning on the management of *unspecification* and the causality of SCTL, enables reasoning about evolutions and satisfaction tendencies in the formal verification process.

When the model-checker finds that an integration property is not satisfied in the system MUS model (or that the model cannot lose *unspecification* so as to satisfy the property), it generates diagnostic information that is passed to the revision phase. This phase uses the diagnostic information to automatically generate modifications of the elicited requirements, that would solve the problems with the properties in question. In doing this, the mechanisms introduced in [5] allow presenting evolutions as specializations of the requirements elicited by the stakeholders, making the revisions easy to understand. Proceeding otherwise would produce artificial requirements that, not capturing the expressive efforts of the stakeholders, do not serve to support the incremental elicitation process.

2.2 Eliciting Scenarios

As it was presented in [5], our analysis-revision cycle serves in the SBSE process to generate revisions of the requirements elicited from the initial scenarios. Besides, as part of the analysis phase, the same mechanisms can be used to provide counter-example scenarios illustrating behaviors that violate the integration properties. Those scenarios highlight the changes implied by the revisions of the requirements being considered with respect to the initial situation.

Notwithstanding, the revision process is not complete only with the aforementioned features. To facilitate the elicitation of more complex scenarios from a set of basic ones, it is very helpful to see the modifications of the global model and the requirements reflected as modifications of the current scenarios. Inspired by the work presented in [17], we aim at supporting the incremental elicitation of scenarios by exploiting the management of incompleteness of SCTL-MUS.

Given a set of initial scenarios, our approach generates scenario revisions from the revisions of the requirements. Again, we believe that it is not a suitable approach to invent new scenarios, since they would be meaningless for the stakeholders; on the contrary, we look for specializations of the scenarios they are handling. To accomplish this, we synthesize a new system MUS model implementing the revised requirements, and then check whether all the materializations of the scenarios over the original model are preserved. In case not, we consider two different situations:

1. **All the materializations have been lost.** When the revised requirements do not represent all the behavior implied by the scenario, the model synthesized from those requirements does not contain materializations of it. Nonetheless, the incompleteness of the model may permit evolving the specification so that the scenario is materialized again. The *unspecified* elements of MUS models allow distinguishing two cases:

    - *It is possible to turn* unspecified *events into* true *or* false *ones so that the scenario is materialized again.* In this case, we provide a revision including: i) the part of the scenario that is materialized, and ii) additional revisions of the requirements that imply the necessary changes in the model. These revised requirements can preserve all the knowledge of the others, and so we talk about *refinements* in the traditional sense.
    - *There is no way to lose* unspecification *so that the scenario is materialized again.* This situation is due to a wrong revision of the requirements or, alternatively, to the scenario being erroneous. In this case, we first provide counter-example scenarios obtained from the traces of the revised model. If none of those scenarios is accepted as representative of the desired system, we start looking for requirements revisions that would lead to a model materializing the scenario. Since it is necessary to contradict the behavior that prevented the scenario from materializing, these modifications cannot imply refinements; instead, we have introduced support for *retrenchments* [1], which allow evolving a specification by contradicting part of the knowledge it captures.

2. **Some materializations have been lost, but others are preserved.** In this case, a specialization of the scenario has occurred as a result of adding knowledge to the specification —through the verification of desirable properties. The specialization consists of refining the original scenario to fit the materializations which are preserved. In addition to that, we provide new scenarios illustrating the materializations that have been lost. It is here where the more complex scenarios are elicited from the simpler ones, initially conceived by the stakeholders.

3. CASE STUDY: THE SHUTTLE SYSTEM

Next, we show a run of our approach over the Shuttle System case study proposed in [13]. This is about a railway of interconnected stations, in which shuttles bid for orders to transport passengers between certain stations. The successful completion of an order results in a monetary reward for the corresponding shuttle, but a penalty is incurred in case an order is not served in a given amount of time. New orders are made known to all the shuttles, so that all of them can make an offer. The shuttle with the best (i.e., the lowest) offer will receive the assignment.

From Scenarios to Requirements

We begin with the scenarios shown in Figures 2 and 3, which have been extracted from [13]. These scenarios represent interactions between a generic *Shuttle Agent* —of which there exist n instances in the system—, the *Broker Agent* and the *Banking Agent*.

Scenario 1 shows that the *Shuttle Agent* can make an offer or not when it receives an order, and makes an invoice when it serves an assigned order in time. Scenario 2 shows that the *Shuttle Agent* can be penalized if it does not serve an assigned order in time.

From these scenarios, we assume that the stakeholders have elicited the following requirements for the i-th *Shuttle Agent* (each SCTL statement is given along with a textual description to help understand its meaning):

1. *"After receiving an order, it is possible to make an offer"*:

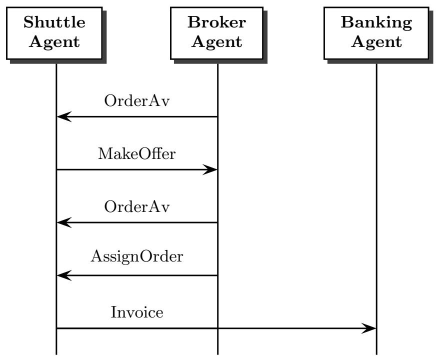


Figure 2: Scenario 1.

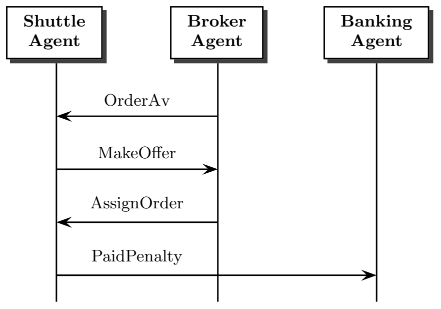


Figure 3: Scenario 2.

$\mathcal{R}1_i \equiv OrderAv \Rightarrow \bigcirc MakeOffer_i$

2. *"When it is possible to make an offer for an order, it is also possible to ignore it"*:

   $\mathcal{R}2_i \equiv MakeOffer_i \Rightarrow IgnoreOrder_i$

3. *"After making an offer, it is possible to receive the assignment of the corresponding order"*:

   $\mathcal{R}3_i \equiv MakeOffer_i \Rightarrow \bigcirc AssignOrder_i$

4. *"Once an order has been served (in time or not) or ignored, it is possible to receive a new order"*:

   $\mathcal{R}4_i \equiv (IgnoreOrder_i \vee Invoice_i \vee PaidPenalty_i) \Rightarrow \bigcirc\, OrderAv$

5. *"If an order has been assigned, then it can be served in time, but it can also provoke a penalty"*:

   $\mathcal{R}5_i \equiv (true \Rightarrow \bigodot AssignOrder_i) \Rightarrow (Invoice_i \wedge PaidPenalty_i)$

Note that, to simplify the presentation, we will assume that it is not possible for two orders to coexist; therefore, we use a unique event *OrderAv* to simultaneously notify all the *Shuttle Agents* that a new order is received.

From Requirements into a System Model

Figure 4 shows the MUS model of the *i*-th *Shuttle Agent*, generated automatically from the requirements above by the SCTL-MUS synthesis algorithm (the *unspecified* events are not represented; the non-possible ones would be indicated next to a symbol like $\neg\bot$).

The MUS graph of the global system, on which integration analysis will be performed, is obtained by applying the rules

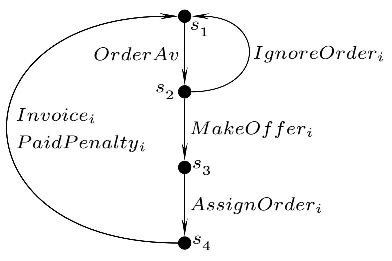


Figure 4: The MUS model of *Shuttle$_i$*.

of the *interleaving* operator $|||_{\mathcal{M}}$. This is an extension of the classical *interleaving* operator of process algebras, intended to support *unspecification* in a way that a composition reflects the potentiality of its components (see [10] for details). For simplicity, Figure 5 shows the MUS graph for the case in which there are only two *Shuttle Agents* in the system, using the following notation: $OrderAv \equiv a$, $MakeOffer_i \equiv b_i$, $IgnoreOrder_i \equiv c_i$, $AssignOrder_i \equiv d_i$, $Invoice_i \equiv e_i$ and $PaidPenalty_i \equiv f_i$.

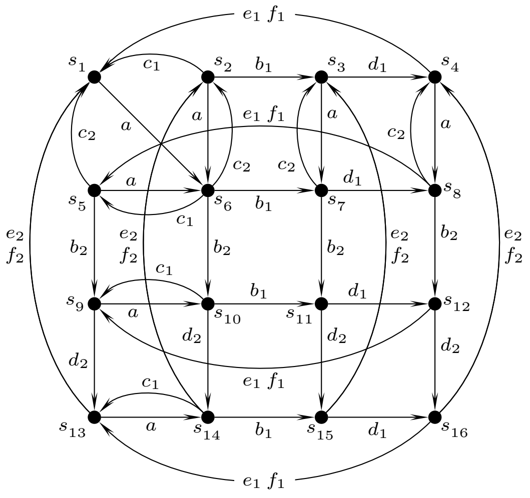

s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16}
a b_1 b_2 c_1 c_2 d_1 d_2
$e_1\, f_1$
$e_2\, f_2$

Figure 5: The system MUS model for two shuttles.

Eliciting Requirements and Scenarios

When the system MUS model has been obtained, the stakeholders can validate the elicited requirements and scenarios by initiating an analysis-revision cycle. Next, we illustrate how integration analysis is performed, and how the results are turned into revisions of the requirements and the scenarios being handled. Particularly, we focus on revisions of Scenario 2 (Figure 3), which has several materializations over the model of Figure 5 for each one of the *Shuttle Agents* in the system. A materialization is any path along the MUS model that follows the sequence of actions of the scenario, possibly with occurrences of other actions in between. Thus, for instance, we have some materializations of Scenario 2 for the first shuttle along the paths $p_1 = \{s_1, s_6, s_7, s_8, s_5\}$, $p_2 = \{s_1, s_6, s_7, s_{11}, s_{12}, s_9\}$ and $p_3 = \{s_1, s_6, s_7, s_{11}, s_{15}, s_{16}, s_{13}\}$.

The Analysis Phase. It is a desirable integration property that an order cannot be assigned to more than one *Shuttle Agent*. That property can be easily expressed using SCTL, relating the fact that an order has been assigned to a given *Shuttle Agent* to the possibility that it can make an invoice:

$$\forall i \in [1,n],\ \mathcal{P}_i \equiv Invoice_i \Rightarrow \bigwedge_{j\neq i}(\neg AssignOrder_j)$$

For the $n = 2$ case, the model-checker detects that these properties are violated in the states s_{12} and s_{15} of the model of Figure 5. For instance, if we are in state s_{12}, the order has already been assigned to the first *Shuttle Agent* —event $AssignOrder_1$ has happened before— but it is still possible to assign it to the second one —$AssignOrder_2$ is possible in s_{12}. In response to these observations, the model-checker generates the following diagnostic information:

$$\Delta_1 = s_{12}[AssignOrder_2] \rightsquigarrow 0$$
$$\Delta_2 = s_{15}[AssignOrder_1] \rightsquigarrow 0$$

Each Δ indicates how the specification of certain events should be modified in certain states to satisfy the properties being analyzed. For example, Δ_1 indicates that, in order to satisfy the property $\mathcal{P}_1$, the event $AssignOrder_2$ should have been specified as *false* in the state s_{12}, whereas it is currently *true*.

The model-checker also provides counter-example scenarios illustrating sequences of events that lead to violations of the properties. Upon successive requests, we can provide as many scenarios as there exist ways to reach the states affected by the Δs. Figure 6 shows one such scenario, that follows the path $p = \{s_1, s_6, s_7, s_{11}, s_{15}, s_{16}\}$ over the model of Figure 5.

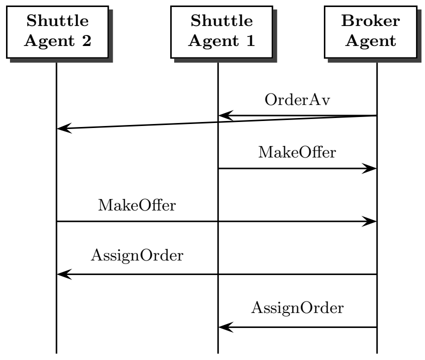


Figure 6: A counter-example scenario.

The Revision Phase. Using the mechanisms we presented in [5], we find that the problems with the integration properties can be solved by changing the original requirements $\mathcal{R}_{4_i}$ for refined versions $\mathcal{R}'_{4_i}$ as follows:

- *"Once an order has been served or ignored by the i-th Shuttle Agent, that order cannot be assigned to another one, but it is possible to receive a new order"*:

$$\mathcal{R}'_{4_i} \equiv (IgnoreOrder_i \vee Invoice_i \vee PaidPenalty_i) \Rightarrow (\bigcirc OrderAv \wedge \bigwedge_{\mathbf{j}\neq\mathbf{i}}(\neg \mathbf{AssignOrder_j}))$$

Figure 7 shows the MUS graph that is obtained by including the revised requirement $\mathcal{R}'_{4_i}$ instead of $\mathcal{R}_{4_i}$, for the case of two *Shuttle Agents*.

In this graph, some of the original materializations of Scenario 2 have been lost, but others are preserved; thereby, as we explained in Section 2.2, the revision entails a specialization of that scenario, to fit the materializations which

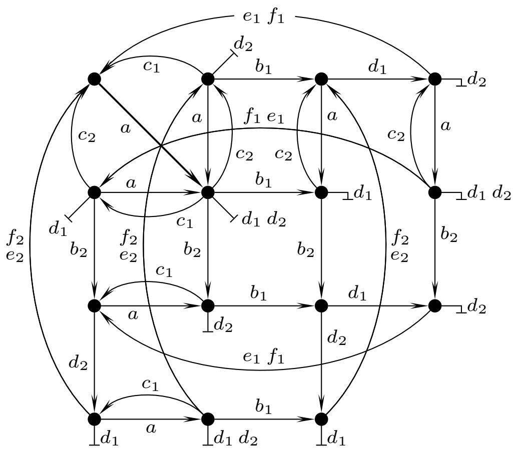


Figure 7: The system MUS model including the revised requirements $\mathcal{R}'_{4_i}$.

are preserved and reflect the ones that have been lost. Neglecting many interesting details, this is done by following the paths of the former materializations, and annotating the changes due to the requirement revisions. For example, the materialization we had for the first *Shuttle Agent* along the path $p_1 = \{s_1, s_6, s_7, s_8, s_5\}$ has been lost, and this is notified by introducing the scenario of Figure 8(a). In contrast, the materialization along $p_2 = \{s_1, s_6, s_7, s_{11}, s_{12}, s_9\}$ is preserved, though with some changes in between that lead to the scenario of Figure 8(b).[1]

Faced with the new scenarios and the revised requirement, the stakeholders can easily reason about the suggested evolution. It is expectable that they would reject it, because the scenarios and the requirement are not representative of what is wanted. According to the scenario of Figure 8(a), the fact that the second *Shuttle Agent* cannot be assigned an offer depends on the other one bidding; likewise, in Figure 8(b), the first *Shuttle Agent* cannot be assigned an order until the second has made an offer for it. Moreover, since the requirement $\mathcal{R}'_{4_1}$ includes $IgnoreOrder_1 \Rightarrow \neg AssignOrder_2$, the first *Shuttle Agent* would prevent the second one from being assigned an order it ignores.

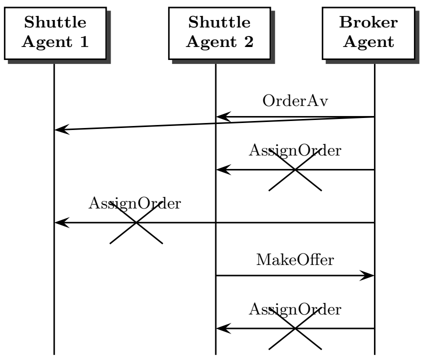
The rejection of a suggested evolution makes the analysis-revision cycle look for alternatives to satisfy the integration properties. The mechanisms of [5] find that the next possibility is to change the original requirements $\mathcal{R}_{5_i}$ for refined versions $\mathcal{R}'_{5_i}$:

- *"If an order has been assigned to the i-th Shuttle Agent, then it can be served in time or provoke a penalty, but it cannot be assigned to another shuttle"*:

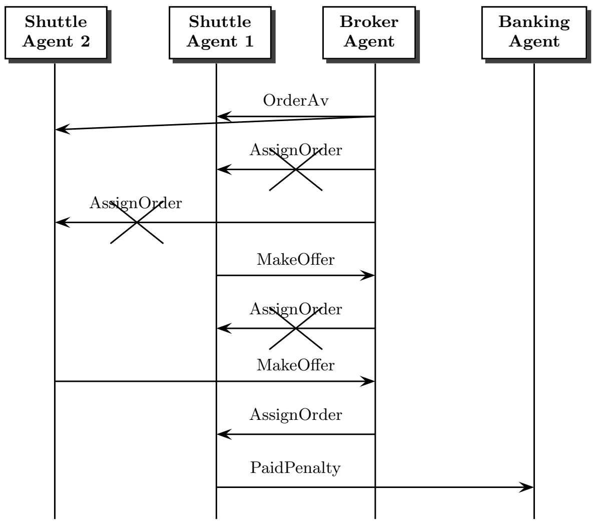
$$\mathcal{R}'_{5_i} \equiv (true \Rightarrow \bigodot AssignOrder_i) \Rightarrow (Invoice_i \wedge PaidPenalty_i \wedge \bigwedge_{\mathbf{j}\neq\mathbf{i}}(\neg \mathbf{AssignOrder_j}))$$

Figure 9 shows the MUS graph that is obtained by including the revised requirement $\mathcal{R}'_{5_i}$ instead of $\mathcal{R}_{5_i}$, for the case of two *Shuttle Agents*. Again, some of the original materializations of Scenario 2 have been lost, but others are

[1]The notation with a crossed-out arrow means that the corresponding action is not allowed to occur until the next action of the scenario takes place.



(a) From a materialization that has been lost.



(b) From a materialization that is preserved.

Figure 8: Revisions of Scenario 2 according to the revised requirements $\mathcal{R}'_{4_i}$.

preserved. The loss of the materialization we had along $p_3 = \{s_1, s_6, s_7, s_{11}, s_{15}, s_{16}, s_{13}\}$ leads to the same scenario of Figure 10(a); on its part, the materialization along $p_2 = \{s_1, s_6, s_7, s_{11}, s_{12}, s_9\}$ produces the scenario of Figure 10(b).

We assume the stakeholders accept this revision, because the scenarios and the revised requirement capture accurately the intention of the integration properties. According to the scenario of Figure 10(a), a *Shuttle Agent* may not be assigned an order for which it has made an offer; on its part, it follows from Figure 10(b) that once an order has been assigned to a given *Shuttle Agent*, it cannot be assigned to any other one. This is precisely the meaning of the revised part of $\mathcal{R}'_{5_i}$: $(true \Rightarrow \bigodot AssignOrder_i) \Rightarrow \bigwedge_{j \neq i} (\neg AssignOrder_j)$.

Because the revision is accepted, the revised requirements $\mathcal{R}'_{5_i}$ become part of the specification of the system, instead of $\mathcal{R}_{5_i}$, and the scenarios of Figures 10(a) and 10(b) replace the original scenario of Figure 3. Now, to go on with the development process, the stakeholders can take a look at the scenario in which a *Shuttle Agent* makes a bid but receives no assignment, to elicit the new requirement *"if it is possible to receive an assignment, then it is also possible that a timer goes off without receiving the assignment"* ($\mathcal{R}_6 \equiv AssignOffer_i \Rightarrow TimeOutOffer_i$). Subsequent analysis and revision activities would support the reasoning needed to further advance towards the desired system.

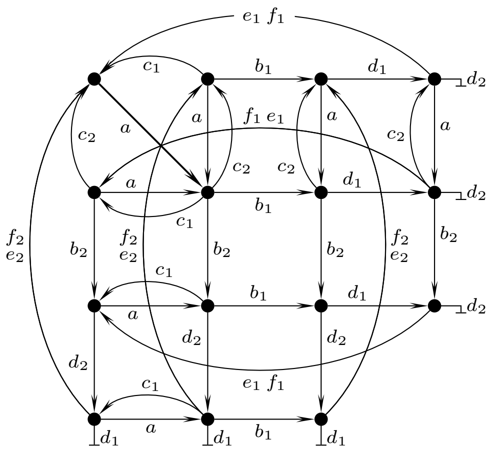


Figure 9: The system MUS model including the revised requirements $\mathcal{R}'_{5_i}$.

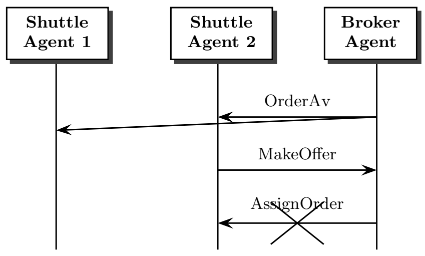
4. OVERVIEW

The SBSE approach we have presented supports the stakeholders in eliciting system requirements and complex scenarios, allowing them to progressively gain understanding about the system they are building. This is due to the fact that the analysis-revision cycle leads to a continuous interaction between the process and the stakeholders, the latter reasoning about the revisions suggested by the former.

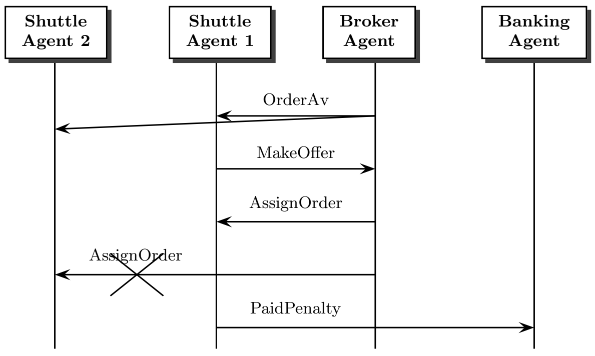
It is important to remark that the revisions of the requirements and scenarios can be easily understood by the stakeholders, because they are not presented as new artifacts —which would be out of context— but as specializations of those elicited by the stakeholders. The methodology also provides great flexibility in managing the revision suggestions, since it is not at all compulsory to accept all or nothing of a revision; for example, the stakeholders can accept some of the scenarios provided, but not the requirement revisions.

The management of incompleteness done by SCTL-MUS results in an incremental approach, in the sense that the system MUS model grows along with the elicitation of new scenarios and requirements. In fact, dealing with *unspecification* is the key to reduce the computational complexity of the SBSE process, in what concerns the size of the system MUS model or the cost of checking whether the scenarios are materialized over it. In addition to the incremental synthesis commented in Section 2.1, *unspecification* allows making incremental model-checking, by reusing verification efforts from previous iterations of the specification. To exploit this feature, we have adapted for the SBSE process the mechanisms we presented in [9]; furthermore, inspired by the work in [2], we are currently initiating research on applying *lightweight* techniques.

Once we have finished implementing the above-mentioned features, we will have completed an incremental development methodology that allows refining three modeling perspectives of a system (scenarios, requirements and state ma-



(a) From a materialization that has been lost.



(b) From a materialization that is preserved.

Figure 10: Revisions of Scenario 2 according to the revised requirements $\mathcal{R}'_{5_i}$.

chines) at the same time, through analysis-revision cycles. This will greatly facilitate the identification of suitable evolutions of a specification, doing validations and verifications over a unique knowledge base, captured in the requirements.

5. ACKNOWLEDGEMENTS

This work has been partially funded by the Xunta de Galicia Research Project PGIDIT04PXIB32201PR.

6. ADDITIONAL AUTHORS

Additional author: Belén Barragáns-Martínez, from the Telematics Engineering Dept. of the University of Vigo (Spain). Email: belen@det.uvigo.es.

7. REFERENCES

[1] R. Banach and M. Poppleton. Retrenching partial requirements into system definitions: a simple feature interaction case study. *Requirements Engineering Journal*, 8(4):226–288, 2003.

[2] Y. Bontemps and P. Heymans. As fast as sound (lightweight formal scenario synthesis and verification). In *Proceedings of SCESM'04 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 27–34, Edimburgh, UK, May 2004.

[3] G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Proceedings of CAV'99 Conference on Computer-Aided Verification*, pages 274–287, Trento, Italy, July 1999.

[4] A. S. d'Avila Garcez, A. Russo, B. Nuseibeh, and J. Kramer. An analysis-revision cycle to evolve requirements specifications. In *Proceedings of ASE'01 Conference on Automated Software Engineering*, pages 354–358, San Diego, USA, Nov. 2001.

[5] J. García-Duque, J. J. Pazos-Arias, and B. Barragáns-Martínez. An analysis-revision cycle to evolve requirements specifications by using the SCTL-MUS methodology. In *Proceedings of RE'02 Conference on Requirements Engineering*, pages 282–288, Essen, Germany, Sept. 2002.

[6] M. Glinz. Improving the quality of requirements with scenarios. In *Proceedings of World Congress for Software Quality*, pages 55–60, Yokohama, Japan, Sept. 2000.

[7] P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of the CONCUR'01 Conference on Concurrency Theory*, pages 426–440, Aalborg, Denmark, 2001.

[8] D. Harel. From play-in scenarios to code: An achievable dream. *Computer*, 34(1):53–60, 2001. IEEE Press.

[9] M. López-Nores, R. P. Díaz-Redondo, J. J. Pazos-Arias, and J. García-Duque. An improved repository system for effective and efficient reuse of formal verification efforts. In *Proceedings of APSEC'04 Conference on Software Engineering*, pages 38–45, Busan, South Korea, Dec. 2004.

[10] M. López-Nores, J. J. Pazos-Arias, J. García-Duque, and B. Barragáns-Martínez. Tracing integration analysis in component-based formal specifications. In *Proceedings of FMOODS'05 Conference on Formal Methods for Open Object-based Distributed Systems*, Athens, Greece, June 2005. To appear.

[11] J. Moffett. A model for a causal logic for requirements engineering. *Journal of Requirements Engineering*, 1:27–46, 1996.

[12] J. J. Pazos-Arias and J. García-Duque. SCTL-MUS: A formal methodology for software development of distributed systems. A case study. *Formal Aspects of Computing*, 13:50–91, 2001.

[13] The shuttle system case study. `http://wwwcs.upb.de/cs/ag-schaefer/CaseStudies/ShuttleSystem/`.

[14] S. Uchitel, J. Kramer, and J. Magee. Behavior model elaboration using Partial Labelled Transition Systems. In *Proceedings of ESEC/FSE'03 meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 19–27, Helsinki, Finland, Sept. 2003.

[15] S. Uchitel, J. Kramer, and J. Magee. Synthesis of behavioral models from scenarios. *IEEE Transactions on Software Engineering*, 29(2), Feb. 2003. IEEE Press.

[16] A. van Lamsweerde. *The future of software engineering*, chapter Formal specification: A roadmap, pages 147–159. ACM Press, 2000.

[17] J. Whittle and I. Krüger. A methodology for scenario-based requirements capture. In *Proceedings of SCESM'04 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, Edimburgh, UK, May 2004.