Tracing Integration Analysis in Component-Based Formal Specifications*

Martín López-Nores, José J. Pazos-Arias, Jorge García-Duque,
Belén Barragáns-Martínez, Rebeca P. Díaz-Redondo, Ana Fernández-Vilas,
Alberto Gil-Solla, and Manuel Ramos-Cabrer

Department of Telematics Engineering, University of Vigo, 36310 Vigo, Spain
{mlnores,jose,jgd,belen,rebeca,avilas,agil,mramos}@det.uvigo.es

Abstract. The correctness of a component-based specification is not guaranteed by the correctness of its components alone; on the contrary, integration analysis is needed to observe their conjoint behavior. Existing approaches often leave the results of the analysis at the level of the integrated system, without tracing them onto the corresponding components. This effectively results in loss of architecture, as it is no longer possible to reason over those components and evolve their specification while keeping the results of integration analysis.
This paper presents a formal approach to automatically translate changes on the integrated system into revisions of the components and the architecture initially defined by the developers. Several architectural alternatives are provided that, besides allowing developers to reason about the system from different points of view, promote its correct modularization in two overlapping perspectives: the encapsulation of crosscutting concerns and the elaboration of the architecture desired for the final implementation.

1 Introduction

Component-based approaches have been around for a long time as a means to split complexity in software development, promising better understanding of a system by its developers, improved quality and easier maintenance. A more recent idea to improve software engineering practice has been to apply incremental development techniques, which are based on obtaining successive revisions of a system until achieving the desired functionality. These techniques are especially suitable to deal with changeable specifications, and also with maintenance and evolution tasks.

Due to the well-known problem of *feature interaction*, the correctness of a system is not guaranteed by the correctness of its parts, considering these in isolation. On the contrary, certain properties can only be verified by observing the conjoint operation of several components. This points out the need for *integration analysis*.

Current approaches to component-based development often limit themselves to finding whether integration analysis succeeds. In case of failure, no information is given on how to modify the components, forcing the developers to attempt manual changes until getting a positive response, which is clearly unsatisfactory. The ideal would be to determine the changes needed to satisfy the integration properties over the integrated system

* Work partially funded by the Xunta de Galicia Research Project PGIDIT04PXIB32201PR.

M. Steffen and G. Zavattaro (Eds.): FMOODS 2005, LNCS 3535, pp. 147–162, 2005.

(i.e., where the properties can be observed), and then trace those changes automatically to the components. Unfortunately, support for this feature is missing nowadays, resulting in an effective *loss of architecture*, as it is no longer possible to reason over the individual components while keeping the results of integration analysis. So, the ability to trace integration results would represent a major aid to incremental development.

In this paper, we present a formal methodology to tackle this concern. Our proposal is to automatically translate the changes resulting from integration analysis into revisions of the components and the architecture defined by the developers. These are provided with several architectural views, that promote the correct modularization of a system and help to elaborate the architecture wanted for its final implementation.

The paper is organized as follows. Section 2 outlines the development model in which our proposal takes place, with Section 3 describing our methodology to trace integration analysis. Section 4 presents a simple example on applying this methodology, which is later discussed in Section 5. Section 6 comments our ongoing work in the line of this paper, and Section 7 discusses related work. Appendixes A, B and C gather the technical details not included elsewhere.

2 The (SCTL/MUS)^A Context

The context for our work lies in the SCTL-MUS methodology [14], a formal approach to the specification of reactive systems that models the usual way in which a system is specified: starting with an initially rough idea of the desired functionality, this is successively refined until the specification is complete.

SCTL-MUS combines property-oriented and model-oriented formal description techniques: on the one hand, the many-valued logic SCTL (*Simple and Causal Temporal Logic*) is used to express the system's functional requirements; on the other, the graph formalism MUS (*Model of Unspecified States*) is employed to model systems for validation and formal verification purposes.

SCTL statements 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.

This causal semantics allows expressing under what circumstances during the operation of a system shall a given condition be satisfied, so that the premise and the temporal operator of a statement delimit the applicability of its consequence.

Given a set of requirements expressed in SCTL, the synthesis algorithm of SCTL-MUS attempts to generate a MUS graph that adheres to all of them. As a distinctive feature with respect to traditional *Labeled Transition Systems* (LTS), the events of a MUS graph can be not only *possible* (*true*, 1) or *non-possible* (*false*, 0) in the different states; on the contrary, if there are no requirements affecting the specification of an event in a given state, that event is given the value $\frac{1}{2}$ (*not-yet-specified* or *unspecified*) in that state. Figure 1 shows two SCTL requirements and the MUS graph that implements them – note that we do not explicitly represent *unspecified* actions (like a in state s_2), because $\frac{1}{2}$ is the default value; instead, we do represent *false* actions (like b in s_2), placing a symbol like $\neg$ next to every state where a given event is *non-possible*.

$$r_1 \equiv a \Rightarrow \left(\neg b \wedge \bigcirc c\right)$$
$$r_2 \equiv c \Rightarrow \bigcirc a$$

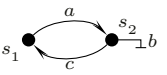


Fig. 1. Two SCTL requirements and the MUS graph that implements them

The commented notion of *unspecification* was introduced to deal with the incompleteness inherent to the intermediate stages of an incremental process, so as to enable reasoning about evolutions and satisfaction tendencies in the formal verification process. A partial specification represents all the systems into which it can evolve by adding new requirements, and the MUS formalism was devised to explicitly capture that potentiality. This is achieved through the inclusion of 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 [14] for the details). With this definition, the addition of 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. These features are not fully catered for by other formalisms intended to support partial specifications, like KPSs [3] or MTSs [6].

2.1 An Incremental Approach to Component-Based Specification

SCTL-MUS lacks the notion of *architecture*, which hinders the simultaneous work of several developers and makes specifications unmanageable for large systems. Moreover, it only defines mechanisms to handle evolutions of a specification to satisfy new requirements, consistent with the current ones, but provides no support to modify the current requirements so as to solve inconsistencies or revise previous design decisions.

To solve these flaws, we are now working on (SCTL/MUS)^A, a fully incremental methodology aimed at facilitating task division and collaborative work. The new methodology, whose motivation was given in [11], inherits most of the philosophy of SCTL-MUS, keeping its dual approach in the use of formal description techniques, its iterative life cycle and its notion of *unspecification*. It also uses the same formalisms (SCTL and MUS), to reason over individual components and their compositions, but extends this basis to handle component-based specifications.

(SCTL/MUS)^A accommodates the multiple parts of a specification in *composition layers* that relate components to the compositions in which they take part, with the overall system at the top. The composition operators allowed have been borrowed from the LOTOS process algebra, though adapting them to the three-valued domain of MUS. This is the key to make compositions reflect the *unspecification* – and, therefore, the potentiality – of their forming components, which is essential to support the incremental approach. Appendix A explains this vision on the *selective parallel composition* operator (denoted by $||[\Lambda]||_{\mathcal{M}}$), which is a powerful way to express the concurrent operation of several components. In brief, a composition $C_1||[\Lambda]||_{\mathcal{M}} C_2$ can advance through an event $a \notin \Lambda$ if a is *possible* in either C_1 or C_2, but it can advance through an event $b \in \Lambda$ only if b is *possible* in both of them; thus, if Λ is the empty set, the effect is that of pure interleaving (in this case, the operator is denoted by $|||_{\mathcal{M}}$).

2.2 Orientation to Aspects

(SCTL/MUS)^A fits within the *Early-Aspects* initiative [16], which aims at extending the principles of aspect-oriented programming [9] to the phases of requirements engineering and architectural design. Aspect-orientation is a way to achieve modularizations that facilitate the management of *crosscutting functionality*, i.e., functionality that appears scattered through the parts of any decomposition in objects. This is done by introducing *aspects* that encapsulate the crosscutting functionality, and by defining mechanisms to *weave* (combine) the aspects with the components they crosscut.

Due to the slight notion of structure available during requirements elicitation tasks, our vision is not to make an explicit distinction between components and aspects; instead, any component that is combined with the composition of several others may be seen as an aspect, because it crosscuts their functionality. Nonetheless, treating a piece of functionality as an aspect is only justified when it can be traced into modifications of the crosscut components as a meaningful addition to their functionality.

In line with the ideas discussed in [13] and the works on *multi-dimensional separation of concerns* [18], the management of aspects in (SCTL/MUS)^A is linked to allowing developers to handle multiple *architectural views*, each one defining a different decomposition of the system or any of its components. Different decompositions enable different reasonings, and it is possible to work over any one of them, since we can propagate what is done on a given view to the others. This way, multiple developers can contribute to construct the desired system by reasoning from different perspectives.

3 A Methodology for Integration Analysis

This section introduces the methodology we have defined to perform integration analysis in (SCTL/MUS)^A, which is targeted at facilitating incremental development. The fundamental idea is that the analysis of a system should not be delayed until its parts have been completely developed, in order to prevent doing much work over incorrect foundations. Consequently, (SCTL/MUS)^A allows integration analysis to be done at intermediate stages of the specification process. Furthermore, it supports analysis at any level of composition, not necessarily on the whole system. Despite, for simplicity, we will refer to the composition being analyzed as "*the system*".

Figure 2 illustrates the steps of the methodology, which we proceed to describe.

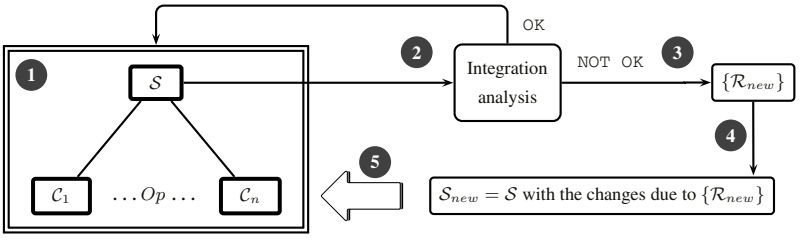


Fig. 2. The complete cycle of integration analysis

3.1 Specification

The starting point (step 1 in Fig. 2) is the specification of the system ($\mathcal{S}$) as the combination of several components ($\mathcal{C}_i$) by means of composition operators (Op). The analysis is done on the MUS graph of the composition, that is computed from the MUS graphs of the components by following the rules of the composition operators. The MUS graphs of the components may have been derived from other components or, for those at the lowest composition levels, synthesized from a set of SCTL requirements.

3.2 Integration Analysis

Once available, the MUS graph of the system is subject to verifying the integration properties (step 2 in Fig. 2), to find whether the system satisfies them (`OK`) or it does not satisfy them *yet* (`NOT OK`).

If the system satisfies the properties, the analysis is finished, and the developers can continue working over the unchanged current specification. On the contrary, if the system does not fulfill the properties, an evolution of its specification is needed to satisfy the developers' expectancies. In this case, as commented in Sect. 1, the goal of the methodology is twofold: to find the modifications needed for the system to satisfy the properties stated for it, and to trace those changes onto the components whose conjoint behavior is being analyzed. Our proposal, as explained below, is to aid the developers in the first task, and to fully automate the second.

3.3 The Creative Part

Determining the changes needed for a system to satisfy certain objectives (step 3 in Fig. 2) requires participation from the developers, because many viable alternatives may exist in the general case. To help in this process, we have adapted the *analysis-revision cycle* presented in [5], that automatically provides some of those alternatives.

An important thing to note here relates to the concept of *unspecification* (Section 2), that allows us to conjugate the two main methods proposed in literature to revise the specification of a system: *refinements* [10, 17] and *retrenchments* [15]. Because of *unspecification*, the developers can evolve a system in two different ways:

(**i**) **By retrenchments, for properties that are explicitly violated.** In this case, the system includes unwanted behavior, that must be eliminated. The analysis-revision cycle points out the circumstances under which the properties are violated, and suggests possibilities to solve the problem.

(**ii**) **By refinements, for properties that are not explicitly violated *yet*.** These properties, which are neither fulfilled nor violated yet (they are *unspecified*), indicate that behavior must be added to make the specification satisfy them. Here, the analysis-revision cycle identifies evolutions that would make the specification violate the properties, and suggests modifications to conduct it the other way.

Assisted by the suggestions of the analysis-revision cycle on how to evolve the system, the developers are expected to come up with a set of requirements ($\{\mathcal{R}_{new}\}$) that specify the changes to be done on the system.

3.4 The Automated Part (I): Incorporating the New Requirements

Once the changes have been decided, it is easy to apply them over the MUS graph of the integrated system (step 4 in Fig. 2), but the result is no longer obtainable from the MUS graphs of the original components. At most, it can be expressed as a new system where the architecture has been lost: $\mathcal{S}_{new} = \mathcal{S}$ with the changes due to $\{\mathcal{R}_{new}\}$.

3.5 The Automated Part (II): Tracing Changes onto the Original Specification

In order to avoid loss of architecture, and to permit future iterations in the specification of the individual components, our approach traces the changes done on the system into revisions of the original architecture and components (step 5 in Fig. 2). We do this by automatically refactoring $\mathcal{S}_{new}$ into two different architectural views, as shown in Fig. 3:

(i) In the first view, a new component is created that materializes the new requirements and that, combined with the original components, makes the system behave as intended. As shown in Fig. 3, the new architecture only adds the new component ($\mathcal{C}_{n+1}$), preserving the original ones and the ways they were combined (Op).
The new component, that manifests the crosscutting nature of the integration properties, may have significance in the domain of application of the system, in which case it can be further developed (in functionality and architecture), and may even be reused for other systems.

(ii) Just because the new component may be meaningless in the implemented system, our methodology provides a second architectural view, where its functionality is discharged over the original components. In other words, since the new component may be seen as an aspect ($\mathcal{A}$) that modifies the original components, we offer a view that re-expresses the original architecture in terms of modified components ($\mathcal{C}_i^*$), possibly changing the original composition operators (note the Op'' instead of Op in Fig. 3). We represent the weaving operation by means of a newly-defined operator, that we call "*projection*" and denote by $\leftarrow^*$.

To complete the process of tracing changes, (SCTL/MUS)^A supports the automatic reformulation of the requirements of the original components. Using mechanisms like the ones presented in [5], we modify the SCTL requirements provided by the developers from the transformations done over the MUS models that implement them, which allows the developers to see the changes made to the system expressed in the same language used to specify it. This is an essential aid to go on with the incremental specification process: the requirements are the mechanism by which developers express their conception of the system, and so their formulation holds the key to understand what is being constructed.

Reformulating requirements is necessary in the second architectural view, to enable reasoning about some behavior of a given component that can only be observed in its combination with the aspect. In this case, the requirements for $\mathcal{C}_i^*$ are derived from the requirements of the corresponding $\mathcal{C}_i$. As for the new components $\mathcal{C}_{n+1}$ and $\mathcal{A}$ in Fig. 3, at most we can annotate the situation and the set of requirements that led to their appearance – inventing a set of requirements from which their MUS graphs could be generated is purposeless, because those requirements would not capture any expressive effort from the developers.

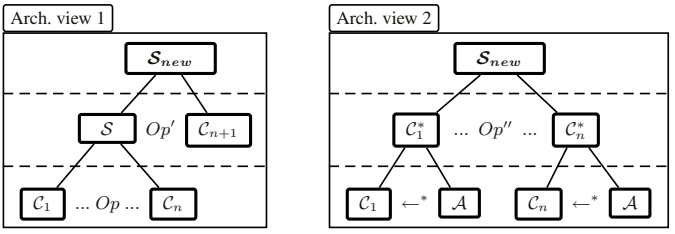


Fig. 3. Two architectural views for the new system

4 An Example on Synchronization Concerns

This section illustrates the methodology presented in Sect. 3, with an example of tracing synchronization requirements over the parallel composition of several components. We describe an evolution of the system based on suppressing unwanted behavior, and show how the two architectural views are automatically computed. These include the restrictions imposed at the composition level while keeping the original architecture. Details about the algorithms applied are left for Appendixes B and C.

4.1 Specification

Let $Sender$ be a component whose functionality, at the current stage of development, is "*a $Sender$ starts a transmission when it has data to send; it waits for new data after each transmission*". A system is wanted that defines a communications network with n senders operating on a shared channel. To model this, the developers initially specify the system $\mathcal{S}^n$ as the interleaved combination of n instances of component $Sender$:

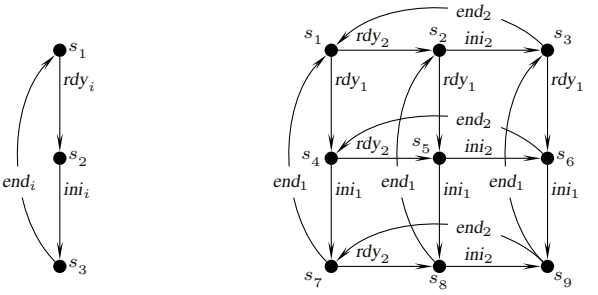
$$\mathcal{S}^n \equiv Sender_1 \ \ldots \ |||_{\mathcal{M}} \ \ldots \ Sender_n = \underset{i=1\ldots n}{|||_{\mathcal{M}}} \ Sender_i \qquad (1)$$

Figure 4(a) shows the MUS graph of the i-th sender in the system, in its current status of specification. The sender waits for data in state s_1 (the initial state), and transitions into s_2 when action rdy_i occurs, meaning that new data are available. Once in s_2, the sender can begin transmitting the data by executing ini_i. It stays at s_3 until the transmission finishes (action end_i), and then goes back to s_1 to wait for new data. All the other actions are *not-yet-specified*.

4.2 Integration Analysis

The specification of the senders is not yet complete, because some *unspecification* remains in their MUS graphs. However, it may be wise to analyze their conjoint behavior before completing them, to make sure that what has been specified so far is correct, avoiding futile efforts in evolving incorrect specifications.

As the senders will operate on a shared channel, it is necessary to ensure that it is not possible for several of them to be transmitting simultaneously. According to the MUS graph of Fig. 4(a), a sender is transmitting only when it is in state s_3, and this can



(a) $\mathcal{M}_{Sender_i}$ (b) $\mathcal{M}_{\mathcal{S}^2}$

Fig. 4. The MUS graphs of $Sender_i$ and $\mathcal{S}^2 \equiv Sender_1 \,|||_{\mathcal{M}}\, Sender_2$

be referred to as the only state where action *end*$_i$ is *possible*. Therefore, for the $n = 2$ case, the developers can specify the desired property as $\mathcal{P} \equiv true \Rightarrow \neg(end_1 \wedge end_2)$ – since the premise ($true$) is satisfied in every state, the property is violated in those states where both *end*$_1$ and *end*$_2$ are *possible*.

The first step to analyze the satisfaction of $\mathcal{P}$ is to obtain the MUS graph of the system $\mathcal{S}^n$, by combining the graphs of n senders according to the rules of the $|||_{\mathcal{M}}$ operator. Figure 4(b) shows the MUS graph of the network with two senders, $\mathcal{M}_{\mathcal{S}^2}$.

4.3 The Creative Part

Analyzing the property $\mathcal{P}$ reveals that it is explicitly violated in the state s_9 of $\mathcal{M}_{\mathcal{S}^2}$, since both *end*$_1$ and *end*$_2$ are *possible* there. Therefore, as explained in Sect. 3.3, a retrenchment is needed to ensure proper behavior. From among the possible evolutions suggested by the analysis-revision cycle, the developers decide to incorporate the requirements of Eq. (2), which prevent one sender from starting a transmission while the other is transmitting:

$$\{\mathcal{R}^2_{new}\} = \{\mathcal{R}_1, \mathcal{R}_2\}, \quad \text{where} \quad \begin{cases} \mathcal{R}_1 \equiv end_1 \Rightarrow \neg ini_2 \\ \mathcal{R}_2 \equiv end_2 \Rightarrow \neg ini_1 \end{cases} \tag{2}$$

For the general case of n senders, the new requirements would be those of Eq. (3):

$$\{\mathcal{R}^n_{new}\} = \{\mathcal{R}_i\}_{1 \le i \le n}, \quad \text{where} \quad \mathcal{R}_i \equiv end_i \Rightarrow \bigwedge_{j \neq i} \neg ini_j \tag{3}$$

4.4 The Automated Part

To describe how the new requirements are applied onto the MUS graph of the original system ($\mathcal{S}^n$), we consider again the case $n = 2$, without loss of generality. Due to $\mathcal{R}_1$, the specification of action *ini*$_2$ changes to *false* in the states s_7, s_8 and s_9 of $\mathcal{M}_{\mathcal{S}^2}$; similarly, $\mathcal{R}_2$ changes the specification of *ini*$_1$ to *false* in s_3, s_6 and s_9. The resulting graph ($\mathcal{M}_{\mathcal{S}^2_{new}}$) is shown in Fig. 5, where it can be seen that the problematic state s_9 is now unreachable – this is the desired effect of the new requirements.

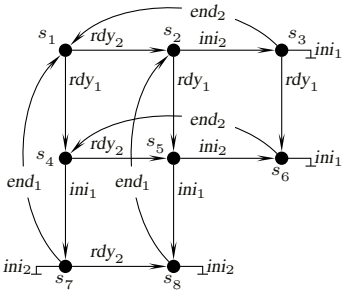


Fig. 5. The MUS graph of the revised overall system, $\mathcal{M}_{\mathcal{S}^2_{new}}$

Returning to the general case, the important thing to note is that, despite $\mathcal{M}_{\mathcal{S}^n_{new}}$ represents the desired functionality, it is not expressed in terms of the original components. In fact, it represents a system without architecture (remember point number 4 in Fig. 2), which prevents from continuing its modular development. (SCTL/MUS)^A addresses this problem by automatically refactoring $\mathcal{S}^n_{new}$ into two architectural views, which are revisions of the original system.

The first architectural view includes a new component, $Synchronizer^n$, that materializes the new requirements and ensures the correct operation of the senders. This view is expressed in Eqs. (4) and (5), where $\Lambda^n_{sync} = \{ini_i, end_i\}_{1 \le i \le n}$.

$$\mathcal{S}^n_{new} = \mathcal{S}^n \; |[\Lambda^n_{sync}]|_{\mathcal{M}} \; Synchronizer^n \tag{4}$$

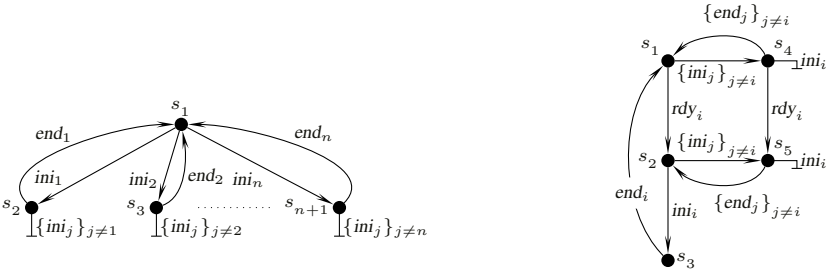
$$\mathcal{S}^n_{new} = (\; |||_{\mathcal{M}}\limits_{i=1\ldots n} \; Sender_i) \; |[\Lambda^n_{sync}]|_{\mathcal{M}} \; Synchronizer^n \tag{5}$$

The $Synchronizer^n$ component, whose MUS graph is shown in Fig. 6(a), and the Λ^n_{sync} set of actions are automatically derived from from $\mathcal{S}^n$ and $\{\mathcal{R}^n_{new}\}$ using the algorithm described in Appendix B. This algorithm guarantees by construction that the MUS graph of the composition $\mathcal{S}^n \; |[\Lambda^n_{sync}]|_{\mathcal{M}} \; Synchronizer^n$ is equal to $\mathcal{M}_{\mathcal{S}^n_{new}}$.

In the second architectural view, $Synchronizer^n$ is seen as an aspect that crosscuts the original senders. This allows re-expressing the system $\mathcal{S}^n_{new}$ as the composition of n modified senders, in a way that resembles the original architecture of Eq. (1). The new expression is shown in Eq. (6), where the Λ^n_{sync} set of actions is the same as above.

$$\mathcal{S}^n_{new} = |[\Lambda^n_{sync}]|_{\mathcal{M}}\limits_{i=1\ldots n} \; Sender^n_{csma_i} \tag{6}$$

We refer to the modified components as $Sender^n_{csma}$, because they have the basic functionality of the original $Sender$, though enhanced with the capability to prevent collisions on a communications channel shared with other $Sender^n_{csma}$ components. Figure 6(b) shows the MUS graph of the i-th instance of $Sender^n_{csma}$ in the system, where it can be seen that the start of a transmission is forbidden while another sender is using the channel (ini_i is *false* in s_4 and s_5). Therefore, a retrenchment of the overall system has led to a loss of *unspecification* of the MUS graphs of its forming components; as noted in Sect. 3.5, this can be translated into a refinement of the requirements provided for those components by the developers.



(a) $\mathcal{M}_{Synchronizer^n}$ (b) $\mathcal{M}_{Sender^n_{csma_i}}$

Fig. 6. The MUS graphs of $Synchronizer^n$ and $Sender^n_{csma_i}$

$Sender^n_{csma_i}$ results from weaving $Synchronizer^n$ with the $Sender_i$ components, by applying the *projection* operator as described in Appendix C (again, the composition of the $Sender^n_{csma_i}$ components is guaranteed to yield a MUS graph equal to $\mathcal{M}_{\mathcal{S}^n_{new}}$). So, Equation (6) can be rewritten with an additional lower composition layer:

$$\mathcal{S}^n_{new} = |[\Lambda^n_{sync}]|_{\mathcal{M}} \underset{i=1\ldots n}{} (Sender_i \leftarrow^* Synchronizer^n) \tag{7}$$

To sum up, Equations (4) and (5) describe two composition layers in the first architectural view proposed for $\mathcal{S}^n_{new}$, and Eqs. (6) and (7) do the analogous with the second one. These two views are illustrated in Fig. 7.

5 Analyzing the Example

About the Architectural Views. Handling different architectural views of a system is useful to advance towards the architecture desired for its implementation, while keeping the ability to reason about different features of its functionality. In our example, if the developers expect $Synchronizer^n$ to have significance of its own in the implemented system (as an arbitrage mechanism controlling the concurrent execution of the senders), they should continue evolving the first architectural view. Conversely, they should work on the second if synchronization will be up to the senders themselves.

Multiple views also help to gain understanding about the desired functionality, because some evolutions of the system may be easier to identify from certain perspectives. For example, moving from CSMA to CSMA/CD synchronization is easier (and less error-prone) to attain by evolving the $Synchronizer^n$ component and projecting it again over the senders, than by directly modifying these. This witnesses the advantages of encapsulating crosscutting concerns.

Furthermore, from the second architectural view, we can recover the notion of a sender in which the start of a transmission can be delayed due to environmental conditions, a feature that is not present in the other view. Thus, $Sender^n_{csma_i}$ can be taken as a reusable component, with the basic functionality of a generic sender and the added value that, in the presence of other components of the same kind, it incorporates additional functionality to model synchronization. Even the $Synchronizer^n$ component may be reused, since it works as a *mutual exclusion semaphore*.

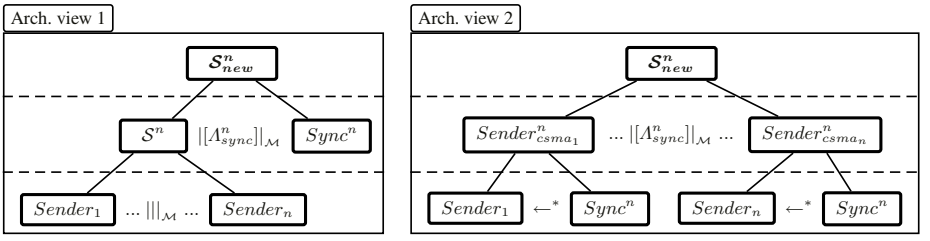


Fig. 7. The two architectural views for the system $\mathcal{S}^n_{new}$

About Tracing Integration Results. Tracing the results of integration analysis back to the composed components is essential to continue their incremental development. It is particularly frequent that a component needs to incorporate considerations from higher layers before doing new iterations of its specification. This would be the case if we wanted to enhance $Sender^n_{csma_i}$ with behavior to execute when the state s_5 is reached (after receiving data to send, the attempt to use the channel is blocked by other sender). Thus, the methodology presented here, together with the management of *unspecification*, endows (SCTL/MUS)^A with great levels of incrementality.

About the Algorithms and Formalisms. Even though the methodology of Sect. 3 is general, we have focused on handling retrenchments over compositions with the $|[\Lambda]|_{\mathcal{M}}$ operator, for which the algorithms described in Appendixes B and C are always valid. This specificity does not imply loss of generality or interest, because parallel composition is, by far, the most important construct in the field of reactive systems (see [1]), being not only valid to model distribution or the concurrent operation of several components, but also for the general management of interacting features.

The algorithm to derive the synchronizer component succeeds at encapsulating the crosscutting concerns. As a proof for this, note that, during integration analysis, the developers referred to the fact that $Sender_i$ was transmitting by the possibility of executing end_i, which would be incorrect if the senders had more actions between *ini* and *end*. Nonetheless, $Synchronizer^n$ remains valid even if new actions are inserted between *ini* and *end* in a posterior evolution. This is because $Synchronizer^n$ captures the intention of the new requirements in the context of their formulation, having nothing to do with the component that would be synthesized from those requirements alone.

On its part, the projection algorithm modifies the original components by adequately introducing *true* and *false* events. It forbids just what the synchronization requirements forbid, and maintains the *unspecification* not affected by those requirements so as not to limit future evolutions unnecessarily. Besides, it does not incorporate irrelevant facts about the environment into the specification of the individual components (note that there is no trace of rdy_j in $Sender^n_{csma_i}$, for any $j \neq i$).

As a final remark, it must be noted that our methodology is not dependent on using SCTL, since other logics could be employed – our use of SCTL is motivated by its causal semantics, which we consider adequate for the first stages of the development process, in line with the comments given in [12]. In contrast, the management of MUS models is indispensable for several reasons: i) to allow deciding when to apply

refinements or retrenchments, ii) for the transformation algorithms to work, and iii) to effectively support an incremental specification approach.

6 Work in Progress

An immediate continuation of our work is to present the algorithms that apply the methodology of Sect. 3 to handle refinements of the integrated system, providing again two architectural views: one in which a new component gathers the added behavior, and other in which the new behavior is allotted over the original components. Analogous comments hold for the algorithms that perform the transformations over compositions involving other operators than $|[\Lambda]|_{\mathcal{M}}$.

We are also considering the possibility of offering other meaningful architectural alternatives, besides the two commented here. For instance, while the second view brings into the components what their environment forbids about them, a third one could incorporate what each component prohibits in its environment. Another option would be to discharge the functionality of the aspect onto a subset of the original components, though we conjecture that this may not be possible in all cases.

Our proposal in this paper represents an aid to go from the customers' requirements to a system that satisfies them, allowing to progressively conduct the specification towards the desired architecture while not preventing reasoning over the different concerns in isolation. In this regard, we intend to provide additional assistance by supporting the automatic identification of crosscutting functionality and its posterior extraction into aspects, which may be weaved with the crosscut components through parallel or sequential composition. To handle the first case, we are experiencing with algorithms similar to those of *bipartition* employed in [2], though considering *unspecification* and taking the requirements into account; to handle the second, we are currently involved with the definition of a suitable language for the definition of *pointcuts*.

The identification of aspects will be helpful to assist the developers when they fail to identify a modular decomposition of a system's functionality. As argued in Sect. 5, encapsulating crosscutting concerns is desirable even when they will not represent components in the final implementation. Besides, we remark the importance of an early identification, before the aspects get so tangled in the hierarchy of components that their identification becomes unfeasible – we believe that the incrementality of (SCTL/MUS)^A can be a good basis to advance research in this topic.

7 Related Work

The work presented here is involved with the conjoint treatment of requirements and architecture in the specification of software systems, an area that has received little attention to date. In [20], it is noted that "*little work has been devoted to techniques for systematically deriving architectural descriptions from requirements specifications*", noticing that this is somewhat paradoxical, as long as architecture has a profound impact on the achievement of a system's goals. The same author discusses in [19] the desirability of doing analysis on specification drafts and carrying out development in an incremen-

tal fashion, whereas "*many specification techniques require that the specification be complete in some sense before the analysis can start*".

To the best of our knowledge, ours is the first formal approach that completes the cycle for integration analysis shown in Fig. 2. Elementary approaches finish at step 2, only informing about whether the analysis succeeds or not. In other cases, counterexamples are provided to help finding the source of the errors [7], but this is limited assistance (insufficient to claim step 3), because no guidance is given on how to modify the system. Even when such an aid is provided – as in [4] –, changes are made only at the level being analyzed, not being traced throughout the architecture of the system.

We only found works on traceability within the paradigm of *assume-guarantee reasoning* [8], which attempts to characterize the influence of the environment over the components to guarantee the satisfaction of the integration properties, resembling what we do with our second architectural view. However, the techniques based on this paradigm usually demand manual intervention and great expertise in formal verification, deviating intellectual efforts from the specification tasks. Our proposal addresses these shortcomings by allowing developers to reason continuously over the problem at hands; the key to achieve it is that we lean intermediately against an operational model (MUS) instead of reasoning directly over mathematical formulae (requirements).

Some transformations similar to the ones presented in this paper have been applied in the Lotosphere environment [2], though with remarkable differences. In Lotosphere, transformations were applied to refine abstract specifications into component processes, with those specifications typically describing the service offered by a communications protocol that was completely known in advance. In contrast, the vision in $(\text{SCTL/MUS})^A$ is that no complete idea of the desired functionality is known beforehand, and that developers gain knowledge about the desired system as the development progresses. Thereby, we do not take a top-down approach to development, but an incremental one in which requirements and architecture can be elaborated in parallel. This way, we relate refinements to a progressive removal of incompleteness, which is supported by the management of *unspecification* and the reformulation of the requirements.

References

1. K. Altisen, F. Maraninchi, and D. Stauch. Exploring aspects in the context of reactive systems. In *Proceedings of Workshop on Foundations of Aspect-Oriented Languages (FOAL), in conjunction with AOSD*, pages 45–51, Lancaster, UK, 2004.
2. T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors. *Lotosphere: Software development with LOTOS*. Kluwer Academic Publishers, 1995.
3. G. Bruns and P. Godefroid. Model checking partial state spaces with 3-valued temporal logics. In *Proceedings of the 11th International Conference on Computer-Aided Verification (CAV)*, pages 274–287, Trento, Italy, 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 the 16th IEEE International Conference on Automated Software Engineering (ASE)*, pages 354–358, San Diego, USA, 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 the 10th IEEE International Conference on Requirements Engineering (RE)*, pages 282–288, Essen, Germany, 2002.

6. P. Godefroid, M. Huth, and R. Jagadeesan. Abstraction-based model checking using modal transition systems. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR)*, pages 426–440, Aalborg, Denmark, 2001.
7. A. Gurfinkel and M. Chechik. Generating counterexamples for multi-valued model-checking. In *Proceedings of the 12th International Symposium on Formal Methods (FME)*, pages 503–521, Pisa, Italy, 2003.
8. T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV)*, pages 440–451, Vancouver, Canada, 1998.
9. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, Jyväskylä, Finland, 1997.
10. S. Liu. Capturing complete and accurate requirements by refinement. In *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 57–67, Maryland, USA, 2002.
11. M. López-Nores and J. J. Pazos-Arias. A Formal Approach to Component-based Specification with Improved Requirements Traceability. In *Proceedings of RE'04 Doctoral Symposium*, Kyoto, Japan, 2004.
12. J. Moffett. A model for a causal logic for requirements engineering. *Journal of Requirements Engineering*, 1:27–46, 1996.
13. B. Nuseibeh. Crosscutting requirements. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 3–4, Lancaster, UK, 2004.
14. 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.
15. M. Poppleton and R. Banach. Retrenchment: Extending the reach of refinement. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 158–165, Florida, USA, 1999.
16. A. Rashid, P. Sawyer, A. Moreira, and J. Araújo. Early aspects: A model for aspect-oriented requirements engineering. In *Proceedings of the 10th IEEE International Conference on Requirements Engineering (RE)*, pages 199–202, Essen, Germany, 2002.
17. S. Schneider. *The B method: An introduction*. Palgrave, 2001.
18. P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 107–119, Los Angeles, USA, 1999.
19. A. van Lamsweerde. *The future of software engineering*, chapter Formal specification: A roadmap. ACM Press, 2000.
20. A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 5–19, Limerick, Ireland, 2000.

A Selective Parallel Composition of MUS Graphs

The *selective parallel composition* operator ($||[\Lambda]||$) has been typically defined over *Labeled Transition Systems*. For example, in Fig. 8(a), the overall system $\mathcal{L}_3 = \mathcal{L}_1 \,||[c]||\, \mathcal{L}_2$ can evolve through actions a and b in any order, because they are not in the Λ set. In contrast, c must be executed by the two individual processes simultaneously, which

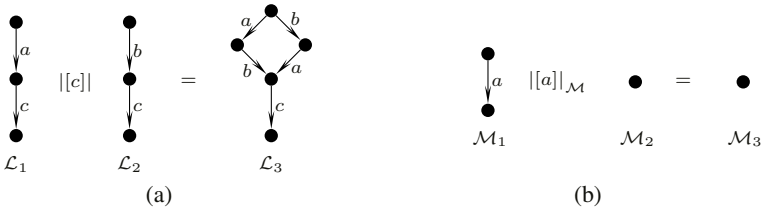


Fig. 8. Parallel composition of LTS and MUS graphs

guarantees that they terminate at the same time. Note that, as LTSs only handle *true* and *false* actions, the actions not explicitly represented are *false*.

Based on the $|[\Lambda]|$ operator, we define $|[\Lambda]|_{\mathcal{M}}$ by adapting its semantics to deal with *unspecification*. In a composition $\mathcal{M}_1|[\Lambda]|_{\mathcal{M}}\mathcal{M}_2$, the specification value for an action a is computed as the *minimum* between its values in $\mathcal{M}_1$ and $\mathcal{M}_2$ if $a \in \Lambda$, and as the *maximum* otherwise (note that $0 \leq \frac{1}{2} \leq 1$). Intuitively, if $a \notin \Lambda$, the two components need no agreement to go on, so it is possible to evolve through a if any of the two components can. On the contrary, if $a \in \Lambda$, the two components have to agree on a. If a is *false* in $\mathcal{M}_2$ and *true* in $\mathcal{M}_1$, the agreement is not possible, but if a were *unspecified* in $\mathcal{M}_2$ (as in Fig. 8(b)), the agreement would be possible if a evolved into *true* in a later iteration of the specification process. This way, the definition of $|[\Lambda]|_{\mathcal{M}}$ preserves the incompleteness of the components in the composition.

B Synthesis of the Synchronizer Component

This appendix outlines the algorithm used in Sect. 4 to derive the $Synchronizer^n$ component. This algorithm is applicable whenever the new requirements suppress behavior in the MUS graph of the composition, by turning *true* or *unspecified* actions into *false* ones. For simplicity, we illustrate the steps for the $n = 2$ case.

1. The first step is to turn *false* actions in the starting MUS graph into *unspecified* ones, to distinguish the actions forbidden by the specification of the components from the actions that will be prohibited by the new requirements. In the example, this returns the original MUS graph $\mathcal{M}_{S^2}$ (Figure 4(b)) because it had no *false* actions.
2. Next, the new requirements are materialized over the composition, by turning into *false* all the actions they forbid. As explained in Sect. 4, the new requirements of Eq. 2 lead to the MUS graph of Fig. 5.
3. The MUS graph resulting from the previous step is reduced, to remove the actions whose occurrence is irrelevant for the integration properties. This is done by successively merging contiguous *compatible states*, that is, states in which the specification values of the actions are not contradictory (what is *true* in the one is not *false* in the other) – remark that no reduction would be possible without the management of *unspecification*.
   In the MUS graph of Fig. 5, it is possible, for instance, to merge state s_1 with s_4, s_2 with s_5 and s_3 with s_6, which yields the reduced graph of Fig. 9(a). The process eventually produces the graph of Fig. 9(b).

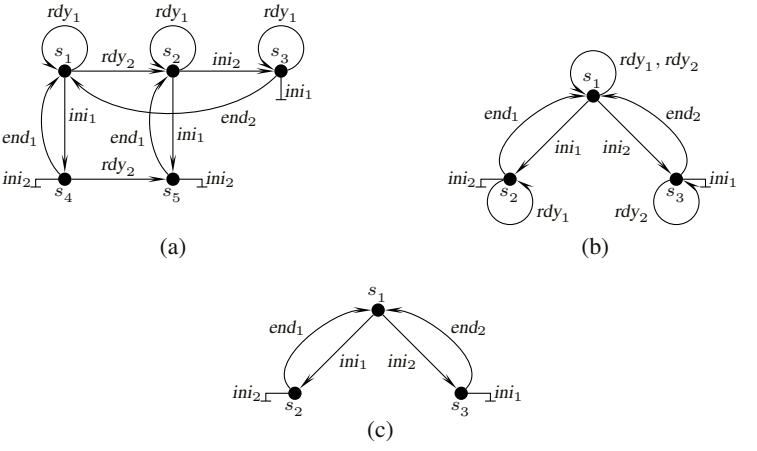


Fig. 9. Synthesis of the synchronizer component

4. To finish, it only remains to turn into *unspecified* the actions that are either *unspecified* or forming unitary loops in all the states – those actions are not relevant at the composition level. Then, the Λ_{sync} set of actions is formed by those actions which are *true* or *false* in any state. In the graph of Fig. 9(b), both rdy_1 and rdy_2 can be turned into *unspecified* actions in all the states, leading to the final component $Synchronizer^2$, shown in Fig. 9(c). The Λ^2_{sync} set is found to be $\{ini_1, ini_2, end_1, end_2\}$.

C Projection of the Synchronizer Aspect (←*)

We briefly describe here how to derive the modified components in the second architectural viewpoint of Sect. 4 by projecting the synchronizer aspect onto the original ones. As shown in Fig. 10, this is achieved by operating the original $Sender_i$ with the $Synchronizer^n$ component using the $||[\Lambda]||_{\mathcal{M}}$ operator, with Λ computed as the intersection of Λ^n_{sync} with the alphabet of actions of each $Sender_i$. This process is applicable whenever the original components are combined with $||[\Lambda]||_{\mathcal{M}}$, for any Λ.

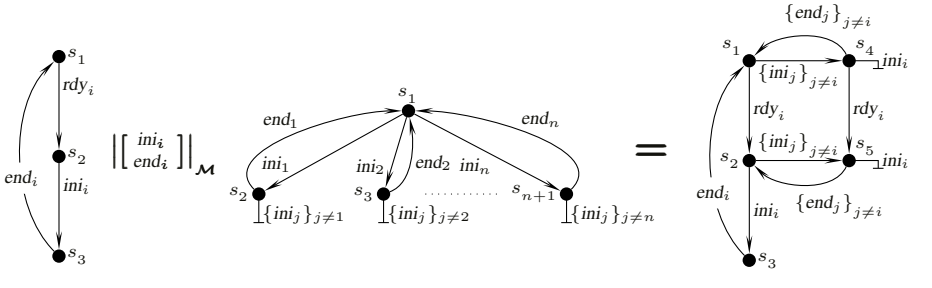


Fig. 10. Projection of the synchronizer aspect