Supporting Software Variability by Reusing Generic Incomplete Models at the Requirements Specification Stage[⋆]

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

Departament of Telematic Engineering. University of Vigo.
36200, Vigo. Spain. Fax number: 34 986 812116
{rebeca,mlnores,jose,avilas,jgd,agil,belen,mramos}@det.uvigo.es

Abstract. Selecting components that satisfy a given set of requirements is a key problem in software reuse, especially in reusing between different domains of functionality. This concern has been treated in the ARIFS methodology, which provides an environment to reuse partial and formal requirements specifications, managing the variability implicit in their incompleteness. In this paper, we define *generic incomplete specifications*, to introduce an explicit source of variability that allows reusing models across different domains, accommodating them to operate in multiple contexts. An extended formal basis is defined to deal with these tasks, that entails improvements in the reuse environment.

1 Introduction

Reusability has been widely suggested to be a key to improve software development productivity and quality, especially if reuse is tackled at early stages of the life cycle. However, while many references in the literature focus on reusing at late stages (basically code), there is little evidence to suggest that reusing at early ones is widely practiced. The ARIFS methodology [1, 2] (A*pproximate* R*etrieval of* I*ncomplete and* F*ormal* S*pecifications*) deals precisely with this concern, providing a suitable framework for reusing formal requirements specifications. It combines the well-known advantages of reusing at the requirements specification stage with the benefits of a formal approach, avoiding ambiguity problems derived from natural language.

In an incremental software process, the elements found at intermediate stages are characterized by their *incompleteness*. ARIFS is involved with a formal treatment of the *variability* inherent to this incompleteness. It covers the prospects of vertical reuse, i.e., reuse within the same domain of functionality. In this paper, we go one step further, to fully comply with the usual definition of software variability as "*the ability of a software artifact to be changed or customized to be used in multiple contexts*" [12]. Our proposal is to support an explicit form of variability that allows reusing models across different domains. We do this by defining *generic formal requirements specifications* and extending ARIFS' formal basis to classify and retrieve them.

[⋆] Partially supported by PGIDT01PX132203PR project (Xunta de Galicia)

The paper is organized as follows. Sections 2 and 3 describe the ARIFS reuse environment and the life cycle where it is applied. Section 4 introduces generic components and defines the basis for their classification and retrieval. Section 5 includes a simple example showing the advantages of the new proposal, compared to the original reuse environment. Section 6 comments other relevant works on the paper's scope and describes our future lines of work. A brief summary is finally given in Section 7.

2 The SCTL-MUS Methodology

SCTL-MUS [7] is a formal and incremental methodology for the development of distributed reactive systems, whose life cycle (Fig. 1) captures the usual way in which developers are given the specification of a system: starting from a rough idea of the desired functionality, this is successively refined until the specification is complete.

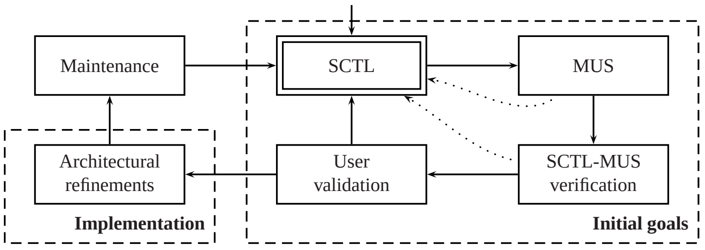


Fig. 1. The SCTL-MUS life cycle

Specifications are elaborated at the "*Initial goals*" stage. The requirements stated up to any given moment (in box *"SCTL"*) are used to synthesize a model or prototype (in box *"MUS"*). When it is ready, a model checker verifies the global objectives (*properties*) of the system ("*SCTL-MUS verification*") to find if the model satisfies them; if it cannot satisfy them, neither in the current iteration nor in future ones (*inconsistency*); or if it does not satisfy them, but it may in future iterations (*incompleteness*). In case of inconsistencies, the user is given suggestions to solve them. Then, by animating the MUS prototype ("*User validation*"), the user can decide whether the current specification is already complete or more iterations are needed. Upon completion, the system enters the *"Implementation"* stage. Here, the prototype is translated into the LOTOS process algebra [5] to obtain an initial architecture, that is progressively refined until allowing its semi-automatic translation into code language.

SCTL-MUS combines three formal description techniques. First, the many-valued logic SCTL (S*imple* C*ausal* T*emporal* L*ogic*) is used to express the functional requirements of a system, following the pattern *Premise* $\Rightarrow\otimes$ *Consequence*. Depending on the temporal operator, the consequence is assessed on the states where the premise is defined ($\Rightarrow$ or *"simultaneously"*), their predecessors ($\Rightarrow\odot$ or *"previously"*) or their successors ($\Rightarrow\bigcirc$ or *"next"*). Second, the graph formalism MUS (M*odel of* U*nspecified* S*tates*) is a variation of traditional labeled transitions systems (LTS), used to obtain system prototypes in terms of states and event-triggered transitions. Finally, the LOTOS process algebra is used to express the architecture of the developed system.

In an incremental specification process, it is essential to differentiate functional features that have been specified to be *false* (impossible) from those about which nothing has been said yet. SCTL introduces this concept of *unspecification* by adding a third value to the logic: an event can be specified to be *true* (1) or *false* (0), being *not-yet-specified* ($\frac{1}{2}$) by default. Analogously, MUS graphs support unspecification in both states and events, thus being adequate to model incomplete specifications.

Unspecification entails that intermediate models have freedom degrees, so that they can evolve into potentially many systems. Therefore, unspecification implies variability. The incremental specification process makes the system under development lose unspecification at each iteration, by evolving *not-yet-specified* elements into *true* or *false* ones, to eventually become the desired system. ARIFS was defined to take advantage of this implicit form of variability for the purposes of reuse.

3 The ARIFS Reuse Environment

ARIFS provides for the classification, retrieval and adaptation of reusable components in SCTL-MUS. Its objectives are twofold: (i) to save specification and synthesis efforts, by reusing suitable incomplete specifications; and (ii) to reduce the extensive resources needed to check (at every single iteration) the consistency of medium to large specifications, by reusing previously obtained formal verification results.

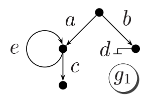
For these purposes, a reusable component is defined by: (a) its functional specification, expressed by a set of SCTL requirements and modeled by the corresponding MUS graph; (b) verification information summarizing the levels of satisfaction of the properties that have been verified on it; and (c) an interface or *profile*, used for classification and retrieval, that is automatically extracted from the functional specification.

Classifying Reusable Components The idea behind classification is that "*the closer two reusable components are classified, the more functional similarities they have*". According to this, we have defined four criteria to identify semantic and structural similarities [2]. This section describes those relevant for this paper: the TC^{∞} and NE^{∞} functions, whose results are included in the profile of the reusable components.

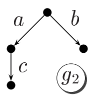
The TC^{∞} function offers a *semantic* viewpoint of reusable components. It associates with every MUS graph $g \in \mathbb{G}$ a set $TC^{\infty}(g)$ that contains sequences of events linked to its evolution paths. It follows the traditional complete-traces semantics [11], although it also reflects non-finite evolution paths and considers both *true* and *false* events, to differentiate these from *not-yet-specified* ones. On the other hand, NE^{∞} offers a *structural* viewpoint: given a MUS graph $g \in \mathbb{G}$, it returns a set $NE^{\infty}(g)$ reflecting the number of transitions that the model makes through every evolution path.

For each $\mathcal{O} \in \{TC^{\infty}, NE^{\infty}\}$, an equivalence relation $=_{\mathcal{O}} \in \mathbb{G} \times \mathbb{G}$ is defined, such that $g =_{\mathcal{O}} g' \Leftrightarrow \mathcal{O}(g) = \mathcal{O}(g')$. This organizes reusable components into equivalence classes of components indistinguishable using $\mathcal{O}$-observations. There is also a preorder relation $\sqsubseteq_{\mathcal{O}} \in \mathbb{G} \times \mathbb{G}$, given by $g \sqsubseteq_{\mathcal{O}} g' \Leftrightarrow \mathcal{O}(g) \sqsubseteq \mathcal{O}(g')$, that establishes a partial order between equivalence classes, so that $(\mathbb{G}, \sqsubseteq_{\mathcal{O}})$ is a *partially ordered set*.

Example 1. The following figure shows the result of applying the TC^{∞} and NE^{∞} functions to a MUS graph, g_1, that has four different evolution paths: (i) it can evolve through event b to a final state where d is not possible; (ii) it can reach another final state through events a and c; (iii) it can enter an endless loop through event a and an infinite number of events e; and (iv), it can reach a final state through event a, any number of events e and then event c.



$$TC^{\infty}(g_1) = (b\neg d, ac, a(e)+, a(e)+c)$$
$$NE^{\infty}(g_1) = (1, 2, (2)+, (3)+)$$



$$TC^{\infty}(g_2) = (ac, b)$$
$$NE^{\infty}(g_2) = (1, 2)$$

All these possibilities are reflected in $TC^{\infty}(g_1)$, where the $()+$ notation means that the sequences of events inside the parenthesis can be repeated any number of times. From the NE^{∞} point of view, for the evolution paths enumerated above, the system makes one, two, at least two and at least three transitions, respectively. On another hand, it is easy to see that the MUS graph g_2 is NE^{∞}- and TC^{∞}-included in g_1. □

The relationships among MUS graphs extrapolate directly to the corresponding reusable components. So, they allow organizing a repository of reusable components in two different lattices: the NE^{∞} lattice, intended for structural similarities (horizontal reuse) and the TC^{∞} lattice, for semantic ones (vertical reuse).

Retrieving Reusable Components The variability commented at the end of Sect. 2 allows the retrieval process to be based on functional proximity instead of on functional equivalence. Taking this into account, ARIFS performs *approximate retrievals*. Queries represent functional patterns of the SCTL statements that describe the system being developed. Actually, they are defined in the same terms as the profiles of the reusable components (NE^{∞} and TC^{∞} patterns), so the equivalence and partial orderings defined above also hold between queries and reusable components.

For efficiency reasons, the retrieval process is split in two steps. In the first phase, the adjacent components of the query in the NE^{∞} and TC^{∞} lattices are located. In the second, the search is refined by quantifying the functional differences between those components and the query, in order to minimize the adaptation efforts. In the case of NE^{∞}, differences are measured in terms of a numerical distance. As for TC^{∞}, two functions assess semantic differences: the *functional consensus* and the *functional adaptation*. A detailed description of these aspects can be found at [2].

Finally, the user is given the choice between the component closest to the query according to NE^{∞} criterion, and the closest according to TC^{∞}. This selection cannot be fully automated because the adaptation cost is expressed differently in each case.

Reusing Verification Information The idea behind the reuse of verification information in ARIFS is that useful information about the system being developed can be deduced from reusable components that are functionally close to it. With this aim, each reusable component stores verification results about every property that has been verified on it. We have proved that, for any two TC^{∞}-related components, interesting verification information can be extracted from one to the other, helping to reduce the amount of formal verification tasks throughout the specification process. The results of this work are summarized in [1].

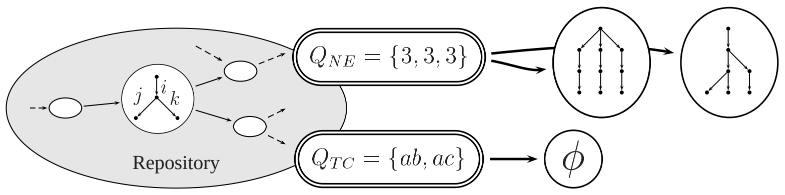


Fig. 2. Limitations of the NE^∞ and TC^∞ criteria

4 Defining Generic Components

Practical experience with ARIFS has revealed some features that can be improved. As shown in Fig. 2, an NE^∞-based retrieval may return wildly different components, hard to adapt to the desired functionality despite being NE^∞-equivalent to the query.

On the other hand, the TC^∞ criterion can only return components whose transitions are labeled the same way as the query. Thus, in the situation depicted in Fig. 2, the TC^∞ search using the pattern $\{ab, ac\}$ would not find any suitable component. However, it is easy to notice that one of the components in the repository would be TC^∞-equivalent to the query under the mapping $(a \rightarrow i, b \rightarrow j, c \rightarrow k)$. What is more, any property R' verified on that component gives the same verification results as R on the current specification, where R' is obtained from R by the same mapping. So, the verification information linked to that component is potentially useful for the system being developed. However, it can not be reused with TC^∞, because this criterion is too linked to the domain of functionality of every particular component.

Generic components are introduced to address these problems. They have the same information as classic components (Section 3), but with the functionality and the verified properties expressed in terms of *meta-events*. Meta-events are identifiers such that different meta-events of a generic component must represent different events. To deal with them, we introduce a new criterion: MT^∞.

MT^∞ associates with every MUS graph $g \in \mathbb{G}$ a set $MT^\infty(g)$ that contains sequences of meta-events linked to its evolution paths. Two graphs g and g' are MT^∞-equivalent ($g =^\infty_{\mathrm{MT}} g'$) iff a one-to-one mapping between the actions of g and g' exists such that, having done the mapping, $g^{map} =^\infty_{\mathrm{TC}} g'$. Analogously, a graph g is MT^∞-included in another graph g' ($g \sqsubseteq^\infty_{\mathrm{MT}} g'$) iff all the actions of g can be mapped to a different action of g' in a way that, having done the mapping, $g^{map} \sqsubseteq^\infty_{\mathrm{TC}} g'$ (see Fig. 3).

An ordering exists (Eq. (1)) such that, if two components are TC^∞-related, they are MT^∞- and NE^∞-related in the same way: $C_i \sqsubseteq^\infty_{\mathrm{TC}} C_j \Rightarrow C_i \sqsubseteq^\infty_{\mathrm{MT}} C_j \Rightarrow C_i \sqsubseteq^\infty_{\mathrm{NE}} C_j$.

$$NE^\infty \preceq MT^\infty \preceq TC^\infty \tag{1}$$

Our proposal in this paper is to organize the repository in a single lattice of generic components, using the MT^∞ relations. MT^∞ merges structural and semantic viewpoints in a convenient way: it is much less permissive than NE^∞ identifying structural similarities, and abstracts the domain of functionality by considering generic actions. This introduces an explicit form of variability in the alphabet of actions of a reusable component. We also propose MT^∞ to be the only criterion to conduct the retrieval process, automating the decision of which component to reuse; and the profile of reusable components to be formed by the results of the NE^∞ and MT^∞ functions.

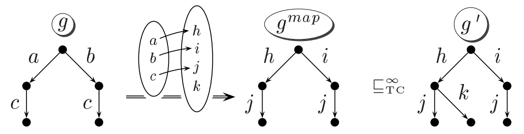


Fig. 3. g is MT^{∞}-included in g'

5 A Simple Case Study on Applying MT^{∞}

This section includes an example to show the advantages of the MT^{∞} criterion compared to TC^{∞}. The bulk of the section deals with the efforts needed to adapt the retrieved components, so we pay no attention to the NE^{∞} criterion, which entails greater efforts in the general case (see Sect. 4). In line with the motivation of MT^{∞}, we consider a repository with reusable components from two different domains of functionality: communication protocols and automata for the control of elevators.

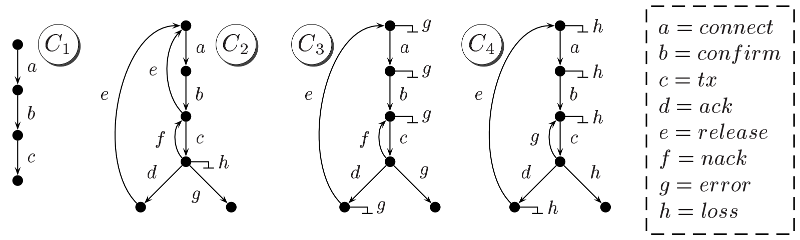


Fig. 4. Components obtained from the specification of stop-and-wait sender processes

The components in Fig. 4 were obtained at intermediate specification phases of stop-and-wait senders. Events $connect$, $confirm$ and $release$ serve to model connections between sender and receiver; after the sender transmits a data frame (action tx), the receiver may acknowledge it positively (ack) or negatively ($nack$); finally, events $error$ and $loss$ can be used to model disruptive behavior of the channel.

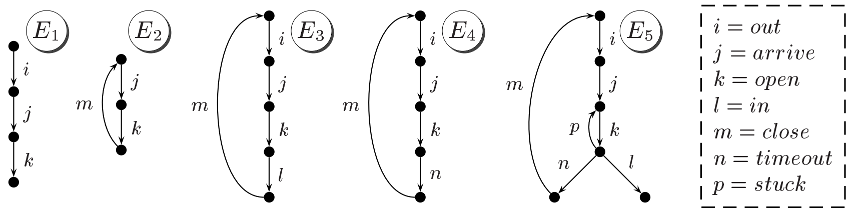


Fig. 5. Components obtained from the specification of elevator automata

Figure 5 shows the elevator automata components. The buttons inside and outside the elevator issue *in* and *out* events; the door is modeled with *open*, *close* and *stuck*; event *arrive* occurs when the elevator gets to the requested floor; finally, *timeout* limits how long the elevator may stay in a given floor before serving other requests.

Suppose that an user is working on the specification of a new stop-and-wait sender, which currently consists of the SCTL requirements shown in Eq. (2) (the actions have been relabeled as in Fig. 4). This partial specification, $Spec$, will serve to compare the performance of the TC^∞ and MT^∞ criteria when it comes to avoid synthesis tasks by adapting a reusable component.

$$Spec = \begin{cases} R_1 \equiv a \Rightarrow\bigcirc\ (b \Rightarrow\bigcirc\ (R_2\ \wedge\ R_3)) \\ R_2 \equiv c \Rightarrow\bigcirc\ (d \Rightarrow\bigcirc\ (e \Rightarrow\bigcirc\ R_1)) \\ R_3 \equiv c \Rightarrow\bigcirc\ (f \Rightarrow\bigcirc\ (R_2\ \wedge\ R_3)) \end{cases} \tag{2}$$

$$Q = TC^\infty(Spec) = ((abcde)+, ab(cf)+, (ab(cf+)de)+) \tag{3}$$

The TC^∞ Retrieval The first step of the TC^∞ retrieval is to place the query in the TC^∞ lattice, by applying the TC^∞ criterion. Then, the second step scrutinizes the adjacent components to return those which minimize the adaptation efforts to the query. In our example, the query is the TC^∞ pattern of $Spec$, shown in Eq. (3).

Figure 6 shows how the repository is organized by applying the TC^∞ criterion. It can be seen that components from different domains form separate sub-lattices, even though there exists strong resemblance between some of them. The first step of the retrieval places Q between components C_1, C_2 and C_3 (see Fig. 6); the second selects C_2, because it has the lowest adaptation efforts, as measured by the *functional consensus* and *functional adaptation* functions (Section 3).

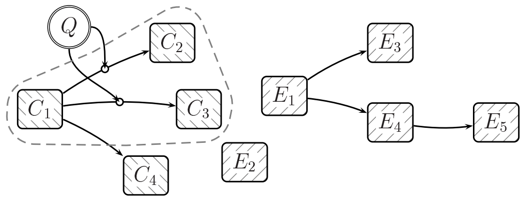


Fig. 6. Placing the query in the TC^∞ lattice

The MT^∞ Retrieval In the MT^∞ retrieval, the query is again the TC^∞ pattern of the specification (Eq. (3)), but it is placed in the MT^∞ lattice by applying the MT^∞ criterion. In this lattice, the different sub-lattices of Fig. 6 become interlaced, because MT^∞ allows relationships between components from different domains. Moreover, it finds several components to be equivalent, and therefore merges them into a single generic one (for instance, GCE_{11} stands for C_1 and E_1). As a result, the size of the repository is reduced from nine components to six.

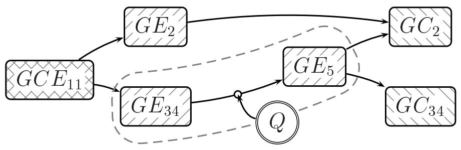


Fig. 7. Placing the query in the MT^∞ lattice

Now, the first step of retrieval returns components GE_{34} and GE_5 (see Fig. 7), and GE_5 is finally selected, according to the *functional consensus* and *functional adaptation* functions (Section 3), which are straightforward to redefine in terms of meta-events.

Comparing the Two Criteria Knowing what components the TC^∞ and MT^∞ criteria retrieve, we can look at the adaptation efforts needed in each case to match the desired functionality. Figure 8 shows components C_2 and GE_5, with their functional adaptation regarding Q emphasized in gray (meta-events are represented by Greek letters). In this case, the desired component C_Q is just the result of eliminating the gray parts of both graphs. Clearly, GE_5 (the component retrieved by MT^∞) has less functional excess than C_2 (the one retrieved by TC^∞), so its adaptation is easier.

In fact, the components retrieved by MT^∞ can never be harder to adapt than those retrieved by TC^∞. As a result of Eq. (1), TC^∞ relationships are kept in the MT^∞ repository, so, in the worst case, the MT^∞ retrieval returns the same components as the TC^∞ retrieval. Since components from different domains get interlaced in the MT^∞ lattice, it increases the chances to retrieve functionally closer elements, which are therefore easier to adapt. So, the new source of variability introduced by the use of generic components (Section 4) turns out to be beneficial.

Reusing Verification Results The fact that TC^∞ organizes components from different domains in separate sub-lattices prevents reuse across domains. The interlacing due to MT^∞ allows sharing verification information between different domains, increasing the amount of information retrievable for any given query.

In our example, reusing component GE_5 entails recovering all the verification results of properties not affected by the adaptation process. For instance, assume that one of the properties previously verified on E_5 is $close \Rightarrow \odot\ timeout$ ("*the door of the elevator cannot close if it has not been opened for the length of a timeout*"). GE_5 stores it as the generic property $\eta \Rightarrow \odot\ \varepsilon$, which, under the mapping that relates the query to GE_5, becomes $release \Rightarrow \odot\ ack$ ("*a connection cannot be released without receiving an acknowledgment first*"). As the events involved are not affected by adaptation, the verification results obtained for E_5 hold directly for the retrieved component.

6 Discussion and Future Work

Approaches to managing collections of reusable components are usually based on the same idea: establishing a *profile* or a set of characterizing attributes as a representation of the components, and use it for classification and retrieval. In many cases, like in [4] and [8], they are based on natural language. Other works resort to formal methods to describe the relevant behavior of a component, thus minimizing the problems derived from natural language. In this case, the typical *specification matching* solutions rely on theorem provers, which requires a large number of formal proofs and makes a practical implementation very hard. Several techniques have been proposed to address this problem in the reuse of source code [9, 14]. In contrast, our proposal is intended to reuse formal components at early stages, and avoids formal proofs by defining an approximate retrieval process based on the incompleteness of intermediate models. Another

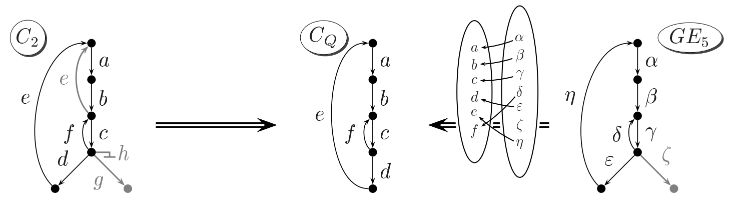


Fig. 8. Adapting the components retrieved by TC^{∞} and MT^{∞}

distinctive feature is that it performs reuse automatically, with no intervention needed from developers. This contributes to solve a major flaw of traditional repository systems, which would offer no help if developers made no attempt to reuse. In this sense, our work is in line with the *active repository system* of [13], although this is applied to code and based on natural language.

Section 2 describes how the concept of *unspecification*, characteristic of the SCTL-MUS methodology, relates to software variability. In this regard, we believe that managing a repository of reusable incomplete models is highly useful, as long as its organization reflects a hierarchy of variability: by choosing the right component to reuse, we are dealing with the right amount of variability, no more no less. As far as we know, our view is innovative in the area of formal specification approaches. Moreover, very little has been published about variability in life cycles other than the classical waterfall model. Only in [3] is it analyzed in the context of *Extreme Programming*, which, just like SCTL-MUS, defines an iterative life-cycle. The incremental approach of SCTL-MUS has two main advantages. First, as everything is *not-yet-specified* by default, variability is not limited to the variation points introduced intentionally by developers. Second, as we use the same software artifacts all through the development process, the techniques for variability realization are greatly simplified. Note that, in a waterfall cycle, different techniques must be used depending on the phase at which a variation point is introduced and the phase at which it is bound [10].

Most of the research on software variability has been done in the field of software product lines [12]. A product line defines an architecture shared by a range of products, together with a set of reusable components that, combined, make up a considerable part of the functionality of those products. The work presented here is not directly applicable in this area, because SCTL-MUS specifications lack architecture. However, we are currently working to extend this methodology following the principles of aspect-oriented programming (AOP) [6]. The mechanisms introduced in this paper still hold in the revised methodology, as long as compositions will be expressed in the same formalisms of the composed elements. In this context, we hope that the philosophy of the MT^{∞} criterion may be useful to detect *crosscutting functionality* —the *leit motif* of AOP—, because it can find common algorithmic patterns between several components, regardless of the particular names of the events.

7 Summary

This paper presents some enhancements to an existing framework for the reuse of formal requirements specifications. The main contribution is the definition of generic reusable components, which, by abstracting their original domain of functionality, allow reusing models and formal verification results across multiple application areas. This introduces an explicit source of variability in the alphabet of actions of the reusable components, that complements the implicit one inherent to incompleteness.

A new criterion, MT^∞, has been defined to organize generic components. It enhances the effectiveness of the retrieval process, by increasing the amount of information available to reuse, and by reducing the efforts needed to adapt the retrieved components to the desired functionality. It has also been experienced to cause efficiency improvements, because it simplifies the management of the repository (one only lattice, fewer components) and, being the only criterion to decide what component to reuse, it fully automates the retrieval. In this regard, the presumable greater cost of finding MT^∞ relationships, compared to TC^∞ or NE^∞ ones, is masked by the incremental approach, because we can start from the event mappings used in preceding iterations.

References

1. R. P. Díaz Redondo, J. J. Pazos Arias, and A. Fernández Vilas. Reuse of Formal Verification Efforts of Incomplete Models at the Requirements Specification Stage. *Lecture Notes in Computer Science*, 2693, 2003.
2. R. P. Díaz Redondo, J. J. Pazos Arias, A. Fernández Vilas, and B. Barragáns Martínez. ARIFS: an Environment for Incomplete and Formal Specifications Reuse. *Electronic Notes in Theorethical Computer Science*, 66(4), 2002.
3. P. Predonzani G. Succi and T. Vernazza. *Extreme Programming Examined*, chapter Tracing Development Progress: A Variability Perspective. Addison Wesley Professional, 2001.
4. M. R. Girardi and B. Ibrahim. Automatic Indexing of Software Artifacts. In *3rd International Conference on Software Reusability*, pages 24–32, Rio de Janeiro, Brazil, November 1994.
5. ISO. *LOTOS – A Formal Description Technique Based on an Extended State Transition Model*. ISO/IEC/8807, International Standards Organization, 1989.
6. G. Kiczales, J. Lamping, A. Mendhekar, and C. Maeda. Aspect-oriented Programming. In *Conference on Object-Oriented Programming (ECOOP)*, volume 1261 of *Lecture Notes in Computer Science (LNCS)*, pages 220–243, Jyväskylä, Finland, June 1997.
7. 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.
8. R. Prieto-Díaz. Implementing Faceted Classification for Software Reuse. *Communications of the ACM*, 34(5):88–97, May 1991.
9. J. Schumann and Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proc. of the 12th ASE*, pages 246–254, 1997.
10. M. Svahnberg, J. van Gurp, and J. Bosch. A Taxonomy of Variability Realization Techniques. In *ISSN: 1103-1581*, Blekinge Institute of Technology, Sweden, 2002.
11. R. J. van Glabeek. *Handbook of Process Algebra*, chapter The Linear Time - Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. Elsevier Science, 2001.
12. J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In *2nd Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2001.
13. Y. Ye. An Active and Adaptive Reuse Repository System. In *Proceedings of 34th Hawaii International Conference on System Sciences (HICSS-34)*, 2001.
14. A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.