Towards Managing Dynamic Reconfiguration of Software Systems in a Categorical Setting

Pablo F. Castro[1], Nazareno M. Aguirre[1], Carlos G. López Pombo[2], and Thomas S.E. Maibaum[3]

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Río Cuarto, Córdoba, Argentina,
{pcastro, naguirre}@dc.exa.unrc.edu.ar
[2] Departamento de Computación, FCEyN, Universidad de Buenos Aires and CONICET, Buenos Aires, Argentina, clpombo@dc.uba.ar
[3] Department of Computing & Software, McMaster University, Hamilton (ON), Canada, tom@maibaum.org

Abstract. Dynamic reconfiguration, understood as the ability to manage at run time the live components and how these interact in a system, is a feature that is crucial in various languages and computing paradigms, in particular in object orientation. In this paper, we study a categorical approach for characterising dynamic reconfiguration in a logical specification language. The approach is based on the notion of *institution*, which enables us to work in an abstract, logic independent, setting. Furthermore, our formalisation makes use of *representation maps* in order to relate the generic specification of components (e.g., as specified through classes) to the behaviour of actual instances in a dynamic environment. We present the essential characteristics for dealing with dynamic reconfiguration in a logical specification language, indicating their technical and practical motivations. As a motivational example, we use a temporal logic, component based formalism, but the analysis is general enough to be applied to other logics. Moreover, the use of representation maps in the formalisation allows for the combination of different logics for different purposes in the specification. We illustrate the ideas with a simple specification of a Producer-Consumer component based system.

1 Introduction

Modularisation is a key mechanism for dealing with the complexity and size of software systems. It is generally understood as the process of dividing a system specification or implementation into *modules* or *components*, which leads to a structural view of systems, and systems' structure, or *architecture* [10]. Besides its crucial relevance for managing the complexity of systems, the systems' architectural structure also plays an important role in the functional and non functional characteristics of systems. The system architecture has traditionally been *static*, in the sense that it does not change at run time. However, many component based specifications or implementations require dealing with dynamic creation and deletion of components. This is the case, for instance, in *object oriented programming*, where the ability of creating and deleting *objects* dynamically, i.e., at run time, is an intrinsic characteristic. Also in other more abstract contexts, such as software architecture, it is often required to be able to dynamically reconfigure systems, involving in many cases the dynamic creation or deletion of components and connectors [17]. Also in some fields related to fault tolerance, such as self healing and self adaptive systems, it is often necessary to perform dynamic reconfigurations in order to take a system from an inconsistent state back to an acceptable configuration.

Category theory has been regarded as an adequate foundation for formally characterising different notions of components, and component compositions. For instance, in the context of algebraic specification, category theory has enabled the formal characterisation of different kinds of specification extensions [6] . Also, in the context of parallel program design, category theory has been employed for formalising the notion of *superposition*, and the synchronisation of components [7]. In this paper, we present a categorical characterisation of the elements of component composition necessary when dealing with dynamic creation and deletion of components. The characterisation is developed around the notion of an *institution*, which enables us to work in an abstract, logic independent, setting. Furthermore, our formalisation makes use of *representation maps* [22] in order to relate the generic specification of components (e.g., as specified through classes) to the behaviour of actual instances in a dynamic environment. The use of representation maps provides an additional advantage, namely that it allows for the combination of different logics for different purposes in the specification. For instance, one might use a logic for characterising datatypes (e.g., equational logic), another for specifying components (e.g., propositional LTL), and another (e.g., first order LTL) for the description of dynamically reconfigurable systems, involving these components and datatypes.

We present the essential characteristics for dealing with dynamic reconfiguration in a logical specification language, indicating their technical and practical motivations. The approach presented is motivated by the view of system composition as a colimit of a categorical diagram representing the system's structure [3]. Moreover, our approach, as presented in this paper (and in particular due to the logic employed for illustrating the ideas), can be seen as an adaptation of the ideas presented in Fiadeiro and Maibaum's approach to concurrent system specification [7], where the system's structure is inherently rigid, to support dynamic creation/deletion of components, and changes in their interactions. As a motivating example, we use a temporal logic, component based formalism, but the analysis is general enough to be applied to other logics. We will use a single logic for the different parts of a specification, although, as we mentioned, the approach enables one to use different logics for different purposes in specification, as it will be made clearer later on. One might benefit from this fact, in particular for analysis purposes, as it will be argued in the paper. We also discuss some related work. The main ideas presented in the paper are illustrated with a simple specification of a (dynamic) Producer-Consumer component based system.

Component: *Producer*
Read Variables: *ready-in*: Bool
Attributes: *p-current*: Bit, *p-waiting*: Bool
Actions: *produce-0*, *produce-1*, *send-0*, *send-1*, *p-init*
Axioms:
1. $\Box(\textit{p-init} \rightarrow \bigcirc(\textit{p-current} = 0 \wedge \neg \textit{p-waiting}))$
2. $\Box(\textit{produce-0} \vee \textit{produce-1} \rightarrow \neg \textit{p-waiting} \wedge \bigcirc \textit{p-waiting})$
3. $\Box(\textit{produce-0} \rightarrow \bigcirc(\textit{p-current} = 0))$
4. $\Box(\textit{produce-1} \rightarrow \bigcirc(\textit{p-current} = 1))$
5. $\Box((\textit{send-0} \rightarrow \textit{p-current} = \textit{0}) \wedge (\textit{send-1} \rightarrow \textit{p-current} = \textit{1}))$
6. $\Box(\textit{send-0} \vee \textit{send-1} \rightarrow \textit{p-waiting} \wedge \bigcirc \neg \textit{p-waiting})$
7. $\Box(\textit{send-0} \vee \textit{send-1} \rightarrow \textit{p-current} = \bigcirc \textit{p-current})$
8. $\Box(\textit{send-0} \vee \textit{send-1} \vee \textit{produce-0} \vee \textit{produce-1} \vee \textit{p-init} \vee$
$(\textit{p-current} = \bigcirc \textit{p-current} \wedge \textit{p-waiting} = \bigcirc \textit{p-waiting}))$

Fig. 1. A linear temporal logic specification of a simple producer.

2 A Motivating Example

In this section, we introduce an example that will be used as a motivation for the work presented in the paper. This example is a simple specification of a Producer-Consumer component based system. The specification is written in linear temporal logic. We assume the reader is familiar with first order logic and linear temporal logic, as well as some basic concepts from category theory [20].

Let us consider a simple Producer-Consumer system, in which two components interact. One of these is a producer, which produces messages (items) that are sent to the other component, the consumer. For simplicity, we assume that the messages communicated are simply bits. The producer's state might then be defined by a bit-typed field *p-current* to hold a produced element, a *boolean* variable *p-waiting* to indicate whether an item is already produced and ready to be sent (so that null values for items are not necessary), and a boolean read variable *ready-in*, so that a producer is informed if the environment is ready to receive a product. We can specify a producer axiomatically, as shown in Figure 1. This specification consists of a set of sorts (Bit and Bool, in this case), a set of fields, some of which are supposed to be controlled by the environment, and a set of action symbols. The axioms of the specification are linear temporal logic formulae characterising the behaviour of the component, in a rather obvious way. Notice Axiom 8, which differentiates local fields from read variables. This is a locality axiom, as in [7], a frame condition indicating that local fields can only be altered by local actions. The axioms of the specification can be thought of as originating in an action language, such as the SMV language, for instance. Notice that the logic used in this specification is *propositional*, which would enable one to algorithmically check properties of producers, by means of model checking tools. A consumer component can be specified in a similar way, as shown in Figure 2.

A mechanism for putting these specifications together is by coordinating them, for instance, by indicating how read variables are "connected" or identi-

Component: **Consumer**
Read Variables: *ready-ext*: Bool
Attributes: *c-current*: Bit, *c-waiting*: Bool
Actions: *consume*, *extract-0*, *extract-1*, *c-init*
Axioms:
1. $\Box(\textit{c-init} \rightarrow \bigcirc(\textit{c-current} = 0 \land \textit{c-waiting}))$
2. $\Box(\textit{extract-0} \lor \textit{extract-1} \rightarrow \neg\textit{c-waiting} \land \textit{c-waiting} \land \textit{ready-ext})$
3. $\Box(\textit{extract-0} \rightarrow \bigcirc(\textit{c-current} = 0))$
4. $\Box(\textit{extract-1} \rightarrow \bigcirc(\textit{c-current} = 1))$
5. $\Box(\textit{consume} \rightarrow \neg\textit{c-waiting} \land \bigcirc\textit{c-waiting})$
6. $\Box(\textit{consume} \rightarrow \textit{c-current} = \bigcirc\textit{c-current})$
7. $\Box(\textit{consume} \lor \textit{extract-0} \lor \textit{extract-1} \lor \textit{c-init} \lor$
$(\textit{c-current} = \bigcirc\textit{c-current} \land \textit{c-waiting} = \bigcirc\textit{c-waiting}))$

Fig. 2. A linear temporal logic specification of a simple consumer.

fied with fields of other components, and by synchronising actions. Basic action synchronisation can be employed for defining more sophisticated forms of interaction, e.g., procedure calls. In [7, 8], the described form of coordination between components is achieved by the use of "channels"; a channel is a specification with no axioms, but only symbol declarations, together with two mappings, identifying the symbols in the channel specification with the actions to be synchronised and the fields/variables to be identified, in the corresponding components. For our example, we would want to make the components interact by synchronising the `send-i` and `extract-i` actions, of the producer and consumer, respectively, and by identifying `ready-in` and `p-waiting`, in the producer, with `c-waiting` and `ready-ext` in the consumer, respectively.

It is known that specifications and symbol mappings in this logic form a category which admits finite colimits [12]. This is important due to the fact that the above described coordination between producers and consumers, materialised as a "channel", forms the following *diagram* (in the categorical sense):

$$\begin{tikzcd}
\textbf{Producer} & & \textbf{Consumer} \\
& {[\mathbf{vars}:\mathbf{x},\mathbf{y}:\mathbf{Bool};\mathbf{acts}:\mathbf{a},\mathbf{b}]} \arrow[ul, "\begin{array}{l} x \mapsto \textit{ready-in} \\ y \mapsto \textit{p-waiting} \\ a \mapsto \textit{send-0} \\ b \mapsto \textit{send-1} \end{array}"] \arrow[ur, "\begin{array}{l} x \rightarrow \textit{c-waiting} \\ y \rightarrow \textit{ready-ext} \\ a \rightarrow \textit{extract-0} \\ b \rightarrow \textit{extract-1} \end{array}"'] &
\end{tikzcd}$$

The colimit object for this diagram is a specification that corresponds to the combined behaviour of the producer and consumer, interacting as the diagram indicates.

The architecture of the system, represented by the diagram, clearly does not directly admit reconfiguration. More precisely, how components are put together is prescribed in an external way with respect to component definition (by the construction of a diagram), and although the represented specification can be constructed as a colimit, the possibility of having a component managing the population of instances of other components (as an example of dynamic reconfiguration, motivated by what is common in object orientation) is not compatible with the way configurations are handled in this categorical approach.

Our aim in this paper is to provide a categorical characterisation of a generalisation of the above situation, when both the population of live components and their connections are manipulated, within a system, dynamically. The actual way in which these elements (components and connections) are dynamically manipulated depends on the particular problem or system being specified. For instance, we might have a system where the number of components is maintained over time, but the way in which these components interact is changed dynamically. Alternatively, we might have a system in which a certain kind of component, e.g., clients, are created dynamically, but there is a fixed number of servers.

3 Dynamic Reconfiguration in an Institutional Setting

In this section, we present our proposal for formally characterising dynamic reconfiguration in a logical specification language. In order to make the approach generic (i.e., to make it applicable to a wide range of logics and related formalisms), we develop the formalisation using the notion of *institution*. This enables us to present the formalisation in a high level, logic independent, setting. The theory of institutions was presented by Goguen and Burstall in [11]. Institutions provide a formal and generic definition of what a logical system is, and of how specifications in a logical system can be structured [21]. Institutions have evolved in a number of directions, from an abstract theory of software specification and development [25] to a very general version of abstract model theory [5], and offered a suitable formal framework for addressing heterogeneity [19, 24], including applications related to widely used (informal) languages, such as the UML [4]. The following definitions were taken from [18].

Definition 1. [Institution]
An institution *is a structure of the form* $\langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models^{\Sigma}\}_{\Sigma \in |\mathbf{Sign}|}\rangle$ *satisfying the following conditions:*

- $\mathbf{Sign}$ *is a category of signatures,*
- $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ *is a functor (let* $\Sigma \in |\mathbf{Sign}|$[1]*, then* $\mathbf{Sen}(\Sigma)$ *returns the set of* Σ*-sentences),*
- $\mathbf{Mod} : \mathbf{Sign}^{\mathsf{op}} \to \mathbf{Cat}$ *is a functor (let* $\Sigma \in |\mathbf{Sign}|$*, then* $\mathbf{Mod}(\Sigma)$ *returns the category of* Σ*-models),*
- $\{\models^{\Sigma}\}_{\Sigma \in |\mathbf{Sign}|}$*, where* $\models^{\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$*, is a family of binary relations,*

and for any signature morphism $\sigma : \Sigma \to \Sigma'$*,* Σ*-sentence* $\phi \in \mathbf{Sen}(\Sigma)$ *and* Σ'*-model* $\mathcal{M}' \in |\mathbf{Mod}(\Sigma)|$ *the following* $\models$*-invariance condition holds:*

$$\mathcal{M}' \models^{\Sigma'} \mathbf{Sen}(\sigma)(\phi) \quad \textit{iff} \quad \mathbf{Mod}(\sigma^{\mathsf{op}})(\mathcal{M}') \models^{\Sigma} \phi \ .$$

[1] $|\mathbf{Sign}|$ denotes the class of objects of category $\mathbf{Sign}$.

Institutions are an abstract formulation of the notion of logical system where the concepts of languages, models and truth are characterised using category theory. Roughly speaking, an institution is made up of a category **Sign** which defines the syntax of the logic in terms of the possible vocabularies and translations between them, a functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ that captures the way in which formulae are built from vocabularies (this functor maps translations between vocabularies to translations between sets of formulae in the obvious way). Moreover, the semantical part of a given logic is captured using a covariant functor $\mathbf{Mod} : \mathbf{Sign}^{\mathsf{op}} \to \mathbf{Cat}$ which maps each vocabulary to the category of its possible models. This functor is covariant since any translation of symbols uniquely determines a model reduct. Finally, an indexed relation $\models^{\Sigma}: \mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$ is used to capture the notion of truth. A restriction is imposed on this relationship to ensure that truth is not affected by change of notation. Examples of institutions are: propositional logic, equational logic, first-order logic, first-order logic with equality, dynamic logics and temporal logics (a detailed list is given in [12]). Note that any of these logics has the four components of institutions. Furthermore, in these logics the notion of truth does not depend on the particular choice of the symbols in a formula, i.e., the truth of a formula depends only on its structure and not on the contingent names of its parts.

The logic we used for specifying components, linear temporal logic, constitutes an institution. Its category of signatures is composed of alphabets (sets of propositional variables, since bit-typed fields are straightforwardly encoded as boolean variables, and action occurrence can directly be represented as boolean variables) as objects, and mappings between alphabets as morphisms. The grammar functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ for this logic is simply the recursive definition of formulae for a given vocabulary. The functor $\mathbf{Mod} : \mathbf{Sign}^{\mathsf{op}} \to \mathbf{Cat}$ maps signatures (alphabets) to their corresponding classes of models, and alphabet contractions (i.e., reversed alphabet translations) to "reducts". The relationship $\models^{\Sigma}$ is the usual satisfaction relation in LTL. By means of a simple inductive argument, it is rather straightforward to prove that this relationship satisfies the invariance condition, and thus LTL is an institution.

The logic we used so far is propositional. A first-order version of this logic is presented in [16], where variables, function symbols and predicate symbols are incorporated, as usual. This first order linear temporal logic admits a single type of "flexible" (i.e., whose interpretation is state dependent) symbol, namely flexible variables. All other symbols (function and predicate symbols, in particular) are rigid, in the sense that their interpretations are state independent. We will consider a generalisation of this logic, in which function and relation symbols are "split" into flexible and rigid (notice that flexible variables become a special case of flexible function symbols). This will simplify our specifications, and the presentation of the ideas in this paper. The propositional specifications given before can be thought of as first-order specifications, where the "first-order" elements of the language are not used. By employing the ideas presented in [21], we can prove in a straightforward way that this first-order linear temporal logic is also an institution.

Component: *ProducerManager*
Read Variables: *ready-in* : NAME $\rightarrow$ Bool
Attributes: *p-current* : NAME $\rightarrow$ Bit,
p-waiting : NAME $\rightarrow$ Bool
Actions: *produce-0*(n: NAME), *produce-1*(n: NAME), *send-0*(n: NAME),
send-1(n: NAME), *p-init*(n: NAME)
Axioms:
1. $\Box(\forall n \in \text{NAME} : \textit{p-init}(n) \rightarrow \bigcirc(\textit{p-current}(n) = 0 \wedge \neg \textit{p-waiting}(n)))$
2. $\Box((\forall n \in \text{NAME} : \textit{produce-0}(n) \vee \textit{produce-1}(n) \rightarrow \neg \textit{p-waiting}(n) \wedge \bigcirc \textit{p-waiting}(n))$
3. $\Box(\forall n \in \text{NAME} : \textit{produce-0}(n) \rightarrow \bigcirc(\textit{p-current}(n) = 0))$
4. $\Box(\forall n \in \text{NAME} : \textit{produce-1}(n) \rightarrow \bigcirc(\textit{p-current}(n) = 1))$
5. $\Box(\forall n \in \text{NAME} : (\textit{send-0}(n) \rightarrow \textit{p-current(n) = 0}) \wedge (\textit{send-1}(n) \rightarrow \textit{p-current(n) = 1}))$
6. $\Box(\forall n \in \text{NAME} : \textit{send-0}(n) \vee \textit{send-1}(n) \rightarrow \textit{p-waiting}(n) \wedge \bigcirc \neg \textit{p-waiting}(n))$
7. $\Box(\forall n \in \text{NAME} : \textit{send-0}(n) \vee \textit{send-1}(n) \rightarrow \textit{p-current}(n) = \bigcirc \textit{p-current}(n))$
8. $\Box(\forall n \in \text{NAME} : \textit{send-0}(n) \vee \textit{send-1}(n) \vee \textit{produce-0}(n) \vee \textit{produce-1}(n) \vee \textit{p-init}(n) \vee (\textit{p-current}(n) = \bigcirc \textit{p-current}(n) \wedge \textit{p-waiting}(n) = \bigcirc \textit{p-waiting}(n)))$

Fig. 3. A first-order linear temporal logic specification of a producer manager.

A specification is essentially a theory presentation, as usually defined [12, 18]. Any category of alphabets and translations can be lifted to categories **Th** and **Pres**, of theories and theory presentations, where morphisms are theorem and axiom preserving translations, respectively [9]. The relationships between these categories and **Sign** are materialised as forgetful functors (which reflect colimits).

A traditional way of dealing with dynamic reconfiguration is by specifying *managers* of components. A manager of a component C is a specification which intuitively provides the behaviour of various instances of C, and usually enables the manipulation of instances of C. For example, for our Producer specification, a manager might look as in Figure 3. Notice that we are using the "first-order" expressive power of the language in this specification.

Notice the clear relationship between our producer specification and the specification of a manager of producers. With respect to the syntax (i.e., the symbols used in the specification), the manager is a *relativisation* of the producer, in which all variables and actions incorporate a new parameter, namely, the "instance" to which the variable belongs, or the action to which it is applied, correspondingly. A first intuition would be to try to characterise the relationship between the signature of a component and the signature of its manager as a signature morphism. However, such a relationship is not possible, since signature morphisms must preserve the arities of symbols, and arities are not preserved in managers. A similar situation is observed with formulae in the theory presentation. It is clear that all axioms of Producer are somehow "preserved" in the producer manager, since what we want to capture is the fact that all (live) producer instances behave as the producer specification indicates. A way to solve this problem, the mismatch between the notion of signature morphism and what is needed for capturing the component-manager relationship, would be to redefine the notion of signature morphism, so that new parameters are allowed when a symbol is translated. We have attempted this approach, which led to a complicated, badly structured, characterisation [2]. In particular, redefining the notion of signature morphism forced us to redo many parts of the traditional definition of institutions. In this paper, we present a different characterisation, which is much simpler and better structured. In this approach, we do not characterise the relationship between components as managers *within* an institution, but *outside* institutions, employing the notion of *representation map* [22].

As we explained before, the static description of components is given in **Pres**, the category of theory presentations (in first-order linear temporal logic), where the objects of the category define the syntax (signature) and axioms characterising component behaviour. Diagrams in this category correspond to component based designs, indicating the way components interact in a system, and colimits of these diagrams correspond to the behaviour of the structured design, "linked" as a monolithic component (the colimit object). These diagrams, and their colimits, characterise *static* composition, in a suitable way. In order to provide a dynamic behaviour associated with components, we start by constructing *managers* of components, as we illustrated for producers. First, let us consider an endofunctor $(-)^M : \mathbf{Sign} \to \mathbf{Sign}$, which maps each signature Σ to the signature Σ^M, obtained simply by incorporating a new sort $\bullet_\Sigma$, and a new parameter of this sort in each of the (flexible) symbols of Σ. Notice that the logic needs to support arguments in symbols (i.e., it needs to provide a notion of parameterisation), since otherwise adding a new parameter to a symbol would not be possible. For the case of first-order linear temporal logic, the functor $(-)^M$ maps a signature $\Sigma = \langle S, V, F_r, F_f, R_r, R_f\rangle$ (where S is the set of sorts, V the set of variables, and F and R the sets of function and predicate symbols, separated into flexible ("f" subscript) and rigid ("r" subscript) symbols) to the signature $\Sigma^M = \langle S^M, V, F_r, F_f^M, R_r, R_f^M\rangle$, where: *(i)* $S^M = S \cup \{\bullet_\Sigma\}$, where $\bullet_\Sigma$ is a sort name such that $\bullet_\Sigma \notin S$, *(ii)* $F_f^M = \{f : \bullet_\Sigma, w \to s \mid f : w \to s \in F_f\}$, and *(iii)* $R_f^M = \{r : \bullet_\Sigma, w \mid r : w \in R_f\}$.

Notice that we incorporate the extra parameter only into the symbols that constitute the state of the component. For statically interpreted symbols, the extra parameter is unnecessary. The way in which $(-)^M$ chooses the new sort name $\bullet_\Sigma$ for each Σ is not important for our current purposes. In our example, we chose a new sort NAME, for the identifiers of instances of components. With respect to morphisms, $(-)^M$ maps each signature morphism $\sigma : \Sigma \to \Sigma'$ to morphism $\sigma^M : \Sigma^M \to \Sigma'^M$, defined exactly as σ but mapping $\bullet_\Sigma \mapsto \bullet_{\Sigma'}$. Functor $(-)^M$ captures the relationship between components and their managers, via a construction which is external to the category of signatures (i.e., via a functor, not a morphism). In order to capture the relationship that exists between the specification of a component and the specification of the manager of this component, we now define a natural transformation $\eta^M : \mathbf{Sen} \stackrel{.}{\to} \mathbf{Sen} \circ (-)^M$. This natural transformation corresponds to a mapping $\eta^M_\Sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma^M)$, which maps any formula φ of $\mathbf{Sign}(\Sigma)$ to a formula φ^M of $\mathbf{Sign}(\Sigma^M)$. The definition of φ^M is simple: for each element of the signature which represents part of the "state" appearing in φ (in our case, a flexible symbol), add an extra parameter of type $\bullet_\Sigma$, and universally quantify it. Notice that this requires the logic to be first-order, so that universal quantification is possible. In the case of our specifications, given any formula φ we choose a $\bullet_\Sigma$-labelled variable $x_{\bullet_\Sigma}$. Each occurrence of any flexible function symbol $f : w \to s$ of the form $f(t_1, \dots, t_n)$ (where $t_1, \dots, t_n$ are terms) is replaced by the term $f(x_{\bullet_\Sigma}, t_1, \dots, t_n)$ (note that $f : \bullet_\Sigma, w \to s$ is a flexible function symbol of Σ^M), and similarly for flexible relations. After this step, we obtain a formula φ'. Finally, we define $\varphi^M = (\forall x_{\bullet_\Sigma} \in \bullet_\Sigma : \varphi')$. Again, we have captured the relationship between a component specification and the specification of its corresponding manager *externally*, via a natural transformation, instead of internally, within the category of specifications.

Finally, let us deal with *models*. We define a natural transformation $\gamma : \mathbf{Mod} \circ ((-)^M)^{\mathsf{op}} \stackrel{\cdot}{\to} \mathbf{Mod}$. That is, we have a natural family of functors $\gamma_\Sigma : \mathbf{Mod}(\Sigma^M) \to \mathbf{Mod}(\Sigma)$, which maps each model M' of $\mathbf{Mod}(\Sigma^M)$ to a model M of $\mathbf{Mod}(\Sigma)$, the model obtained by taking away the parameters of type $\bullet_\Sigma$ in every function and relation in M'. In a similar way, any morphism $m : M' \to M$ of $\mathbf{Mod}(\Sigma^M)$ can be translated to a morphism in $\mathbf{Mod}(\Sigma)$ $\gamma_\Sigma : \gamma_\Sigma(M') \to \gamma_\Sigma(M)$, corresponding to the restriction of m to the set of sorts different from $\bullet_\Sigma$. For the sake of brevity, we skip the detailed definition of these natural transformations.

For our first-order linear temporal logic, we have the following property:

Property 1. For every signature Σ, $\varphi \in \mathbf{Sen}(\Sigma)$ and $M' \in \mathbf{Mod}(\Sigma^M)$ the following holds: $M' \models_{\Sigma^M} \eta_\Sigma(\varphi) \Leftrightarrow \gamma_\Sigma(M') \models_\Sigma \varphi$.

Intuitively, this property states that γ and η preserve the satisfaction relation. This kind of relation between institutions is called a *representation map* [22]. Since the logic for components and for managers is the same, we have an endo-representation map, which relates components with their managers. Let us recall the definition of representation map, as given in [22].

Definition 2. *(Representation map between institutions)*
Let $\langle \mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \{\models_\Sigma\}_{\Sigma \in |\mathbf{Sign}|}\rangle$ *and* $\langle \mathbf{Sign}', \mathbf{Sen}', \mathbf{Mod}', \{\models'_\Sigma\}_{\Sigma \in |\mathbf{Sign}'|}\rangle$ *be the institutions* I *and* I' *respectively, then* $\langle \gamma^{Sign}, \gamma^{Sen}, \gamma^{Mod}\rangle : I \to I'$ *is a* representation map *of institutions if and only if:*

- $\gamma^{Sign} : \mathbf{Sign} \to \mathbf{Sign}'$ *is a functor,*
- $\gamma^{Sen} : \mathbf{Sen} \stackrel{\cdot}{\to} \mathbf{Sen}' \circ \gamma^{Sign}$*, is a natural transformation (i.e. a natural family of functions* $\gamma^{Sen}_\Sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\gamma^{Sign}(\Sigma))$*), such that for each* $\Sigma_1, \Sigma_2 \in |\mathbf{Sign}|$ *and* $\sigma : \Sigma_1 \to \Sigma_2$ *morphism is* $\mathbf{Sign}$*,*

$$\begin{tikzcd}
\mathbf{Sen}(\Sigma_2) \arrow[r, "\gamma^{Sen}_{\Sigma_2}"] & \mathbf{Sen}'(\gamma^{Sign}(\Sigma_2)) & \Sigma_2 \\
\mathbf{Sen}(\Sigma_1) \arrow[r, "\gamma^{Sen}_{\Sigma_1}"] \arrow[u, "\mathbf{Sen}(\sigma)"'] & \mathbf{Sen}'(\gamma^{Sign}(\Sigma_1)) \arrow[u, "\mathbf{Sen}'(\gamma^{Sign}(\sigma))"'] & \Sigma_1 \arrow[u, "\sigma"']
\end{tikzcd}$$

– $\gamma^{Mod} : \mathbf{Mod}' \circ (\gamma^{Sign})^{\mathsf{op}} \xrightarrow{\cdot} \mathbf{Mod}$, *is a natural transformation (i.e. the family of functors* $\gamma^{Mod}_{\Sigma} : \mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\Sigma)) \to \mathbf{Mod}(\Sigma)$ *is natural), such that for each* $\Sigma_1, \Sigma_2 \in |\mathbf{Sign}|$ *and* $\sigma : \Sigma_1 \to \Sigma_2$ *morphism in* $\mathbf{Sign}$*,*

$$\begin{tikzcd}
\mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\Sigma_2)) \arrow[r, "\gamma^{Mod}_{\Sigma_2}"] \arrow[d, "\mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\sigma^{\mathsf{op}}))"] & \mathbf{Mod}(\Sigma_2) \arrow[d, "\mathbf{Mod}(\sigma^{\mathsf{op}})"] & \Sigma_2 \\
\mathbf{Mod}'((\gamma^{Sign})^{\mathsf{op}}(\Sigma_1)) \arrow[r, "\gamma^{Mod}_{\Sigma_1}"] & \mathbf{Mod}(\Sigma_1) & \Sigma_1 \arrow[u, "\sigma"']
\end{tikzcd}$$

such that for any $\Sigma \in |\mathbf{Sign}|$*, the function* $\gamma^{Sen}_{\Sigma} : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}'(\gamma^{Sign}(\Sigma))$ *and the functor* $\gamma^{Mod}_{\Sigma} : \mathbf{Mod}'(\gamma^{Sign}(\Sigma)) \to \mathbf{Mod}(\Sigma)$ *preserves the following satisfaction condition: for any* $\alpha \in \mathbf{Sen}(\Sigma)$ *and* $\mathcal{M}' \in |\mathbf{Mod}(\gamma^{Sign}(\Sigma))|$*,*

$$\mathcal{M}' \models_{\gamma^{Sign}(\Sigma)} \gamma^{Sen}_{\Sigma}(\alpha) \;\; \textit{iff} \;\; \gamma^{Mod}_{\Sigma}(\mathcal{M}') \models_{\Sigma} \alpha \ .$$

Representation maps have been studied in detail in [22, 13]. The intuition that leads us to think that "all instances of a certain component type behave as the component (type) specification indicates" is justified by the following property of representation maps (see [22]):

Property 2. Semantic deduction is preserved by representation maps: for any institution representation $\rho : \mathbf{I} \to \mathbf{I}'$, signature $\Sigma \in |\mathbf{Sign}|$, set $\Phi \subseteq \mathbf{Sen}(\Sigma)$ of Σ-sentences, and Σ-sentence $\varphi \in \mathbf{Sen}(\Sigma)$, if $\Phi \models_{\Sigma} \varphi$, then $\rho^{Sen}_{\Sigma}(\Phi) \models'_{\rho^{Sign}(\Sigma)} \rho^{Sen}_{\Sigma}(\varphi)$.

Intuitively, this property says that managers of components preserve the properties that the specification of the corresponding components imply. Notice that this is the usual intuition: when one is reasoning about a *class* in object oriented design, one does so thinking of a *generic* template of instances of the class, so that the programmed behaviour will be that of all instances of the class[2]. This construction is also associated with some specification related mechanisms; for instance, the notion of *schema promotion* in Z [27] is captured by this very same notion of representation map. The promoted schemas are obtained via natural transformations from the original set of schemas.

4 Managing Dynamic Population and Interaction

In our initial example, a basic structure of a system is given in terms of component specifications, as well as specifications of the interactions between components. We also mentioned that these interactions are materialised as channels,

[2] Obviously, classes also define additional behaviour, associated with the manipulation of instances (constructors, destructors, etc.).

which enable one to define (categorical) diagrams, corresponding to static architectural designs of systems. We have already dealt with part of the generalisation of this situation to allow for dynamic creation/deletion of components, via component managers. We still need to describe the way in which the component population is actually managed, and how instances of components interact. For example, we would need ways of dynamically managing the population of components, and dynamically allocating live producers to live consumers, in the context of our example. In order to achieve the first of these goals, one needs to provide extensions of the manager components, introducing some specific behaviours into the managers (for example, some actions and properties related to the creation or deletion of components at run time). This, of course, needs to be manually specified, whereas the relativisation of component behaviour to instance behaviour is directly handled by the above presented representation map. For the case of our producer manager, such an extension could be the one presented in Figure 4. Notice how a set of live instances is introduced (via a flexible predicate), and how actions for population management can be specified. In this example, we have *new*, which allows us to create new producers. Axiom 10, for instance, indicates that in order to make an instance live, it must be originally "dead", and that *p-init* is executed at creation, on the newly created instance.

Now, let us deal with the connections. In order to dynamically allocate producers to consumers, we define a kind of connection template (we then exploit the previously introduced representation map to build a connection manager). We start by identifying the parts of the components that possibly need to be coordinated. In our case, we identify the fields of producers and consumers that need to be "exported" to other components, and the actions that need to be synchronised. These communicating elements are combined via a coproduct, yielding a vocabulary with parts from producers and parts from consumers, that will be used in order to describe the possible interactions between these types of components. This situation can be generically illustrated as follows:

$$\begin{tikzcd}
\mathbf{C_1} & & \mathbf{Con_1^1} + \mathbf{Con_2^1} & & \mathbf{C_2} \\
& \mathbf{Con_1} \arrow[ul, "m"'] \arrow[ur, "i"] & & \mathbf{Con_2} \arrow[ul, "j"'] \arrow[ur, "n"] & \\
& & 0 \arrow[uu, "!"] \arrow[uull, bend left, "!"] \arrow[uurr, bend right, "!"'] & &
\end{tikzcd}$$

This diagram involves two components, $\mathbf{C_1}$ and $\mathbf{C_2}$. The first component has a communicating language $\mathbf{Con_1}$ and the second component has a communication language $\mathbf{Con_2}$. As stated above, we want to connect these types of components using these communication languages. In contrast to our initial (static) example, we do not identify common parts in the components, but use the coproduct of the communicating languages to obtain a language in which to describe the interactions. This also provides more flexibility in the communication definition. In addition, the diagram involves the initial object of category **Sign**, together

Component: *ExtendedProducerManager*
Read Variables: *ready-in* : NAME → Bool
Attributes: *producers* : NAME → Bool, *p-current* : NAME → Bit,
p-waiting : NAME → Bool
Actions: *produce-0*(n: NAME), *produce-1*(n: NAME), *send-0*(n: NAME),
send-1(n: NAME), *p-init*(n: NAME), *new*(n: NAME)
Axioms:
... /* axioms of ProducerManager */
9. $\forall n \in \text{NAME} : \neg \textit{producers}(n)$
10. $\Box(\forall n \in \text{NAME} : \textit{new}(n) \rightarrow \neg \textit{producers}(n) \wedge \bigcirc(\textit{producers}(n) \wedge \textit{p-init}(n)))$
11. $\Box(\forall n \in \text{NAME} : \textit{produce-0}(n) \rightarrow \textit{producers}(n))$
...

Fig. 4. An extension of the producer manager, which handles instance creation.

with the (unique) morphisms ! from this object to the other components. This is necessary since, after applying $(-)^M$, we obtain a component 0^M which has only one sort, and the arrows $!^M$ (obtained applying the functor $(-)^M$ to the arrows !) identify all the new sorts added in the other components by $(-)^M$. The explicit inclusion of the initial object has as a consequence that only one new sort (of component names) is included in the final design.

A suitable connection template for our example could be the one in Figure 5. A manager of this specification is built in the same way that managers of components are constructed. In the same way that we extended the managers of components, we will need to extend the manager of connections, to indicate how connections work. A sample extension of the manager of connections is also shown in Fig. 5. The generic situation depicted in the previous diagram can be expanded, by means of the introduced representation map, to the following:

$$\begin{tikzcd}
\mathbf{MC_1} & & \mathbf{MI} & & \mathbf{MC_2} \\
\mathbf{C_1}^M \arrow[u] & & (\mathbf{Con_1^1} + \mathbf{Con_2^1})^M \arrow[u] & & \mathbf{C_2}^M \arrow[u] \\
& \mathbf{Con_1^M} \arrow[ul, "m^M"'] \arrow[ur, "i^M"'] & & \mathbf{Con_2}^M \arrow[ul, "j^M"] \arrow[ur, "n^M"] & \\
& & 0^M \arrow[uu, "!^M"] \arrow[uull, bend left, "!^M"] \arrow[uurr, bend right, "!^M"'] & &
\end{tikzcd}$$

The specifications C_1^M, $(\mathbf{Con_1^1} + \mathbf{Con_2^1})^M$ and $\mathbf{C_2}^M$ are obtained via the functor $(-)^M$. The components $\mathbf{MC_1}$, $\mathbf{MI}$ and $\mathbf{MC_2}$ are the (ad-hoc) extensions of the manager components. As for the case of static configurations, the colimit of this diagram gives us the final design. It is interesting to note that, since we have used abstract concepts such as institutions and representation maps, the concepts introduced in this section can be instantiated with other logics.

Component: *ConnectionTemplate*
Attributes: *ready-in* : Bool, *ready-ext* : Bool, *p-waiting* : Bool, *c-waiting* : Bool
Actions: *send-0*, *send-1*, *extract-0*, *extract-1*

Component: *ExtendedConnectionManager*
Attributes: *ready-in* : NAME → Bool, *ready-ext* : NAME → Bool,
p-waiting : NAME → Bool, *c-waiting* : NAME → Bool,
connected : NAME, NAME → Bool
Actions: *send-0*(n: NAME), *send-1*(n: NAME),
extract-0(n: NAME), *extract-1*(n: NAME), *connect*(n, m: NAME)
Axioms:
1. $\Box(\forall n, m \in \text{NAME} : \mathit{connect}(n,m) \to \neg \mathit{connected}(n,m) \wedge \bigcirc(\mathit{connected}(n,m))$
2. $\Box(\forall n, m \in \text{NAME} : \mathit{connected}(n,m) \to (\mathit{send\text{-}0}(n) \leftrightarrow \mathit{extract\text{-}0}(m)))$
...

Fig. 5. A connection template, indicating the vocabulary relevant for communication, and an extension of its manager.

In particular, there is no need to use the same logic for component specification and manager specification. Notice that to the extent that these logics can be connected by a representation map, all of the presented characterisation is applicable. For example, we can use a propositional temporal logic to describe the components, taking advantage of decision procedures for such a logic, and use a first-order temporal logic to describe the managers. The representation map between these two logics still enables us to "promote" the properties verified for components (algorithmically, if the logic for component specification is decidable) to properties of all instances of components. Furthermore, we could take this idea even further, and use yet another logic for datatype specification (e.g., a suitable equational logic), and promote the properties of datatypes to components and managers, again by exploiting representation maps.

5 Conclusions

Many specification languages need to deal with dynamic reconfiguration and dynamic population management. In Z, for instance, this is done via schema promotion [27], which is understood simply as a syntactical transformation. In the context of B [1], there is a similar need, in particular when using B as a target language for analysis of object oriented models (e.g., the UML-B approach). Object oriented extensions of model oriented languages, such as VDM++, Object Z or Z++, have built in mechanisms for dealing with dynamic reconfiguration, as is inherent in object orientation. Other logical languages, for instance some logical languages used for software architectures (e.g., ACME), also require dynamism in specifications. Generally, the mechanisms for dynamism in the mentioned languages are syntactical.

Besides the work mentioned above, there exist some other related approaches, closer to what is presented in this paper. A useful mechanism for formally characterising dynamic reconfiguration is that based on graph transformation, as in [14], which has been successfully applied in the context of dynamic software architectures [26]. As opposed to our work, in this approach the notion of manager is not present, and thus it is less applicable to contexts where this notion is intrinsic (e.g., object orientation, schema promotion, etc.). Another related approach is that of Knapp et al. [15], who present an approach for specifying service-oriented systems, with categorical elements. Knapp et al. employ mappings (from local theories to a global one) for specifying component synchronisation, but composition is not characterised via universal constructions, as in our approach. Another feature of our approach, not present in Knapp et al.'s, is the preservation of the original design's modularisation, via representation maps (notice that each component is mapped to a similar component in a more expressive setting, where its dynamic behaviour is expressed).

We presented the requirements for dealing with dynamic reconfiguration in a logical specification language, in a categorical way. Our categorical characterisation is general enough so that it applies to a wide variety of formalisms, which we have illustrated using a temporal logic. An essential characteristic of the logical system is that quantification is required, so that collections of instances of components can be handled. Our work might help in understanding the relationship between basic "building block" specifications and the combined, whole system, in the presence of dynamic reconfiguration.

We also believe that this work has interesting practical applications. The categorical setting we are working with admits working with different (but related) logics for component specification, and the description of dynamic systems. This might enable one, for instance, to use a less expressive (perhaps decidable) logic for the specification of components, whose specifications could be mapped to more expressive (generally undecidable) logics, where the dynamism of systems is characterised. Understanding the relationships between the different parts of the specification can be exploited for practical reasons, for example for promoting properties verified in components (using for example a decision procedure) to the specification of the system. Also, some recently emerged fields, such as service oriented architectures, require dealing with highly dynamic environments, where formalisations as the one proposed in this paper might be useful. As expressed in [23], there is a clear need for work in the direction of our proposal, so we plan to investigate the applicability of our approach in some of the contexts mentioned in [23].

References

1. J.R. Abrial, *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. N. Aguirre and T. Maibaum, *Some Institutional Requirements for Temporal Reasoning about Dynamic Reconfiguration*, in Proc. of Symposium on Verification, Theory and Practice, Taormina, Italy, LNCS, Springer, 2003.
3. R. Burstall and J. Goguen, *Putting Theories together to make Specifications*, in Proc. of the Fifth International Joint Conference on Artificial Intelligence, 1977.

4. M. Cengarle, A. Knapp, A. Tarlecki, M. Wirsing, *A heterogeneous approach to UML semantics*, in Proc. of Concurrency, graphs and models (Essays dedicated to Ugo Montanari on the occasion of his 65th. birthday). LNCS 5065, Springer, 2008.
5. Diaconescu, R., ed., *Institution-independent Model Theory*, Vol. 2 of Studies in Universal Logic, Birkhäuser, 2008.
6. H. Ehrig and B. Mahr, *Fundamentals of Algebraic Specification 2*, Springer, 1990.
7. J. Fiadeiro and T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification.* Formal Aspects of Computing, 4(3), Springer, 1992.
8. J. Fiadeiro and T. Maibaum, *Describing, Structuring and Implementing Objects*, In Proc. of the REX Workshop, LNCS 489, Springer-Verlag, 1990.
9. J. Fiadeiro, *Categories for Software Engineering*, Springer, 2004.
10. D. Garlan, *Software Architecture: A Roadmap*, in The Future of Software Engineering, A. Filkenstein (ed), ACM Press, 2000.
11. J. Goguen, R. Burstall, *Introducing institutions*, in Proc. of the Carnegie Mellon Workshop on Logic of Programs. LNCS 184, Springer, 1984.
12. Joseph A. Goguen and Rod M. Burstall, *Institutions: Abstract Model Theory for Specification and Programming.* J. ACM, vol. 39, No. 1, 1992.
13. Joseph A. Goguen and Grigore Rosu, *Institution Morphisms*, Formal Asp. Comput., volume 13, number 3-5, 2002.
14. D. Hirsch, U. Montanari, *Two Graph-Based Techniques for Software Architecture Reconfiguration.* Electr. Notes Theor. Comput. Sci. 51. 2001.
15. A. Knapp, G. Marczynski, M. Wirsing and A. Zawlocki, *A heterogeneous approach to service-oriented systems specification*, in Proc. of SAC 2010, ACM Press, 2010.
16. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
17. N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proc. of the Second Int. Software Architecture Workshop (ISAW-2), 1996.
18. J. Meseguer, *General Logics*, in Logic Colloquium'87, North Holland, 1989.
19. T. Mossakowski, C. Maeder, K. Luttich, *The heterogeneous tool set, Hets*, in Proc. of TACAS 2007. LNCS 4424, Springer, 2007.
20. B. Pierce, *Basic Category Theory for Computer Scientists* , The MIT Press, 1991.
21. D. Sannella and A. Tarlecki, *Specifications in an Arbitrary Institution*, Inf. Comput. 76(2/3), 1988.
22. A. Tarlecki, *Moving Between Logical Systems*, in Proc. of 11th Workshop on Specification of Abstract Data Types joint with the 8th COMPASS Workshop on Recent Trends in Data Type Specification, LNCS 1130, Springer, 1995.
23. A. Tarlecki, *Toward Specifications for Reconfigurable Component Systems*, ICATPN 2007.
24. Tarlecki, A., *Towards heterogeneous specifications*, in Gabbay, D., de Rijke, M., eds.: Frontiers of Combining Systems. Vol. 2 of Studies in Logic and Computation, Research Studies Press, 2000.
25. A. Tarlecki, *Abstract specification theory: an overview*, in Proc. of the NATO Advanced Study Institute on Models, Algebras and Logic of Engineering Software. NATO Science Series, Marktoberdorf, Germany, IOS Press, 2003.
26. M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.
27. J. Davies and J. Woodcock, *Using Z*, Prentice-Hall, 1996.