Design in CommUnity with Extension Morphisms

Xiang Ling[1], Tom Maibaum[1], and Nazareno Aguirre[2]

[1] Department of Computing and Software, McMaster University
1280 Main St West, Hamilton ON, Canada L8S 4K1
lingx@univmail.cis.mcmaster.ca, tom@maibaum.org
[2] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Ruta 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina
naguirre@dc.exa.unrc.edu.ar

Abstract. We have been engaged over the past few years in studying and formalizing software architecture concepts such as hierarchical design, dynamic reconfiguration and the application of the concept of aspects to software architecture descriptions. Our attention has focused on the language CommUnity, developed by Fiadeiro and Maibaum, and an extension that we call DynaComm that incorporates support for dynamic reconfiguration, hierarchical design, a general notion of connector and other supporting mechanisms. In applying DynaComm, we have found that the relationships normally used in CommUnity, i.e., regulative superposition (used to regulate the behaviour of a component) and refinement (used to instantiate a role in a higher order connector) are not sufficient for dealing with some required changes to a software architecture or a component that we would like to be able to affect. To this end, we have defined the concept of *extension morphism* between two components. Such morphisms do not preserve encapsulation of components, as do regulative superpositions and refinements, but they do give us substitutability, in the sense of object-oriented systems, and, hence, a basis of predictability about its application to designs. In this paper, we describe the nature of extension morphisms and illustrate their use by means of a non trivial example.

1 Introduction

1.1 Motivation and Background

Software architecture research is directed at addressing the high-level decomposition and organization of systems, where component interactions are incorporated into the notion of connectors and identified as first-class design entities. Architecture description languages (ADLs) have been proposed to provide formal modelling notations, analysis and development tools to support architecture-based development, which focuses on the system's high-level structure rather than the implementation details of any specific modules [33].

There has been some work in surveying ADLs providing broad comparisons. The survey in [33] compared ADLs with respect to their ability to model components, connectors and configurations, as well as their tool support for analysis and refinement. The survey in [13] focused on the characteristics of different ADLs supporting self-managing architectures, which not only implement the concept of dynamic change, but also initiate, select and assess the change itself without the assistance of an external user. We are currently interested in ADLs with support for dynamic software architectures and that also support the essential engineering concept of hierarchical design. Examples of ADLs, which support such a view, are Dynamic Wright [11], Darwin [31] and Dynamic Acme [19][38]. An assessment of these languages can be found in [13] and a more thorough review of their language constructs, associated styles of specification and mechanisms to achieve dynamic reconfiguration can be found in [27]. However, these ADLs have important shortcomings in relation to their support for hierarchical design and having a formal semantics that enable us to perform useful analyses.

CommUnity [15, 29, 30] was designed to study the problem of the 'magic step' from specification to program, in the context of component based design using temporal and multi modal logics for component specification. It is always the case that the languages used for specifications and programs are ontologically very different. Specifications are about properties, whilst programs are about operational behaviour, even if this behaviour is described abstractly. For one thing, a programming language has no facility to express *properties* of programs; a meta language of properties is required for this. So, programs and specifications occupy different conceptual worlds and there is not a simple notion of homomorphism or refinement that relates them directly: hence the reference to the 'magic step' above. What *is* the relationship between specifications and programs and how can one remove the magic? We cannot talk about the program being a refinement of a specification, as refinement is an internal notion in the language of component specifications. We might introduce a notion of *realization*, which relates a program to its specification by assigning to the program a minimal (not unique and minimum) specification, which *is* a refinement of the specification.

CommUnity explored this space and addressed the important issue of compositionality in this context: when can we say that a program constructed from parts, where each part is a correct realization as a program of the corresponding specification part, is correct with respect to the specification constructed from the component specifications in a way that mimics the construction of the program? Not too surprisingly, compositionality in this sense is not an easy property to achieve. Given an arbitrary specification language and some programming language, not every program constructed from parts is correct with respect to the corresponding specification. This is not surprising, as the structural properties of the specification category may not mimic that of the category of programs, or *vice versa*.

We have been extending CommUnity to encompass features we regard as essential for architecture based design, namely hierarchical organization of subsystems and dynamic reconfiguration [26]. However, in this paper the new features of DynaComm are not essential for the presentation, so, for the sake of clarity, we avoid the presentation of its extra details and features.

Recently, we have been exploring the issue of 'early aspects' [39], attempting to see if these ideas can be rationalized, based on traditional software engineering principles of modularity and hierarchy, by analyzing them at the architectural level. After numerous case studies, we have come to the conclusion that aspects are just the soft goals or non functional requirements traditionally found in requirements engineering, and that they can be handled uniformly at the architectural level by formalizing a specific aspect as an architectural pattern used to replace an existing pattern in the underlying architecture (by means of, for example, a graph transformation). Then, aspect weaving is achieved by the colimit construction used to obtain the semantics of any architectural configuration, the latter being defined as a categorical diagram of component objects and relationships between them. Aspect composition then becomes the sequential application of different transformations, corresponding to the different aspects, to the underlying diagram depicting the original architecture. As with features, there may be (unforeseen) interactions between different aspects and the order of application is crucial to achieving the right system. Many aspects require the replacement of a component in the original architecture by another, closely related, component that is a subtype of the original component, in the sense of object-oriented design. This requires a formal relationship between components that involves breaking encapsulation of the original component in the design. We have developed the notion of extension as a realization of this controlled breaking of encapsulation. The application of extension morphisms in the construction of software architectures is the aim of this paper.

1.2 Introducing CommUnity

CommUnity was developed to explore the relationships between specifications and programs in a component based development setting. José Fiadeiro and his (former) students developed the language extensively in the interim [18,29,30], making of CommUnity a (proto) ADL. A review of CommUnity and its semantics are given, and, in particular, we rehearse the idea that the notion of superposition can be formalized as a morphism between designs in CommUnity. The concept of superposition is defined as a structure preserving transformation on designs through the extension of their state space and control activity while preserving their properties [29,30]. So, a regulative superposition morphism is proposed in CommUnity as a means of augmenting an existing component by superposing a regulator over it while preserving its functionality, thus supporting a layered approach to system design. In addition, several different kinds of morphisms (other than regulative superposition morphisms) between designs as well as their relationships are also investigated to explain the language's well-founded support for compositionality, reusability, and enforcement of design principles.

The syntax of a CommUnity design is:

```
design component P
out out(V)
in in(V)
prv prv(V)
init I
do
      [prv] g[D(g)] : L(g), U(g) -> R(g)
endofdesign
```

A fixed collection of data types (say S) is assumed to be given by a first-order algebraic specification and the design is defined over such data types. Because data types chosen in the design determine the nature of the elementary computations that can be performed locally by the components, the emphasis in the language is put on the co-ordination mechanisms between system components rather than data refinement, which focuses on computational aspects. As a result, CommUnity does not support polymorphism directly.

In the above example, V is the set of *channels* in the design P. Each channel v is typed with a sort from S. in(V) represents input channels, which read data from the environment of the component and the component has no control over them. out(V) and prv(V) are output channels and private channels, respectively. They are controlled locally by the component. Output channels allow the environment to read data produced by the component, while private channels support internal activity that does not involve the environment. We use loc(V) to represent out(V) $\cup$ prv(V). The formula I constrains potential initial states of the corresponding program. I is a formula in first-order logic over the channels of the design.

For any action g, D(g) is a subset of loc(V) consisting of the local channels that can be written to by action g (we call it the write frame of g). U(g) is a progress condition, which establishes the upper bound for enabledness and L(g) indicates the lower bound. In a program, L(g) = U(g), so the guards in a design define the "interval" within which the guard of the action in a program implementing the design must lie. R(g) is a condition on V and D(g)', where by D(g)' we mean the set of primed channels from D(g). Primed channels account for references to the values of channels after the execution of an action. The condition is a first-order logic formula built from V and D(g)'. Usually, we define it as a conjunction of implications of the form pre $\Rightarrow$ post, which corresponds to a pre/post condition specification in the sense of Hoare and where pre does not contain primed channels. Using this form, the number of conjuncts in the formula will correspond to the number of channels in the write frame of g, so that we can understand the meaning of the action fairly easily. Moreover, it will be convenient for us to calculate the colimit of the diagram if we have put all the designs in this form.

In order to study the relationship between designs, we need the formal definition for designs as follows:

Definition 1. A *design signature* is a tuple (V, Γ, tv, ta, D) where:

- V is the set of *channels*, which is an S-indexed family of mutually disjoint sets. The channel is typed with sorts in S, which is a fixed set of data types specified as usual via a first-order specification.
- Γ is a finite set of actions.
- tv is a total function from V to {prv, in, out}, which partitions V into three disjoint sets of channels, namely private, input and output channels, respectively. Loc(V) represents the union of private and output channels.
- ta is a total function from Γ to {sh, prv}, which divides Γ into private and shared actions. Only shared actions can serve as the synchronization points with other designs.
- D is total function from Γ to $2^{loc(V)}$. The write frame of action g is represented by D(g).

All these sets of symbols are assumed to be finite and mutually disjoint. Channels are used as atoms in the definition of terms:

Definition 2. Given a design signature θ=(V, Γ, tv, ta, D), the language of *terms* is defined as follows: for every sort s ∈ S,

- t_s ::= a , where a ∈ V and of type s
- t_s ::= c, where c is a constant with sort s
- $t_s ::= f(t_1,\ldots,t_n)$, where $t_1: s_1,\ldots, t_n: s_n$ and $f{:}s_1 \times \ldots \times s_n \to s$

The language of *propositions* is defined as follows:

- $\phi ::= (t_{1s}\ p_s\ t_{2s}) \mid \phi_1 \Rightarrow \phi_2 \mid \phi_1 \wedge \phi_2 \mid \neg\phi$

where p_s is a binary predicate defined on sort s. The set of predicates defined on sort s must contain $=_s$.

Having defined the signature of designs and given the language of terms and propositions, we can formalize the notion of designs as follows:

Definition 3. A *design* is a pair (θ, Λ), where θ = (V, Γ, tv, ta, D) and Λ is (I, R, L, U) where:

- I is a proposition defined on θ, which constrains the values of the channels when the program is initialized.
- R assigns to every action g ∈ Γ an expression R(g).
- For every action g ∈ Γ, L(g) assigns the enabling guard to it and U(g) assigns the progress guard.
- For every action g ∈ Γ, for any a ∈ D(g), tv(a) ∈ {prv, out}.

Recall that R(g) specifies the effect of action g on its write frame. For any channel a ∈ D(g), we will use R(g,a) to denote the expression that represents the effect of action g on channel a.

Before we define the semantic structures for a design, a model for the abstract data type specification (S) needs to be introduced. The model is given by a Σ-algebra *U*, i.e., a set s^U is assigned to each sort symbol s ∈ S, a value in $s^U(c^U)$ is assigned to each constant symbol c of sort s, a (total) function $f^U: s_1^U \times \ldots \times s_n^U \to s^U$ is assigned to each function symbol f in S, and a relation $p_s^U \subseteq s \times s$ is assigned to each binary predicate p_s defined on sort s.

The semantic interpretation of designs is given in terms of transition systems:

Definition 4. A *transition system* (W, w_0, E, →) consists of:

- a non-empty set *W* of states or possible worlds
- $w_0 \in W$, the initial state
- a non-empty set *E* of events
- an *E*-indexed set of partial functions → on *W*, $W \to (E \to W)$, defines the state transition performed by each event.

Having transition systems to represent the state transitions of a design, we can interpret the signature of a design with the following structure:

Definition 5. A *θ-interpretation structure* for a signature θ=(V, Γ, tv, ta, D) is a triple (*T*, *A*, *G*) where:

- *T* is a transition system (W, w_0, E, $\rightarrow$)
- *A* is an S-indexed family of maps A_s: $V_s \rightarrow (W \rightarrow s^U)$.
- G: $\Gamma \rightarrow 2^E$.

That is to say, *A* interprets attribute symbols as functions that return the value that each attribute takes in each state, and *G* interprets the action symbols as sets of events -- the set of the events during which the action occurs.

It is possible that no action will take place during an event. Such events correspond to environment steps, which means steps performed by the other components in the system. Interpretation structures are intended to capture the behavior of a design in the context of a system of which it is a component. Because environment steps are taken into account, state encapsulation techniques can be formalized through particular classes of interpretation structures.

Definition 6. A *θ-interpretation structure* (*T*, *A*, *G*) for a signature θ=(V, Γ, tv, ta, D) is called a *locus* iff, for every a ∈ loc(V) and w, w'□ ∈ *W*, if (w, e, w') is in →, and for any g ∈ D(a), e ∉ *G*(g), then *A*(a)(w') = *A*(a)(w).

This means a *locus* is an interpretation structure in which the values of the program variables remain unchanged during events in which no action occurs that contains them in their write frame.

Having defined the interpretation structures for designs and the model for the abstract data type specification (S), we are able to give the semantics of the terms and propositions in the language given by the design signature.

Definition 7. Given a signature θ = (V, Γ, tv, ta, D) and a *θ-interpretation structure* *S*= (*T*, *A*, *G*), the semantics of terms (for every sort s, term t of sort s and w ∈ *W*, $[t]^s(w) \in s^U$, the value taken by t in the world w, is defined as follows:

- if t is a ∈ A_s, $[a]^s(w) = A(a)(w)$
- if t is a constant c, $[c]^s(w) = c^U$
- if t is $f^U : s_1^U \times \ldots \times s_n^U \rightarrow s^U$, $[f(t_1,t_2,\ldots,t_n)]^s(w) = f^U([t_1]^s(w), [t_2]^s(w), \ldots, [t_n]^s(w))$

The semantics of propositions is defined as:

- (S,w) t ($t_1 =_s t_2$) iff $[t_1]^s(w) = [t_2]^s(w)$
- (S,w) t ($t_1\ p_s\ t_2$) iff $[t_1]^s(w)\ p_s^U\ [t_2]^s(w)$
- (S,w) t $\phi_1 \Rightarrow \phi_2$, iff ($S$,w) t ϕ_1 implies (S,w) t ϕ_2
- (S,w) t ($\neg\phi$) iff $\neg$((S,w) t ϕ)

Now on the semantic level, we can represent whether a proposition (in a signature) is true or valid in the interpretation structure of the signature:

Definition 8. A *θ*-proposition ϕ is *true* in an *θ*-interpretation structure *S*, written *S* t ϕ, iff (S,w) t ϕ at every state w. A proposition ϕ is *valid*, written t ϕ, iff it is true in every interpretation structure.

Having introduced the above concepts, we can now define when an interpretation structure is a model of a design.

Definition 9. Given a design (θ, Λ), where θ = (V, Γ, tv, ta, D) and Λ is a triple (I, R, L, U), a *model* of (θ, Λ) is an interpretation structure S=(T, A, G) for θ, such that:

- (S,w$_0$) ⊨ I
- for every g ∈ Γ, a ∈ D(g), e ∈ G(g) , and (w, e, w') ∈ →, then A(a)(w')= $[R(g,a)]^S(w)$
- for every w ∈ W and g ∈ Γ, if e ∈ G(g) and for some w' ∈ W, (w, e, w') ∈ →, then (S,w) ⊨ L(g).

That is to say, a *model* of a design is an interpretation structure for its signature that enforces the assignments, only permits actions to occur when their enabling guards are true, and for which the initial state satisfies the initialization constraint.

A model is said to be a *locus* if it is a locus as an interpretation structure, which enforces the encapsulation of local attributes.

This classification of models reflects the existence of different levels of semantics for the same design (taken as a set of models), depending on which subset of the set of its models is considered. These different semantics are associated with different notions of superposition (design morphism) that have been used in the literature, namely regulative, invasive and spectative. This means that there is no absolute notion of semantics for designs: it is always relative to the use one makes of designs. This corresponds to the categorical way of capturing the "meaning" of objects through the relationships (morphisms) that can be defined between them.

1.3 The Morphisms Between Designs

The concept of superposition has been proposed and used as a structuring mechanism for the design of parallel programs and distributed systems. Structure preserving transformations are usually formalized in terms of morphisms between the objects concerned, thus justifying the formalization of superposition in terms of morphisms of designs in CommUnity.

Having defined designs over signatures in the above section, we first introduce signature morphisms as a means of relating the "syntax" of two designs.

Definition 10. A *signature morphism* σ from a signature θ_1=(V_1, Γ_1, tv_1, ta_1, D_1) to θ_2=(V_2, Γ_2, tv_2, ta_2, D_2) consists of a total functions σ_α: $V_1 \rightarrow V_2$, and a partial mapping σ_γ: $\Gamma_2 \rightarrow \Gamma_1$ such that:

- For every v ∈ V_1 , σ_α(v) has the same type as v.
- For every o ∈ out(V_1), σ_α(o) ∈ out(V_2).
- For every p ∈ prv(V_1), σ_α(p) ∈ prv(V_2).
- For every i ∈ in(V_1), σ_α(i) ∈ out(V_2) ∪ in(V_2).

For every g ∈ Γ_2 , such that σ_γ(g) is defined:

- g ∈ sh(Γ_2), then σ_γ(g) ∈ sh(Γ_1).
- g ∈ prv(Γ_2), then σ_γ(g) ∈ prv(Γ_1).
- $\sigma_\alpha(D_1(\sigma_\gamma(g)) \subseteq D_2(g)$.

A signature morphism maps attributes of a design to attributes of the system of which it is a component, and the direction of the mapping is reversed for actions. The first condition enforces the preservation of the type of each attribute by the morphism. Output and private attributes of the component should keep their classification in the system, while input attributes may be turned into output attributes, when they are synchronized with output channels of other components and thus represented as output channels of the system. The restriction over action domains means that the type of each action is preserved by the morphism. In other words, the images of the write frame of an action in the source program must be contained in the write frame of the corresponding action in the target program. Notice that more attributes may be included in the domain of the target program's action via a morphism. This is intuitive because an action of a component may be shared with other components within a system and, hence, has a larger domain.

Signature morphisms provide us with the means for relating a design with its superpositions. However, superposition is more than just a relationship between signatures on the level of syntax. To capture its semantics, we need a way of relating the models of the two designs as well as the terms and propositions that are used to build them.

Signature morphisms define translations between the languages associated with each signature in the obvious way:

Definition 11. Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$, we can define translations between the languages associated with each signature:

- if t is a term:
  $\sigma(t)$ ::= $\sigma(a)$ if t is a variable a
  c if t is a constant c
  $f(\sigma(t_1),\ldots, \sigma(t_n))$ if $t= f(t_1,\ldots, t_n)$
- if ϕ is a proposition:
  $\sigma(\phi)$ ::= $\sigma(t_1) = \sigma(t_2)$ if ϕ is $t_1 = t_2$
  $\sigma(t_1)\ p_s\ \sigma(t_2)$ if ϕ is $t_1\ p_s\ t_2$
  $\sigma(\phi_1) \Rightarrow \sigma(\phi_2)$ if ϕ is $\phi_1 \Rightarrow \phi_2$
  $\sigma(\phi_1) \wedge \sigma(\phi_2)$ if ϕ is $\phi_1 \wedge \phi_2$
  $\neg\sigma(\phi')$ if ϕ is $\neg\phi'$

Definition 12. Given a signature morphism σ: $\theta_1 \rightarrow \theta_2$ and a θ_2–interpretation structure $S = (T, A, G)$, its *σ-reduct,* $S|_\sigma$, is the θ_1–interpretation structure $(T, A|_\sigma, G|_\sigma)$, where $A|_\sigma(a) = A(\sigma(a))$, $G|_\sigma(g) = \cup\, G(\sigma^{-1}(g))$.

That is, we take the same transition system of the target design and interpret attribute symbols of the source design in the same way as their images under σ, and action symbols of the source design as the union of their images under σ^{-1}. Reducts provide us with the means for relating the behavior of a design with that of the superposed one. The following proposition establishes that properties of reducts are characterized by translation of properties.

Proposition 1. Given a θ_1 proposition ϕ and a θ_2–interpretation structure $S=(T, A, G)$, we have for every $w \in W$: $(S, w) \models \sigma(\phi)$ iff $(S|_\sigma, w) \models \phi$.

Superposition morphisms that preserve locality are called *regulative* superposition morphisms and are defined as follows:

Definition 13. A *regulative superposition morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a signature morphism σ: $\theta_1 \rightarrow \theta_2$ such that:

0 $\vdash (I_2 \Rightarrow \sigma(I_1))$.

1 If $v \in loc(V_1)$, $g \in \Gamma_2$ and $\sigma_\alpha(v) \in D_2(g)$, then g is mapped to an action $\sigma_\gamma(g)$ and $v \in D_1(\sigma_\gamma(g))$.

For every $g \in \Gamma_2$ for which $\sigma_\gamma(g)$ is defined,

3 If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then $\vdash (R_2(g, \sigma_\alpha(v)) \Leftrightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)))$.

4 $\vdash (L_2(g) \Rightarrow \sigma(L_1(\sigma_\gamma(g))))$.

5 $\vdash (U_2(g) \Rightarrow \sigma(U_1(\sigma_\gamma(g))))$.

Notice that we do not require σ_α to be injective, and two channels of the same category (output/private/input) in the source design can be mapped to one channel of the target design. Because we only consider the actions in the target design mapped to the source design, σ_γ does not need to be surjective.

The second condition implies that actions of the system in which a component C is not involved cannot have local channels of the component C in their write frame, which corresponds to the locality condition: new actions cannot be added to the domains of attributes of the source program. The justification is as follows: suppose system action g has $\sigma_\alpha(v)$ in its write frame, $v \in loc(V_1)$, then $\sigma_\gamma(g)$ must be defined, and $\sigma_\gamma(g) \in D_1(v)$. Therefore, component C is involved in the system action.

Regulative superposition morphisms require that the functionality of the base design in terms of its variables be preserved (the underspecification cannot be reduced) and allows for the enabling and progress conditions of its actions to be strengthened. Strengthening of the lower bound reflects the fact that all the components that participate in the execution of a joint action have to give their permission for the action to occur. On the other hand, the progress of a joint action can only be guaranteed when the involved components can locally guarantee so. Regulative superpositions preserve encapsulation and do not change the actions themselves, as far as they relate to the basic variables.

Proposition 2. Let σ: $(\theta_1, \Lambda_1) \rightarrow (\theta_2, \Lambda_2)$ be a regulative superposition morphism. Then the reduct of every model of (θ_2, Λ_2) is also a model of (θ_1, Λ_1).

We find that in the proof of proposition 2.2, we do not use condition 2 of regulative superposition morphism, which means this proposition will hold without enforcing the encapsulation principle. When we consider condition 2 and the definition of signature morphism, we will have the following assertion:

Proposition 3. If $v \in loc(V_1)$, then $D_1(v) = \sigma_\gamma(D_2(\sigma_\alpha(v)))$.

This result implies the following property:

Proposition 4. Let σ: $(\theta_1, \Lambda_1) \rightarrow (\theta_2, \Lambda_2)$ be a regulative superposition morphism; then the reduct of every locus of (θ_2, Λ_2) is also a locus of (θ_1, Λ_1).

The reason is that through regulative superposition, the domains of the attributes remain the same up to translation, as stated above. Therefore, it will prevent “old attributes” from being changed by “new actions”, i.e., actions of the target design not mapped to the source design.

Now we will introduce the notion of extension morphism, related to ideas of model-expansiveness. The motivation for extension morphisms originated from the substitutability principle from object oriented program design, which says if a component P_2 extends another component P_1, then we can replace P_1 by P_2 and the “clients” of P_1 must not perceive the difference. This principle cannot be characterized by regulative superpositions or refinement morphisms, as we may want to extend the component by breaking encapsulation. This controlled breaking of encapsulation is necessary when dealing with many aspects.

Definition 14. An *extension morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a signature morphism σ such that:

1 σ_γ is surjective.
2 σ_α is injective.
3 There exists a formula β, which contains only channels from $(V_2 - \sigma_\alpha(V_1))$, such that β is satisfiable and t $I_2 \Leftrightarrow \sigma(I_1) \wedge \beta$.

For every $g \in \Gamma_2$ for which $\sigma_\gamma(g)$ is defined,

4 If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then there exists a formula β, which contains only primed channels from $(V_2' - \sigma_\alpha(V_1)')$, and β is satisfiable and such that t $\sigma(L_1(\sigma_\gamma(g))) \Rightarrow (R_2(g, \sigma_\alpha(v)) \Leftrightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)) \wedge \beta)$.
5 If $v \in loc(V_1)$, $g \in D_2(\sigma_\alpha(v))$, then $v \in D_1(\sigma_\gamma(g))$.
6 t $(\sigma(L_1(\sigma_\gamma(g))) \Rightarrow L_2(g))$.
7 t $(\sigma(U_1(\sigma_\gamma(g))) \Rightarrow U_2(g))$.

This definition of extension morphism was first given [8]. Because we expect that the extended design can replace the original design in a system and the clients of the original component should not perceive any difference, the first two conditions ensure the preservation of its interface. The initialization condition of the original design can be strengthened in its extended version, while respecting the initialization of the channels of the original component, as required in the third condition. The fourth condition indicates that the actions corresponding to those of the original design should preserve the assignments to old channels and the assignments to new channels must be realisable, when the safety guards of their image actions in the original design are satisfied. The fifth condition establishes that for each action of the extended design that is mapped to an action of the original design, it can only modify old channels that have been modified by the corresponding action of the original design. The last two conditions indicate that both the enabling and progress guards can be weakened, but not strengthened.

Because an extension morphism relaxes the enabling guard of the source design, the reduct of a model of the target design may not be a model of the source design. However, the model-expansive property holds for extension morphism [8], which means the extended design can replace the source design and the clients of the original design will not perceive the difference.

Proposition 5. Let σ be an extension morphism from a design (θ_1, Λ_1) to another design (θ_2, Λ_2). Then, every model of (θ_1, Λ_1) can be expanded to a corresponding model of (θ_2, Λ_2).

The rationale behind the definition of extension morphisms is the characterization of the substitutability principle (a property that can be shown to fail for invasive superposition, a more general and less predictable way of breaking encapsulation, as defined in [15]). The above result shows that, if there exists an extension morphism σ between two designs (θ_1, Λ_1) and (θ_2, Λ_2) (and this extension is realisable), then all behaviours exhibited by (θ_1, Λ_1) are also exhibited by (θ_2, Λ_2). Since superposition morphisms, used as a representation of "clientship" (strictly, the existence of a superposition morphism between two designs indicates that the first is part of the second, as a component is part of a system when the first is used by the system), restrict the behaviours of superposed components, it is guaranteed that all behaviours exhibited by a component when this becomes part of a system will also be exhibited by an extension of this component, if replaced by the first one in the system. Of course, one can also obtain *more behaviours*, and this is the intention behind the definition of extension morphisms, resulting from the explicit use of new actions of the component. But if none of the new actions are used, then the extended component behaves exactly as the original one did.

Now we introduce the relationship of refinement between two components, which we need to enable us to use the architectural concept of connector.

Definition 15. A *refinement morphism* σ from a design (θ_1, Λ_1) to another design (θ_2, Λ_2) is a signature morphism $\sigma: \theta_1 \rightarrow \theta_2$ such that:

1 For every $i \in in(V_1)$, $\sigma_\alpha(i) \in in(V_2)$.

2 σ_α is injective on input and output channels.

3 σ_γ is surjective on shared actions in Γ_1.

4 $\vdash (I_2 \Rightarrow \sigma(I_1))$.

5 If $v \in loc(V_1)$, $g \in \Gamma_2$ and $\sigma_\alpha(v) \in D_2(g)$, then g is mapped to an action $\sigma_\gamma(g)$ and $v \in D_1(\sigma_\gamma(g))$.

For every $g \in \Gamma_2$ where $\sigma_\gamma(g)$ is defined,

6 If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then $\vdash (R_2(g, \sigma_\alpha(v)) \Rightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)))$.

7 $\vdash (L_2(g) \Rightarrow \sigma(L_1(\sigma_\gamma(g))))$.

For every shared action $g \in \Gamma_1$,

8 $\vdash (\sigma(U_1(g)) \Rightarrow \wedge U_2(\sigma_\gamma^{-1}(g)))$.

A refinement morphism identifies a way in which design (θ_1, Λ_1) is refined by a more concrete design (θ_2, Λ_2). The first three conditions must be established to ensure that refinement does not change the interface between the system and its environment. Notice that we do not require σ_γ to be injective because the set of actions in the target design that are mapped to action g of the source design can be viewed as a menu of refinements that is made available for implementing g. Different choices can be made at different states to take advantage of the structures available at the more concrete level.

As for the "old actions", the last two conditions in the refinement morphism definition require that the interval defined by their enabling and progress conditions must be preserved or reduced. This is intuitive because refinement should reduce underspecification, so the enabling condition of any implementation must lie in the "old interval": the lower bound cannot be weakened and the upper bound cannot be strengthened. This is also the reason why the underspecification regarding the effects of the actions of the more abstract design are intended to be reduced.

Proposition 6. The structure composed of CommUnity designs and superposition/refinement/extension morphisms constitutes a category SUP/REF/EXT, respectively, where the composition of two morphisms σ_1 and σ_2 is defined in terms of the composition of the corresponding channel and action mappings of σ_1 and σ_2.

So, we can build superpositions/refinements/extensions incrementally. Most importantly, SUP has finite colimits, i.e., we can compute the system corresponding to a configuration of CommUnity designs whose channels and actions are synchronized via cables and superposition morphisms. So called *higher order connectors* [29] are defined in CommUnity to enable designers to use complex connectors between components, in the style of software architecture approaches. These higher order connectors are just CommUnity designs in which some components play a designated *role*, namely stating minimum requirements of actual components to be connected by the connector in question. One can instantiate a role with a 'real' component by defining a refinement from the role to the component. Thus, when designing a system using components and connectors, we may end up with a configuration in which we see both regulative superpositions and refinements. In order to calculate the intended system form this configuration, we must eliminate the refinements and thus get a configuration in SUP.

Luckily, we have the following crucial result about the joint use of refinement and superposition morphisms. If we restrict the kinds of components used to interconnect components to so called *cables*, we can combine superposition morphisms from such a cable with a refinement. A cable is a design containing only input channels and its actions having the following form `g: true -> skip`. We only expect input channels in the cable, which can be used to interconnect designs, because output channels cannot be used to connect the input channel of one design with the output channel of another design, and it will make no sense to interconnect output channels of different designs. Also we set the enabling guard and progress guard of each action in the cable to true and set R(g) to `skip` (by `skip` we mean this action has no effects on the local channels of the design), which is good enough to synchronize the actions.

Proposition 7. Suppose m is a regulative superposition morphism from cable θ to design C_i and n is a refinement morphism from design C_i to design E_i; there exists a regulative superposition morphism n' from cable θ to design E_i such that n'=n•m.

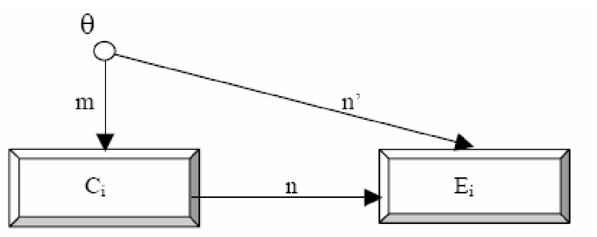


Fig. 1. Combining regulative superposition and refinement morphisms

2 CommUnity and Extension Morophisms

It has been shown in [5] that higher-order connectors provide a very convenient basis for enhancing the behavior of an architecture of component designs, by the superimposition of aspects, such as fault tolerance, security, monitoring, compression, etc. Owing to the coordination mechanism of CommUnity, which externalizes completely the definition of interaction between components, the coupling between the components has been reduced to a minimum so that we can superimpose aspects on existing systems through replacement, superposition and refinement of components. However, higher-order connectors are not powerful enough for defining various kinds of aspects, because some of them require *extensions* of the components and connectors [8], which break encapsulation of the extended component, though in a controlled and predictable way. (The usual relationships used in CommUnity, i.e., regulative superposition and refinement, preserve encapsulation: channels (attributes) of the original component are not modified by new actions of the new component and actions of the original component can only have their enabling guards and effects strengthened in the new component.) Hence, we defined an extension morphism as a mechanism for modifying/adapting components, in a way that satisfies the notion of substitutability arising in the context of object oriented design and programming [8], enables us to predict properties of extended components in a safe manner and enables the design of various aspects [8].

This means that in a well-formed configuration diagram we should be able to replace component C by its valid extension, component C', and preserve the well formedness (our ability to compute the colimit) of the diagram. We prove this property in the next section. To illustrate the application of this principle in designing systems with the CommUnity language, a vending machine system example will be discussed below to show how we can combine regulative superpositions with extension morphims to derive an "augmented" version of the original system, where the modified system is not simply a refinement of the original, nor is it a regulated version of the original obtained by the use of regulative superpositions (the usual structuring relationship in CommUnity).

2.1 Combining Regulative Superpositions with Extension Morphisms

In this section we will consider the case where, in a well-formed configuration diagram, one component is extended by a design through an extension morphism. Since we know that, in a well formed configuration diagram, all the components are interconnected by cables through regulative superposition morphisms, the component to be replaced by the extended design is connected to a cable by the regulative superposition morphism, as shown in Figure 2. We will show that the regulative superposition can be combined with the extension morphism to obtain a new regulative superposition from the cable to the extended component. This then allows us to apply the mechanisms of CommUnity to obtain the semantics of the extended configuration diagram, the colimit, which again consists of components connected through cables and superposition morphisms. Again, it is crucial to have the notion of cables to interconnect the components, to ensure that the composition of regulative superposition and extension morphism will give a new regulative superposition.

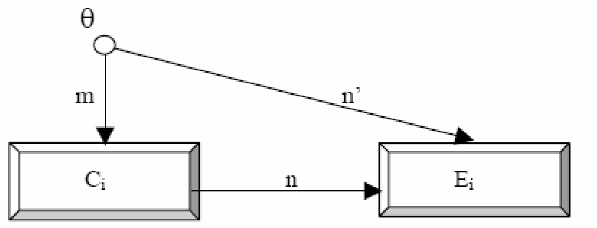


Fig. 2. Combining regulative superposition and extension morphisms

Proposition 8. Suppose m is a regulative superposition morphism from cable θ to design C_i and n is an extension morphism from design C_i to design E_i; there exists a regulative superposition morphism n' from cable θ to design E_i such that n'=n•m.

Proof
The morphism n' is defined as follows:

- n'_α is a total function: for every channel v in θ, $n'_\alpha(v) = n_\alpha(m_\alpha(v))$.
- n'_γ is a partial mapping: for every action g in E_i, if $n_\gamma(g)$ is defined and $m_\gamma(n_\gamma(g))$ is also defined, $n'_\gamma(g) = m_\gamma(n_\gamma(g))$; otherwise, it is undefined.

Since an extension morphism is also a signature morphism, we know n' is a signature morphism. To check if n' is a regulative superposition morphism, we need to check the following conditions:

- $I_{Ei} \Rightarrow n'(I_\theta)$.

Because n is an extension morphism, there exists a formula α, using only channels contained in $(V_{Ei} - n_\alpha(V_{Ci}))$, and α is satisfiable, $\vdash I_{Ei} \Leftrightarrow n(I_{Ci}) \wedge \alpha$.
We have $I_{Ei} \Rightarrow n(I_{Ci})$, $I_{Ci} \Rightarrow m(I_\theta)$, so $n(I_{Ci}) \Rightarrow n(m(I_\theta)) \Leftrightarrow n'(I_\theta)$, and $I_{Ei} \Rightarrow n'(I_\theta)$.

- If $v \in loc(\theta)$, $g \in \Gamma_{Ei}$ and $n'_\alpha(v) \in D_{Ei}(g)$, then g is mapped to an action $n'_\gamma(g)$ and $v \in D_\theta(n'_\gamma(g))$.
- For every $g \in \Gamma_{Ei}$ where $n'_\gamma(g)$ is defined, if $v \in loc(\theta)$ and $g \in D_{Ei}(n'_\alpha(v))$, then $R_{Ei}(g, n'_\alpha(v)) \Leftrightarrow n'_\alpha(R_\theta(n'_\gamma(g), v))$.

Because θ only contains input channels, $loc(\theta)$ is empty, so these two conditions hold.

- $L_{Ei}(g) \Rightarrow n'(L_\theta(n'_\gamma(g)))$.
- $U_{Ei}(g) \Rightarrow n'(U_\theta(n'_\gamma(g)))$.

From our definition of "middle" design, $L_\theta(n'_\gamma(g)) \Leftrightarrow$ true, $U_\theta(n'_\gamma(g)) \Leftrightarrow$ true, so these two conditions hold.

With this property, in a well-formed configuration diagram, we are able to replace a component by its extension component, by combining the regulative superposition from the cable to the old component with the extension morphism between the old component and its extension, to obtain a new regulative superposition from the cable to the extended component. If we build several extensions, each built on top of the previous one, then the fact that extensions compose in the category of CommUnity designs and extension morphisms guarantees that this composition is an extension. Hence, the above result still applies when we build extensions incrementally. Therefore, we reach the conclusion that in a well-formed configuration diagram of a system, we can extend any subcomponents of the system (through extension morphisms), and thus obtain an updated well-formed configuration diagram only containing regulative superpositions, through which the semantics of the new system can be derived from its colimit. Moreover, it can be shown that the colimit of the new configuration diagram is an extension of the colimit of the old configuration diagram [8].

By examining the proof of proposition 8, we can see that, if θ_i is not a cable, the composition of a regulative superposition and an extension morphism may not give a regulative superposition. Therefore, it is necessary to enforce designs to be interconnected by cables in a well-formed configuration diagram, so that the colimit will exist after extending any of the designs in the diagram through extension morphisms. (This result mimics the properties of refinements in the context of cables and regulative superpositions.)

3 An Example Vending Machine System

Now we want to model a system consisting of a customer and a vending machine with the DynaComm language, to illustrate the use of hierarchical design and then to illustrate the use of extension morphisms to enable us to modify our design in a way not allowed by refinements and regulative superpositions. The requirement of this system is described as follows: *The vending machine maintains a list of items, along with the price and amount of each item. The customer can place an order by inputting the name of the item and the payment to the vending machine. Initially, we only allow the customer to order one item in a transaction; this will be extended later. The vending machine will check the price of the item and decide if the order is accepted. If so, it will deliver the item along with the change to the customer; otherwise, the payment is returned to the customer. Initially, the vending machine will only accept payment comprised of nickels, dimes, quarters and loonies (Canadian single dollars using the image of a local bird), so it will refuse the order if the customer puts a one cent piece in the payment slot. Meanwhile, if the vending machine is not able to make the change, it will also refuse the order and return the payment.*

3.1 The Design of the Customer

We consider the machine's interface, operated by the customer, as the simulation of the customer's behavior. To make the system simple and general at first, the interface is divided into two parts: the buttons and the slot. The names of different items label the corresponding item buttons, and after the customer presses one of them, other item buttons will be disabled, so that he can only choose one item in an order. Then the customer can choose the "confirm" button to continue the order, where the slot will indicate to him to put the coins in and the complete order will be sent to the vending machine. If the customer chooses the "cancel" button, all the item buttons will be enabled and he can start another order.

The vending machine will check the price of this order and whether the ordered item is still available in its storage. If so, it will ask the slot to make the change. Then the vending machine will deliver the product to the slot and enable the item buttons, if the change can be made. Otherwise, the order will be refused and the payment is returned to the customer.

3.1.1 The Interface Controller

According to the above requirement, the customer places his order of an item through the buttons (including the item buttons and the command buttons: confirm and cancel) on the machine's interface, so we design an interface controller to model these buttons, as well as the customer's interaction with the interface of the machine. A finite set of actions for the item buttons and "confirm", "cancel" buttons are specified in the following design. The slot_get and slot_ret actions are designed to interact with the slot component to obtain the payment from the customer. Meanwhile, we use the order action to send the complete order to the vending machine, and after the order has been processed by the vending machine, the order_ret action will be called to reset the controller.

```
design component controller
in      // the customer's payment in the slot
        i_pay: int
prv     b_item: array(int);
        bt_g: bool;  //guard for item buttons
        bt_confirm: bool;  //guard for confirm/cancel buttons
        slot_g: bool;  // guard for slot get action
        s_req: bool;
        ord_g: bool;  // guard for order action
        o_req: bool
out     // order to vending machine
        c_item: list (int);
        c_pay: int
init    ord_g = false ∧ o_req = false ∧ bt_g = true ∧ bt_confirm
        = false ∧ slot_g = false ∧ s_req = false ∧ c_item = NULL
actions
        button_select(id: int)[bt_g,c_item,bt_confirm]: bt_g,
        false ->
        bt_g' = false ∧ c_item' = c_item * b_item [id] ∧
        bt_confirm' = true
```

```
    [] button_confirm[bt_confirm,slot_g]: bt_confirm, false ->
       bt_confirm′ = false ∧ slot_g′ = true
    [] button_cancel[bt_confirm,bt_g,c_item]: bt_confirm, false
       -> bt_g′ = true ∧ bt_confirm′ = false ∧ c_item′ = NULL
    [] slot_get[slot_g,s_req]: slot_g, false ->
       slot_g′ = false ∧ s_req′ = true
    [] slot_ret[c_pay, s_req, ord_g]: s_req, false ->
       c_pay′ = i_pay ∧ s_req′ = false ∧ ord_g′ = true
    [] order[o_req,ord_g]: ¬o_req ∧ ord_g, false ->
       o_req′ = true ∧ ord_g′ = false
       // enable all the item buttons
    [] order_ret[o_req, bt_g, c_item]: o_req, false ->
       o_req′ = false ∧ bt_g′ = true ∧ c_item′ = NULL
endofdesign
```

The input channel i_pay indicates the payment received from the customer. A finite set of item button actions (button_select) are defined, which correspond to the sequence of item buttons on the machine's interface. These actions are examples of *schema actions* indexed by the id (in the above sequence) of the item buttons. Such schema actions may be used to describe succinctly a finite set of related actions, distinguishable by means of some index set. See [27] for a full explanation of such families of actions and their precise semantics. We use a fixed size array b_item to store the item's index in the storage of the vending machine, and the index of array b_item will correspond to the id of the item button, e.g., the second item button b_item[2] may correspond to the item index 6 in the item list of the vending machine's storage.

The workflow of the controller component is described as follows. After one item button is selected, the guard bt_g is set to false to disable all the item buttons, so that the customer can only choose buttons confirm or cancel (as the enabling guards of other actions are disabled). If he chooses the confirm button, the guard slot_g is enabled and the slot_get action will be executed to request the customer's payment in the slot component. If the cancel option is selected, the controller will enable all the item buttons and wait for the customer's input of a new transaction. After the payment is obtained from the slot, the order action will be called and it will send the order

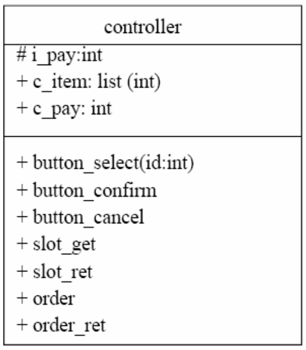


Fig. 3. Graphical representation of the controller component

(c_item, c_pay) to the vending machine, then wait for the result of the order. After the vending machine processes the order and indicates the result to the order_ret action of the controller, the order_ret action will reset the item buttons and the c_item list, to be ready to accept another order. The graphical notation for the (syntax of the) controller component is shown in Figure 3 (we suppress private channels and actions):

Notice that we use a number of guards to control the sequence of actions in the controller, and the correctness of our design can be ensured by maintaining the right workflow of the component through the appropriate use of these guards. We also use the list data structure to record the ordered items, although currently only one item is allowed in the order. The reason is that in the different kinds of design morphisms we have discussed so far, the mapping of channels requires the types of channels to be preserved. (Refinement morphisms do not support data refinement, so a refinement solution to get around this problem is not available.) If we use one channel of integer type to record the ordered item now and there is a new requirement to allow the customer to select multiple items in an order, we have to add new channels to the component and modify the corresponding actions as well, which seems awkward. Therefore, we choose the list data structure for the ordered items and the corresponding actions are designed to process the list of items.

We have also designed a pattern for a pair of actions of one component (e.g. slot_get and slot_ret), which sends a request to another component and waits for its response to proceed. The trick is to assign a guard (initialized to be false) to the call-back action to make sure that it will not be called arbitrarily in an unexpected situation, and it will only be enabled in the request action.

3.1.2 The Slot

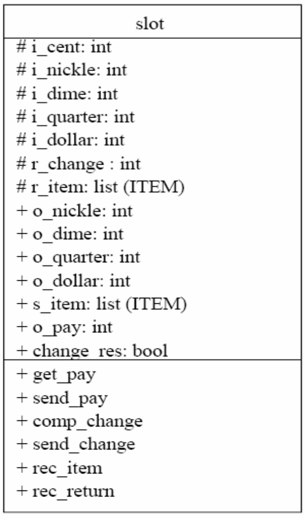
The slot component takes care of the acceptance of the customer's payment and decides if the correct change can be made depending on its current store of coins. When the interface controller requests the payment from the customer, the slot will distinguish the various kinds of coins and it will refuse the payment and indicate this event to the controller if there exists an illegal coin in the customer's input. Otherwise, it will store the coins and send the payment amount to the controller. Regarding the function for making the change, the slot is able to compute the composition of coins for the amount of change requested by the vending machine, based on its current store. If the computation is not successful, the vending machine will refuse the order and inform the slot to return the payment, which can certainly be made.

In the following design of component slot, a set of input channels such as i_dollar, i_quarter, etc. represents the payment from the customer, a set of private channels is included as the coin store of the slot, and we also use output channels o_nickel, o_dime, o_quarter and o_dollar to represent the change made by the slot. The get_pay action stores the coins in the payment and the send_pay action puts the amount of payment in the output channel o_pay. According to the amount of change that should be made in the input channel r_change, the comp_change action will compute the composition of coins, and the send_change action will send the result of the computation (change_res) and update the storage of coins if needed. While the ordered item is accepted by the action rec_item, and the rec_return action receives the returned payment amount and returns the coins to the customer.

```
design component slot
in     // input coins from customer
       i_cent: int;
       i_nickle: int;
       i_dime: int;
       i_quarter: int;
       i_dollar: int;
       // received change amount and items from vending machine
       r_change : int;
       r_item: list(ITEM)
prv    // coins storage
       s_nickle: int;
       s_dime: int;
       s_quarter: int;
       s_dollar: int;
       // guards for action sequence
       get_g: bool;
       change_g:bool;
       item_g: bool
out    // changes made by the slot
       o_nickle: int;
       o_dime: int;
       o_quarter: int;
       o_dollar: int;
       s_item: list (ITEM);  // items to slot
       o_pay: int; // payment amount to the controller
       change_res: bool
init   get_g = true ∧ change_g = true ∧ change_res = false ∧
       item_g = false
actions
       get_pay[get_g, s_nickle, s_dime, s_quarter, s_dollar]:
       get_g ∧ i_cent = 0, false ->
       get_g' = false ∧ s_nickle' = s_nickle + i_nickle ∧
       s_dime' = s_dime + i_dime ∧ s_quarter' = s_quarter +
       i_quarter ∧ s_dollar' = s_dollar + i_dollar
    [] send_pay[get_g, o_pay]: ¬ get_g, false ->
       get_g' = true ∧ o_pay' = 100*i_dollar + 25*i_quarter +
       10*i_dime + 5*i_nickle
    [] comp_change[change_g, change_res]: change_g, false ->
       get_changed ∧ change_g' = false
    [] send_change[change_g]: ¬change_g, false ->
       change_g' = true ∧ (change_res = true ⇒  item_g' = true ∧
       s_nickle' = s_nickle - o_nickle ∧ s_dime' = s_dime -
       o_dime ∧ s_quarter' = s_quarter - o_quarter ∧ s_dollar' =
       s_dollar - o_dollar)
    [] rec_item[s_item, item_g]: item_g, false ->
       s_item' = r_item ∧ item_g' = false
    [] rec_return[ret_g, s_item,]: true, false ->
       s_item' = NULL ∧ get_changed
endofdesign
```

In the above design, we assume the function to compute the composition of change, namely get_change, has already been defined, which takes r_change as input and computes the number of nickels, dimes, quarters and dollars. If the computation is successful, it will set change_res to be true and the output channels for the change. Otherwise, change_res is set to false and this event is sent to the vending machine. Actually, get_change solves a linear programming problem, which takes s_nickel, s_dime, s_quarter, s_dollar and r_change as parameters. To simplify the specification of the slot component, we do not describe the detailed procedure here.

The workflow of the slot component is described as below. When the interface controller requests the payment from the customer, the get_pay and send_pay actions will be executed to provide the payment amount to the controller. After the vending machine receives the order and recognizes that the payment is enough, it will ask the slot to compute the change. So, the comp_change action is called and the result of computation (change_res) is sent to the vending machine by the send_change action. If the result is successful, the change is given to the customer by the slot and the vending machine will send the product to the slot by means of the rec_item action. Otherwise, the rec_return action will get the amount of payment from the vending machine and give it back to the customer by calling the get_change function. The graphical notation for the slot component is as follows, where we again suppress the private channels and actions.


slot

i_cent: int
i_nickle: int
i_dime: int
i_quarter: int
i_dollar: int
r_change : int
r_item: list (ITEM)
+ o_nickle: int
+ o_dime: int
+ o_quarter: int
+ o_dollar: int
+ s_item: list (ITEM)
+ o_pay: int
+ change_res: bool

+ get_pay
+ send_pay
+ comp_change
+ send_change
+ rec_item
+ rec_return

Fig. 4. Graphical representation of the slot component

3.2 The Design of the Vending Machine

Based on the functional requirement of the vending machine, we will divide it into two components: vender and inventory, where the vender is in charge of the interaction with the customer interface (controller and slot), and the inventory serves as a database for storing the actual products (items) and maintaining the price and amount of each item.

3.2.1 The Vender

The job of the vender is to accept the order from the customer (the accept action), ask the inventory to check the price and amount of the ordered item(s) (actions check_inv and check_ret), send the amount of change to the slot and ask if the change can be made (actions change and change_ret), request the item(s) from the inventory (actions req_item and req_return), deliver the item(s) to the customer (the delivery action) or return the payment (the return_ord action), and inform the interface controller to be reset to start a new order (the reset_controller action). The design of the vender component is as follows, the meaning of the channels being explained in the comments.

```
design component vender
in      // the ordered item(s) and payment from the controller
        in_item: list(int);
        in_pay: int;
        // the price of the ordered item(s) from the inventory
        inv_price: int;
        inv_item: list(ITEM);
        // the result of checking whether the change can be made
        from the slot
        chg_res: bool
prv     // the set of guards to control the sequence of actions
        ac: bool;
        ck: bool;
        cg: bool;
        rt: bool;
        rq:bool;
        rc: bool;
        dl:bool;
        // stores the requested item(s) from the inventory
        v_item: list(ITEM);
        // stores the order and payment from the customer
        ord_item: list(int);
        ord_pay: int
out     // the order and payment to be sent to the inventory
        ck_item: list(int);
        ck_pay: int;
        // the amount of change to be sent to the slot
        chg_amt: int;
        // the ordered item(s) sent to the customer
        out_item: list(ITEM);
        // the returned amount of payment to be sent to the slot
        ret_amt: int
init    ac′ = false ∧ ck′ = false ∧ cg′ = false ∧ rt′ = false ∧
        dl′ = false ∧ rq′ = false ∧ rc′ =false
actions
         [ac, ord_item, ord_pay, ck]: ¬ac, false ->
        ac’ = true ∧ ord_item’ = in_item ∧ ord_pay’ = in_pay ∧ ck’
        = true
```

```
    []  check_inv[ck, ck_item, ck_pay]: ck, false ->
        ck_item’ = ord_item ∧ ck_pay′ = ord_pay ∧ ck’ = false
    []  check_ret[cg, rt, v_item]: true, false ->
        (inv_price >= 0 ⇒ cg’ = true) ∨(inv_price = 0 ⇒ rt’ =
        true)
    []  change[cg, chg_amt]: cg, false ->
        chg_amt′ = ord_pay − inv_price ∧ cg’ = false
    []  change_ret[rq, rt ]: true, false ->
        (chg_res = true ⇒ rq’ = true) ∨(chg_res = false ⇒ rt’
        = true)
    []  req_item[rq, ck_item]: rq, false ->
        ck_item’ = ord_item ∧ rq’ = false
    []  req_return[v_item, dl]:true, false ->
        v_item’ = inv_item ∧ dl’ = true
    []  return_ord[rt, ret_amt, out_item, ac, rc]: rt, false ->
        rt’ = false ∧ ret_amt’ = ord_pay ∧ out_item’ = NULL ∧
        rc’ = true
    []  delivery[dl, ac, out_item]: dl, false ->
        dl’ = false ∧ out_item’ = v_item ∧ rc’ = true
        // inform the controller to accept another order
    []  reset_controller[rc]: rc, false ->
        rc’ = false ∧ ac’ = false
endofdesign
```

According to the initialization condition of this design, only the accept action is enabled and it is synchronized with the order action of controller to accept the order of the customer. It also sets the guard ck to be true, so that the check_inv action will be executed to ask the inventory to check the price and amount of the ordered item(s). The check_ret action waits for the response from the inventory: if inv_price>=0, it means that the transaction can continue and this action sets the guard cg to be true, to call the slot to check if the change can be made; otherwise, it enables the guard rt to call the return_ord action, if any item is not available or the payment is not enough.

If the order can continue, the change action is synchronized with the comp_change action of the slot to make the appropriate change to the customer. Then the change_ret action will wait for the response from the slot indicated by the input channel chg_res: if the change can be made, the vender will request the item from the inventory using the req_item action, which is synchronized with the rec_req action of the inventory; otherwise, the return_ord action is called to return the payment. After the vender receives the requested item from the inventory using the req_return action, the delivery action will be called, which is synchronized with the rec_item action of the slot to deliver the item. Otherwise, the action return_ord will be executed and the slot’s action rec_return will be synchronized to return the payment to the customer. Finally, the vender will call the reset_controller action to synchronize with the order_ret action of the controller to inform it that the next order can now be taken.

The graphical notation for the vender component is depicted in Figure 5 below (we ignore private channels and actions).

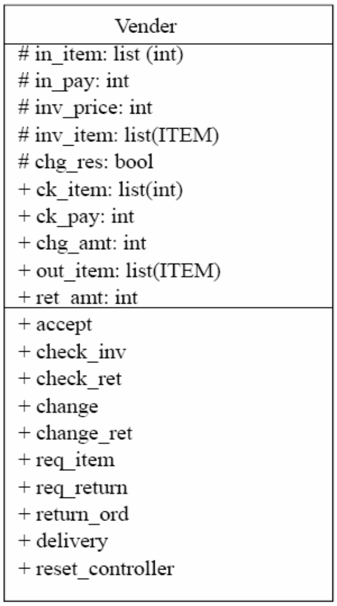


Fig. 5. Graphical representation of the vender component

Again, we use a set of guards to control the sequence of actions in the vender component, and, in the above explanation of the component's work mechanism, we are able to control the right workflow of the design through the appropriate use of these guards, so that the correctness of our design can be ensured.

3.2.2 The Inventory

The inventory component maintains a list of items along with their price and remaining amount: (item_id:int, item:ITEM, price:int, amount:int), where item_id is the item's index in the storage and item represents the real item product. We use an array db (with a fixed size) to store this list of items, and the index of this array corresponds to item_id. Meanwhile, we assume functions first, second and third have been defined to return the first, second and third member of db, respectively.

The private action count_item calculates the amount of each ordered item and stores it in the channel s_item. It also computes the total price of the order. The check_price action goes through the inventory database and compares the amount of each ordered item with the amount of that item in the storage. If the storage is not enough or the payment is less than the price of the order, the output channel will be set to 0; otherwise, it will set to the value in p_price. The get_item action will retrieve the items from the storage according to the order and update the db channel. The specification of the inventory component is as follows:

```
design component inventory
in     // the ordered item(s) and payment from the vender
       i_item: list (int);
       i_pay: int
```

```
prv    // stores the ordered item(s)
       p_item: list (int);
       r_item: list (int);
       p_price: int;
       // array index is item id
       db: array (ITEM, int, int);
       // stores the amount of each ordered item, all the en-
       tries are initialized to be 0.
       s_item: array (int);
       j :int;
       // the guards to control the sequence of actions
       price_g: bool;
       amt_g: bool;
       ret_g: bool;
       send_g : bool
out    o_item: list (ITEM);
       // the price of the order sent to the vender
       o_price: int
init   p_item = NULL ∧  price_g = false ∧ amt_g = false ∧ ret_g
       = false ∧ o_price = 0 ∧ o_item = NULL ∧ r_item = NULL ∧
       send_g = false
actions
       check[]: true, false -> p_item′ = i_item
    [] prv count_item[]: p_item != NULL, false ->
       s_item[head(p_item)]′ = s_item[head(p_item)] + 1 ∧
       p_price′ = p_price + second(db[head(p_item)]) ∧ p_item′ =
       tail(p_item) ∧ (tail(p_item) = NULL ⇒ price_g′ = true)
    [] prv check_price[: price_g, false ->
       price_g’ = false ∧ ((i_pay >= p_price ⇒ amt_g’ = true ∧ j’
       = 1) ∨(i_pay < p_price ⇒ ret_g’ = true ∧ o_price’ = 0))
    [] prv check_amt[]: amt_g ∧ (j <= sizeof(db)) , false ->
       ((s_item[j] <= third(db[j]) ⇒ j’ = j + 1 ∧ (j =sizeof(db)
       ⇒ ret_g’ = true ∧ o_price’ = p_price)) ∨(s_item[j] >
       third(db[j]) ⇒ amt_g’ = false ∧ o_price’ = 0 ∧ ret_g’ =
       true))
    [] inv_ret[]: ret_g, false -> ret_g′ = false
    [] rec_req[]: true, false -> r_item′ = i_item
    [] prv get_item[]: r_item != NULL, false -> o_item ′ =
       o_item * first(db[head(r_item)]) ∧
       third(db[head(r_item)])′ = third(db[head(r_item)])-1 ∧
       r_item′ = tail(r_item) ∧ (tail(r_item) = NULL ⇒ send_g′
       = true)
    [] send_item[]: send_g, false -> send_g′ = false
endofdesign
```

The workflow of this component is as follows. First, the check action is called to enable the guard of the count_item action. Then the action check_price is called to decide if the total price is less than ck_pay. If so, the inv_ret action will be enabled to return the result (inv_price) to the vender. Otherwise, the check_amt action is executed to check if the amount of each ordered item in the inventory is greater than the number of this item requested in the order. If so, it will call action inv_ret to return inv_price > 0 (the total price of the items in ck_item); otherwise, it will return inv_price = 0 in the inv_ret action. After the vender verifies that the change can be made, it will call the req_item action, which is synchronized with the rec_req action of the inventory, to get the ordered items and update the storage, and the inventory has the send_item action to send the ordered items back to the vender.

Notice that in the count_item action we use the guard p_item != NULL to iterate through the list of ordered items. It can be generalized as a mechanism to implement the loop structures in the DynaComm language. (See future work.)

3.2.3 The Vending Machine Subsystem

According to our design of the vender and inventory components and the discussion of their interactions, we can put them together by interconnecting the vender and the inventory through a cable. The CommUnity Workbench like notation of Figure 5 describes the configuration diagram of the vending machine subsystem. The solid circles attached to a component description represent elements of the interface of that component. A line connecting two such interface elements, say sync1 of cable and chaeck_inv of Vender, indicate that in the categorical diagram corresponding to that of Figure 5, the regulative superposition from cable to Vender maps the action sunc1 to the action check_inv. So, this configuration diagram corresponds exactly to a well defined and well formed diagram in the category of CommUnity designs and regulative superposition morphisms. The colimit of this categorical diagram is the intended semantics of the configuration.

The specification of the vending machine subsystem can be obtained easily from the above configuration diagram and we do not describe it in detail here. We can also determine the interface of this subsystem by looking at the left part interface of the vender component in the diagram, which will interact with the interface controller and the slot.

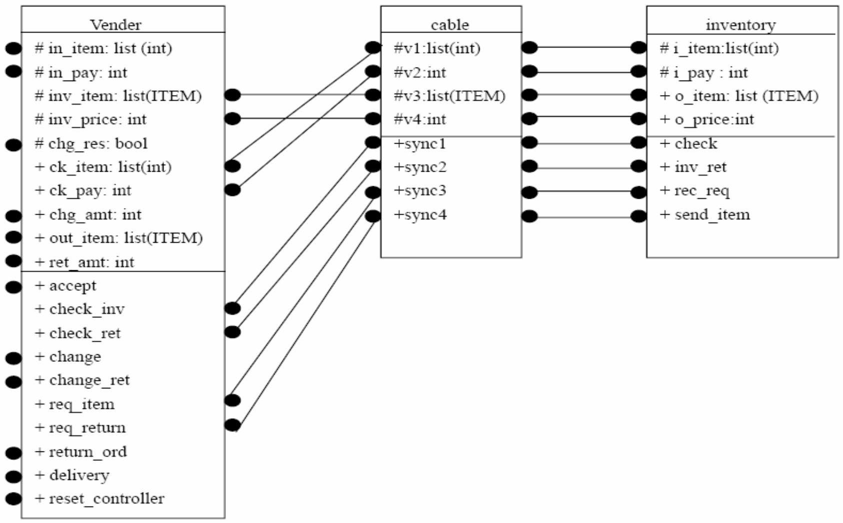


Fig. 6. Configuration diagram of the vending machine subsystem

Now we can put the vending machine subsystem together with the interface part (the controller and slot) to obtain the required vending machine system, which satisfies the design requirements, and the morphims between them are described in the configuration diagram depicted in Figure 6.

The interface of the vending machine system is shown in the left interface section of the controller component and the right interface section of the slot component, in which the controller provides the buttons for the customer to select his favorite item and confirm or cancel the order, and the slot indicates to the customer to put the coins in and to get his ordered item and change.

3.3 The Extended Vending Machine System

Now we want to add more behaviors to the vending machine system to improve the quality of its service. There are two extensions to be made, one for allowing a customer to order more than one item in a single transaction and the other to allow more kinds of coins to be used in payments, and we will show that they can only be achieved by the usage of extension morphisms.

3.3.1 The Extension Allowing Multiple Items in an Order

One extension we want to make is to allow the customer to select more than one item in an order, which should be done in the controller component. We must modify the actions of item buttons to achieve this effect. First, we will extend the controller component and show there is an extension morphism from the old controller to this extended new component. Then a proof is given to justify that it is impossible to regulate or refine the controller to obtain the required functionality and the extension morphism is necessary for our purpose.

We introduce a new channel ac: bool (initialized to be true) and weaken the guards of item buttons actions by taking the disjunction of ac with bt_g. The modified actions of the controller are as follows:

```
init    ord_g = false ∧ o_req = false ∧ bt_g = true ∧ bt_confirm
        = false ∧ slot_g = false ∧ s_req = false ∧ c_item = NULL
        ∧ ac = true
actions
        button_select(id: int) [bt_g,c_item,bt_confirm] :
        bt_g ∨ ac, false -> bt_g′ = false ∧ c_item′ = c_item *
        b_item [id] ∧ bt_confirm′ = true
    [}  button_confirm[bt_confirm,slot_g,ac] : bt_confirm, false
        -> bt_confirm′ = false ∧ slot_g′ = true ∧ ac′ = false
    []  button_cancel[bt_confirm,ac,bt_g,c_item] : bt_confirm,
        false -> bt_g’ = true ∧ ac’ = true ∧ bt_confirm’ = false ∧
        c_item’ = NULL
    []  order_ret[o_req,bt_g,c_item,ac]: o_req, false -> o_req’ =
        false ∧ bt_g’ = true ∧ c_item’ = NULL ∧ ac’ = true
```

We call the extended version of the controller component controller’. It is easy to determine that controller’ satisfies the new requirement. After the customer selects an item button, the enabling guards of button_select actions will remain true because ac

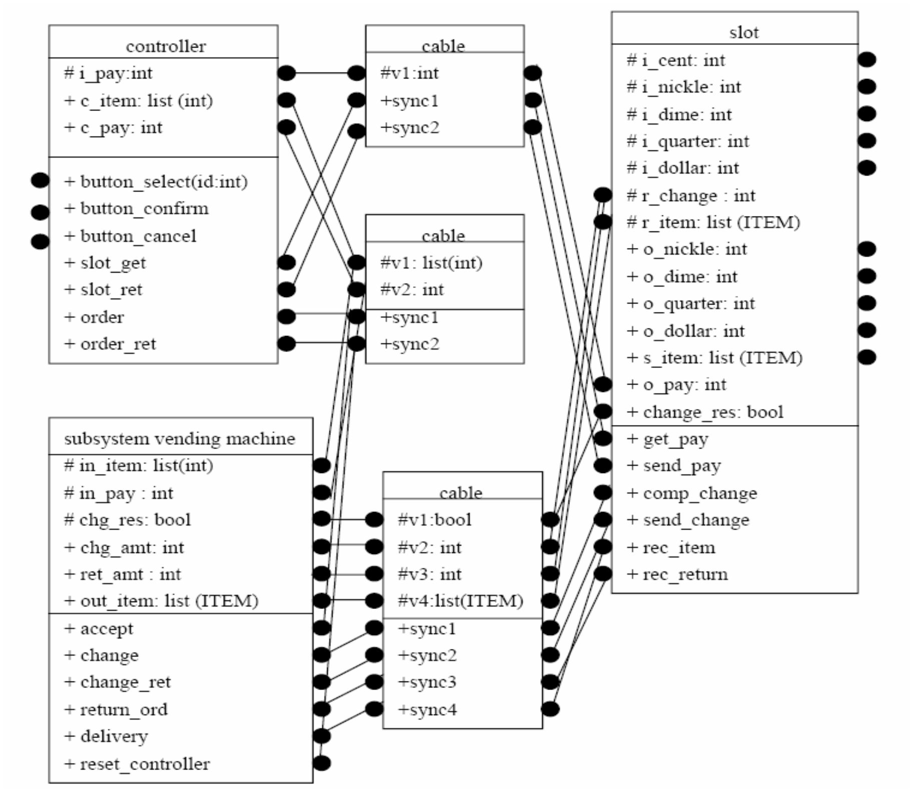


Fig. 7. Configuration diagram of the vending machine system

is true. They will not be disabled until the customer selects the confirm button, and after the vending machine subsystem informs the controller that the order has been processed by calling the order_ret action, all the item buttons will be reset.

We need to show that there exists an extension morphism from the old controller component (say P_1) to controller' (say P_2). The morphism σ is defined as follows: the mapping of the channels σ_α will map each channels of P_1 to the identical channel of P_2, and σ_γ defines the mapping of actions from each action in P_2, to the identical action in P_1.

Lemma 1. σ is an extension morphism from P_1 to P_2.

Proof
First we will show that σ is a signature morphism. Since the mappings of channels and actions are the identity, it is easy to see that all the conditions of a signature morphism are satisfied, except possibly for the condition $\sigma_\alpha(D_1(\sigma_\gamma(g))) \subseteq D_2(g)$. Since the actions in P_2 keep the effect of assignment to the mapped channels of P_1, this condition also holds. Therefore, σ is a signature morphism.

Then we will check the conditions of extension morphisms according to definition 14:

- Obviously σ_γ is surjective and σ_α is injective.
- There exists a formula α, which contains only channels from $(V_2 - \sigma_\alpha(V_1))$, and α is satisfiable, t $I_2 \Leftrightarrow \sigma(I_1) \wedge \alpha$. As $\alpha \Leftrightarrow$ ac = true, this condition holds.

For every $g \in \Gamma_2$ where $\sigma_\gamma(g)$ is defined,

- If $v \in loc(V_1)$ and $g \in D_2(\sigma_\alpha(v))$, then there exists a formula α, which contains only primed channels from $(V_2' - \sigma_\alpha(V_1)')$, and α is satisfiable, t $\sigma(L_1(\sigma_\gamma(g))) \Rightarrow (R_2(g, \sigma_\alpha(v)) \Leftrightarrow \sigma_\alpha(R_1(\sigma_\gamma(g), v)) \wedge \alpha)$.
  - For action button_confirm, $\alpha \Leftrightarrow$ ac' = false, this condition holds.
  - For action button_cancel, $\alpha \Leftrightarrow$ ac' = true, this condition holds.
  - For action order_ret, $\alpha \Leftrightarrow$ ac' = true, this condition holds.
- If $v \in loc(V_1)$, $g \in D_2(\sigma_\alpha(v))$, then $v \in D_1(\sigma_\gamma(g))$. Since the mappings of channels and actions are the identity and the actions in P_2 maintain the effect of assignments to the mapped channels of P_1, this condition will hold.

t $(\sigma(L_1(\sigma_\gamma(g))) \Rightarrow L_2(g))$. For each action button_select(id: int), bt_g $\Rightarrow$ bt_g $\vee$ ac.
t $(\sigma(U_1(\sigma_\gamma(g))) \Rightarrow U_2(g))$. The progress guards of each mapped action are the same.

Lemma 2. The new functional requirement cannot be achieved by regulating or refining the controller component.

Proof
The enabling guards of these button_select actions cannot be strengthened because, in that case, all the buttons will be disabled after the customer selects one item button. The justification for this statement is as follows:

Suppose we have regulated or refined the controller component, then, in the target component, the enabling guards of the button_select actions will be strengthened; say one of the actions is g, its enabling guard is f and f $\Rightarrow \sigma$(bt_g) (bt_g must be translated). According to the definition of regulative superposition and refinement morphism, we have $R_2(g, \sigma(\text{bt\_g})) \Rightarrow \sigma(R_1(\sigma_\gamma(g), \text{bt\_g}))$. Since bt_g is set to false after the button_select action is called in the old controller, we know that σ(bt_g) should also be set to false after the execution of g in the extended controller. Because we have f $\Rightarrow \sigma$(bt_g) and it should hold all the time, if σ(bt_g)' is false, we know f' must be false. Therefore, after the button_select action is executed in the target component, this action will be blocked, which means this item button is disabled.

3.3.2 The Extension of Payment Options

We expect that instead of only accepting payment consisting of nickels, dimes, quarters and loonies, the vending machine system can also accept payment including one cent pieces and make the correct change. It is clear that we cannot refine or regulate component slot to achieve this goal, because we must modify its action get_pay and relax its enabling guard, which is not allowed in regulative superpositions and refinement morphisms. Therefore, we have to apply an extension morphism to the slot by modifying the get_pay action as follows and obtain the extended slot component.

```
get_pay [get_g, s_nickle, s_dime, s_quarter, s_dollar, s_cent]:
get_g, false -> get_g′ = false ∧ s_nickle′ = s_nickle + i_nickle
∧ s_dime′ = s_dime + i_dime ∧ s_quarter′ = s_quarter + i_quarter
∧ s_dollar′ = s_dollar + i_dollar ∧ s_cent′ = s_cent + i_cent
```

Notice that we need to add a new channel s_cent into the slot component to store the cents and make the corresponding assignment to this channel. However, based on the definition of extension morphism, there will exist an extension morphism from the old component to this extended component (the proof is similar to the case above). For the same reason, we can modify the following actions of the slot component as well (and add the channel o_cent):

```
send_pay [get_g, o_pay]:
¬ get_g , false -> get_g′ = true ∧ o_pay′ = 100*i_dollar +
25*i_quarter + 10*i_dime + 5*i_nickel + i_cent

send_change [change_g]:
¬change_g, false -> change_g′ = true ∧ ( change_res = true ⇒
item_g′ = true ∧ s_nickle′ = s_nickle - o_nickle ∧ s_dime′ =
s_dime - o_dime ∧ s_quarter′ = s_quarter - o_quarter ∧
s_dollar′ = s_dollar - o_dollar ∧ s_cent′ = s_cent – o_cent)
```

Since we have divided the functionality of the system in an appropriate way, we can simply reuse the vending machine subsystem and the controller component.

4 Conclusions

Extension morphisms were originally motivated by their use in the application of aspects [8]. The examples developed in [8] were related to the application of a monitoring and a performance aspect to an unreliable communication system. We would like to impose behaviour on the existing architecture of an unreliable medium between a sender and receiver, to make the communication reliable by implementing a reset in the communication when packets are lost. The mechanism we used was very simple, and required a "reset" operation in the sender, which can be achieved by component extension. In order to complete the enhanced architecture to implement the reset acknowledgement mechanism, we need a monitor that, if it detects a missing packet, issues a call for reset. The idea is that, if a message is not what the monitor expected, then it will go to a "reset" cycle, and wait to see if the expected packet arrives. If the expected packet arrives, then the component will start waiting for the next packet. (Note that, since the superposed monitor is *spectative*, i.e., it has no effect on the underlying component – simply "observing" it.) Because of the properties of extension, we can guarantee that, if the augmented system works without the need for reset in the communication, i.e., no messages are lost, then its behaviour is exactly the same as the one of the original architecture with unreliable communication.

As we hope to have demonstrated in this paper, extension morphisms have a life of their own, independently of their usefulness in defining some aspects. They provide an interesting and predictable mechanism for software architects interested in change and evolution of their designs.

We have said very little about the new features of DynaComm, as this was unnecessary to talk about extensions. However, many aspects cannot be dealt with without hierarchical designs incorporating subsystem specific dynamic reconfigurability. This is what DynaComm sets out to provide. It also offers mechanisms to make design of architectures easier, such as the idea of indexed actions: a family of actions that "do the same thing" but to different elements, which can be indexed via a finite set of names. We are developing a DynaComm Workbench on the basis of experience with the CommUnity Workbench [37]. We are also putting together a catalogue of aspects and methods for developing formalizations of them. In particular, we are interested in reasoning about the applications of aspects to architectures to provide analyses for systems built in this way.

References

1. Aguirre, N., Maibaum, T.: A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In: ASE 2002, pp. 271–274 (2002)
2. Aguirre, N., Maibaum, T.: A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In: Pezzé, M. (ed.) ETAPS 2003 and FASE 2003. LNCS, vol. 2621, pp. 37–51. Springer, Heidelberg (2003)
3. Aguirre, N., Maibaum, T.: Some Institutional Requirements for Temporal Reasoning on Dynamic Reconfiguration of Component Based Systems, Verification: Theory and Practice 2003, 407–435 (2003)
4. Aguirre, N.: A Logical Basis For the Specification of Reconfigurable Component Based Systems, Ph.D. Thesis, King's College London, Department of Computer Science (2004)
5. Aguirre, N., Alencar, P., Maibaum, T.: Aspect Modularity in a High-level Program Design Language. In: CASCON Workshop on Aspect Oriented Software Development, IBM (2005)
6. Aguirre, N., Regis, G., Maibaum, T.: Verifying Temporal Properties of CommUnity Designs. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, Springer, Heidelberg (2007)
7. Aguirre, N., Maibaum, T., Alencar, P.: Abstract Design with Aspects (submitted, 2007)
8. Aguirre, N., Maibaum, T., Alencar, P.: Extension Morphisms for CommUnity, Essays Dedicated to Joseph A. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 173–193. Springer, Heidelberg (2006)
9. Allen, R., Garlan, D.: Formalizing Architectural Connections. In: ICSE'94, IEEE CS Press, Los Alamitos (1994)
10. Allen, R.J.: A Formal Approach to Software Architecture, Ph.D. Thesis, Carnegie Mellon University, School of Computer Science, available as TR# CMU-CS-97-144 (May 1997)
11. Allen, R., Douence, R., Garlan, D.: Specifying and Analyzing Dynamic Software Architectures. In: Astesiano, E. (ed.) ETAPS 1998 and FASE 1998. LNCS, vol. 1382, pp. 21–37. Springer, Heidelberg (1998)
12. Bicarregui, J.C., Lano, K.C., Maibaum, T.: Towards a Compositional Interpretation of Object Diagrams, Algorithmic Languages and Calculi, pp. 187–207. Chapman & Hall, Sydney, Australia (1997)
13. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications, Workshop on Self-Healing Systems, ACM Digital Library (2004)

14. Corradini, A., Hirsch, D.: An Operational Semantics of COMMUITY Based on Graph Transformation Systems. Electr. Notes Theor. Comput. Sci. 109, 111–124 (2004)
15. Fiadeiro, J.L., Maibaum, T.: Categorical Semantics of Parallel Program Design, Technical Report, FCUL and Imperial College (1995)
16. Fiadeiro, J.L., Maibaum, T.: Design Structures for Object Based System, Formal Methods and Object Technology, pp. 183–204. Springer, Heidelberg (1996)
17. Fiadeiro, J.L., Maibaum, T.: Interconnecting Formalisms: Supporting Modularity, Reuse and Incrementality. In: FSE, pp. 72–80 (1995)
18. Fiadeiro, J.L.: Categories for Software Engineering. Springer, Heidelberg (2005)
19. Garlan, D., Monroe, R., Wile, D.: ACME: An Architecture Description Interchange Language. In: CASCON'97 (1997)
20. Garlan, D.: Software Architecture: A Roadmap, The Future of Software Engineering. In: Filkenstein, A. (ed.), ACM Press, New York (2000)
21. Georgiadis, I.: Self-Organising Distributed Component Software Architectures, Ph.D. Thesis, Imperial College of Science, Technology and Medicine, Department of Computing (2002)
22. Goguen, J.: Mathematical Representation of Hierarchically Organised Systems. In: Attimger, E. (ed.) Global Systems Dynamics, Krager, pp. 112–128 (1971)
23. Goguen, J., Ginali, S.: A Categorical Approach to General Systems Theory. In: Klir, G. (ed.) Applied General Systems Research, pp. 257–270. Plenum, New York (1978)
24. Goguen, J.: Categorical Foundations for General Systems Theory. In: Pichler, F., Trappl, R. (eds.) Advances in Cybernetics and Systems Research, Transcripta Books, pp. 121–130 (1973)
25. Kiczales, G.: An overview of Aspect J. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, Springer, Heidelberg (2001)
26. Larsen, K.G., Skou, A.: Bisimulation through probabilistic testing. Information and Computation 94, 1–28 (1991)
27. Ling, X.: DynaComm: The Extension of CommUnity to Support Dynamic Reconfiguration, MSc Thesis, McMaster University, available as SQRL Technical Report 40 (2007), http://www.cas.mcmaster.ca/sqrl/sqrl_reports.html
28. Liskov, B., Wing, J.: A Behavioral Notion of Subtyping, ACM Transactions on Programming Languages and Systems, vol. 16(6). ACM Press, New York (1994)
29. Lopes, A., Wermelinger, M., Fiadeiro, J.: Higher-Order Architectural Connectors. ACM Transactions on Software Engineering and Methodology 12(1) (2003)
30. Lopes, A., Fiadeiro, J.: Superposition: Composition vs. Refinement of Non-Deterministic, Action-Based Systems, Formal Aspects of Computing, vol. 16(1). Springer, Heidelberg (2004)
31. Magee, J., Kramer, J.: Dynamic Structure in Software Architectures. In: Gollmann, D. (ed.) Fast Software Encryption. LNCS, vol. 1039, pp. 24–32. Springer, Heidelberg (1996)
32. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, Heidelberg (1991)
33. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. on Software Engineering 26(1), 70–93 (2000)
34. Perry, D.E., Wolf, A.L.: Foundations for the study of software architectures. SIGSOFT Software Eng. Notes 17(4), 40–52 (1992)
35. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, Englewood Cliffs (1996)

36. Szyperski, C., Pfister, C.: Component–Oriented Programming: WCOP '96 Workshop Report, In: Cointe, P. (ed.) ECOOP 1996. LNCS, vol. 1098, pp. 127–130. Springer, Heidelberg (1996)
37. Wermelinger, M., Oliveira, C.: The CommUnity Workbench, ICSE. ACM Press, New York (2002)
38. Wile, D.S.: Using Dynamic Acme, Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia (2001)
39. Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design: //www.early-aspects.net/