A Categorical Semantics of Event-based Architectures

José Luiz Fiadeiro[1] and Antónia Lopes[2]

[1] Department of Computer Science, University of Leicester
University Road, Leicester LE1 7RH, UK
jose@mcs.le.ac.uk

[2] Department of Informatics, Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa, PORTUGAL
mal@di.fc.ul.pt

Abstract. We propose a mathematical semantics for event-based architectures that serves two main purposes: to characterise the modularisation properties that result from the algebraic structures induced on systems by this discipline of coordination; and to further validate and extend the categorical approach to architectural modelling that we have been building around the language CommUnity with the "implicit invocation", aka "publish/subscribe" architectural style. We then use this formalisation to bring together synchronous and asynchronous interactions within the same modelling approach. We see this effort as a first step towards a form of engineering of architectural styles. Our approach adopts transition systems extended with events as a mathematical model of implicit invocation, and a family of logics that support abstract levels of modelling.

1 Introduction

Event-based interactions are now established as a major paradigm for large-scale distributed applications (e.g. [2,4,7,13,19]). In this paradigm, components may declare their interest in being notified when certain events are published by other components of the system. Typically, components publish events in order to inform their environment that something has occurred that is relevant for the behaviour of the entire system. Events can be generated either in the internal state of the components or in the state of other components with which they interact.

Although Sullivan and Notkin's seminal paper [21] focuses on tool integration and software evolution, the paradigm is much more general: components can be all sorts of run-time entities. What is important is that components do not know the identity, or even the existence, of the publishers of the events they subscribe, or the subscribers of the events that they publish. In particular, event notification and propagation are performed asynchronously, i.e. the publisher cannot be prevented from generating an event by the fact that given subscribers are not ready to react to the notification.

Event-based interaction has also been recognised as an "abstract" architectural style, i.e. as a means of coordinating the behaviour of components during high-level

design. The advantages of adopting such a style so early in the development process stem from exactly the same properties recognised for middleware: loose coupling allows better control on the structural and behavioural complexity of the application domain; domain components can be modelled independently and easily integrated or removed without disturbing the whole system. These very claims can be found precisely in [21] applied to the development of integrated environments but, as already mentioned, they should derive from the structural properties that the paradigm induces in more general classes of systems.

However, in spite of these advantages and its wide acceptance, implicit invocation remains relatively poorly understood. In particular, its structural properties as an architectural style remain to be clearly stated and formally verified. One has to acknowledge the merit of several efforts towards providing methodological principles and formal semantics (e.g. [5]), including recent incursions on using model-checking techniques for reasoning about such systems [3,12]. However, we are still far from an accepted "canonical" semantic model over which all these efforts can be brought together to provide effective support and formulate methodological principles that can steer development independently of specific choices of middleware.

This paper is an extended version of [10] in which we presented initial contributions in this direction by investigating how event-based interactions can be formalised in a categorical setting similar to the one that we started developing in [9] around the language CommUnity. Like for CommUnity, our goal is not to provide a full-fledged architectural description language but to restrict ourselves to a core set of primitives and a reduced notation that can capture the bare essentials of an architectural style. The use of category theory [8] is justified by the fact that it provides a mathematical toolbox geared to formalising notions of structure, namely those that arise in system modelling in a wide sense [14].

Herein, we take this formalisation forwards and address two different but interrelated aspects of event-based architectures. On the one hand, we provide a mathematical model of the computational aspects using a new extension of transition systems with event publication, notification and handling. On the other hand, we address "implicit-invocation" as a discipline of decomposition and interconnection, i.e. we characterise the modularisation properties that result from the algebraic structures induced on systems by this discipline of coordination. In particular, we justify a claim made in [21] about the externalisation of mediators: "Applying this approach yields a system composed of a set of independent and visible [tool] components plus a set of separate, or *externalised*, integration components, which we call *mediators*". Our interest is in investigating and assigning a formal meaning to notions such as "independent", "separate" and "externalised", and in characterising the way they can be derived from implicit invocation. Finally, we use the proposed formal model for investigating extensions of event-based interactions with i/o-communication and action synchronisation (rendez-vous) as available in CommUnity. We see this as a first step towards a formal approach to the engineering of architectural styles.

In Section 2, we introduce our primitives for modelling publish/subscribe interactions using a minimal language in the style of CommUnity that we call e-CommUnity. In Section 3, we provide a mathematical semantics for this language

that is based on transition systems extended with events, including their publication and handling. In Section 4, we define the category over which we formalise architectural properties. We show how the notion of morphism can be used to identify components within systems and the way they can subscribe events published by other components. We also show how event bindings can be externalised and made explicit in configuration diagrams. Section 5 defines the notion of refinement through which under-specification may be removed from designs and investigates compositionality of refinement wrt superposition as captured through the morphisms studied in Section 4. Finally, in Section 6, we analyse how we can use the categorical formalisation to combine event-based interactions with synchronous communication, namely i/o interconnections and action synchronisation as available in CommUnity. After the references, we provide a glossary that collects the different terms used in the definition of the syntax and semantics of e-CommUnity. The proofs of the main results of Section 5 are provided in an Appendix.

2 Event-based Designs

In e-CommUnity, we model components that keep a local state and perform services that can change their state and publish given events. Although we do intend to address service-oriented architectures at a later stage, this paper is concerned only with event-based interactions in general. Therefore, we will use the term service in a rather loose way, i.e. without committing to a full service-oriented approach in sense of, say, web-services [1] or wide-area computing [20].

Components can also subscribe to a number of events. Upon notification that a subscribed event has taken place, a component invokes one or more services. If, when scheduled for execution, a service is enabled, it is executed, which may change the local state of the component and publish new events.

We start discussing our approach by showing how we can model what is considered to be the "canonical" example of event-based interactions: the set-counter. Consider first the design of a component *Set* that keeps a set *elems* of natural numbers as part of its local state. This component subscribes two kinds of events – *doInsert* and *doDelete* – each of which carries a natural number as a parameter. Two other kinds of events – *inserted* and *deleted* – are published by *Set*. Each of these events also carries a natural number as a parameter.

As a component, *Set* can perform two kinds of services – *insert* and *delete*. These services are invoked upon notification of events *doInsert* and *doDelete*, respectively. When invoked, *insert* checks if the parameter of *doInsert* is already in *elems*; if not, it adds it to *elems* and publishes an *inserted* event with the same parameter. The invocation of *delete* has a similar behaviour.

```
design Set is
publish inserted
   par which:nat
publish deleted
   par which:nat
subscribe doInsert
   par which:nat
   invokes insert
      handledBy insert? ∧
         which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
      handledBy delete? ∧
         which=delete.lm

store elems: set(nat)
provide insert
   par lm:nat
   assignsTo elems
   guardedBy [lm∉elems,false]
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elems
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
```

We formalise the languages that are used for specifying component behaviour in Section 3, together with a denotational semantics for the underlying computational model. In the meanwhile, we provide an informal overview of all aspects involved.

- The events that the component publishes are declared under *publish*. Events are published when services execute. The way a service publishes a given event *e* is declared in the specification *provide* of the service under *effects* using the proposition *e!* to denote the publication of *e*.
- The events that the component subscribes are declared under *subscribe*. The services that can be invoked when handling such an event are declared under *invokes*. Given a service *s*, we denote its invocation by *s?* when specifying how the notification that the event has taken place is handled, which we do under *handledBy*.
- Parameter passing is made explicit through expressions used when specifying how event notifications are handled and events are published. For instance, the clause *inserted.which=lm* in the definition of the effects of *insert* means that the event *inserted* is published with its parameter *which* equal to the value of the parameter *lm* of *insert*. In a similar way, the clause *which=insert.lm* in the specification of *doInsert* means that the parameter *which* is passed on to the service *insert* with the same value as *lm*. Because we are providing high-level designs of components, we are not saying how such properties are guaranteed, i.e. which mechanism is being used for parameter passing.
- Designs can be under-specified, leaving room for further design decisions to be made during development. Therefore, we allow for arbitrary expressions to be used when specifying how parameters are passed, events are handled and services change the state.
- Under *store* we identify the state variables (*variables* for short) of the component; state is local in the sense that the services of a component cannot change the state variables of other components.
- Through *assignsTo* we identify the state variables that a service may change when it is executed, what we normally call the *write-frame* or *domain* of the

service. Notice that, because designs can be under-specified, the write-frame of a service cannot be inferred from the specification of its effects; it is possible for a specification not to state any properties of the effects that a service has on a state variable belonging to its write-frame, meaning that the specification is still open for further refinement.

- When specifying the *effects* of services, we use primes to denote the values that state variables take after the service is executed; as already mentioned, it is possible that the effects of some services are not fully specified.
- Through *guardedBy* we specify the enabling condition of a service, i.e. the set of states in which its invocation is accepted and the service is executed, implementing whichever effects are specified. The specification consists of a pair of conditions *[l,u]* such that *u* implies *l*: when false, the lower-guard *l* implies that the service is not enabled; when true, the upper-guard *u* implies that the service is enabled. For instance, the lower-guard of *insert* is *lm∉elems* meaning that the invocation of *insert* is refused when the element whose insertion is requested already belongs to the set; because the upper-guard is *false*, there is no commitment as to when the service is actually enabled. This allows us to model sets that are bounded without committing to a specific bound, as well as sets that are subject to restrictions that we may wish to refine at later stages of development or leave to be decided at run time. When the two guards are the same, we only indicate one condition – the enabling condition proper. This is the case of *delete*, which is specified to be enabled iff the element whose deletion is being requested belongs to the set.

Consider now the design of a system in which a counter subscribes *inserted* and *deleted* to count the number of elements in the set:

```
design Set&Counter is
store elems: set(nat),
  value:nat
publish&subscribe inserted
   par which:nat
   invokes inc
      handledBy inc?
publish&subscribe deleted
   par which:nat
   invokes dec
      handledBy dec?
subscribe doInsert
   par which:nat
   invokes insert
      handledBy insert? ∧
         which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
      handledBy delete? ∧
         which=delete.lm
provide insert
   par lm:nat
   assignsTo elems
   guardedBy [lm∉elems,false]
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elems
   effects  elems'=elems\{lm} ∧
     deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide dec
   assignsTo value
   effects  value'=value-1
```

This design illustrates how given events may be published and subscribed within the same component; this is the case of *inserted* and *deleted*. Indeed, there are no

restrictions as to the size and role that components may take within a system: designs address large-grained components, what are sometimes called sub-systems, not just atomic components. We will discuss the mechanisms that are available for structuring and composing systems in Section 4.

We can keep extending the system by bringing in new components that subscribe given events. For instance, we may wish to keep a record of the sum of all elements of the set by adding an adder that also subscribes *inserted* and *deleted*. This is captured by the following design:

```
design Set&Counter&Adder is
store elems: set(nat),
      value:nat, sum:nat
publish&subscribe inserted
   par which:nat
   invokes inc
      handledBy inc?
   invokes add
      handledBy add? ∧
          which=add.lm
publish&subscribe deleted
   par which:nat
   invokes dec
      handledBy dec?
   invokes sub
      handledBy sub? ∧
          which=sub.lm
subscribe doInsert
   par which:nat
   invokes insert
      handledBy insert? ∧
          which=insert.lm
subscribe doDelete
   par which:nat
   invokes delete
      handledBy delete? ∧
          which=delete.lm
provide insert
   par lm:nat
   assignsTo elems
   guardedBy [lm∉elems,false]
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elems
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
provide dec
   assignsTo value
   effects  value'=value-1
```

This example illustrates how a subscribed event can invoke more than one service and also how we can declare more than one handler for a given event subscription. For instance, the event *inserted* invokes two services – *inc*, as before, but also *add* – and uses two handlers: one handler invokes *add* and the other invokes *inc*. Because each invocation has a separate handler, they are independent in the sense that they take place at different times. This is different from declaring just one handler:

```
invokes inc, add
   handledBy inc? ∧ add? ∧ which=add.lm
```

Such a handler would require synchronous invocation of both services; this is useful when one wants to enforce given invariants, for which it may be important to make sure that separate state components are updated simultaneously. For instance, we may wish to ensure that the values of *sum* and *value* apply to the same set of elements so that we can compute the average value of the elements in the set.

As a design, *Set&Counter&Adder (SCA)* does not seem to provide any structure for the underlying system: we seem to have lost the original *Set* as an autonomous component; and where is the *Counter*? and the *Adder*? Later in the paper, we show how this system can be designed by interconnecting separate and independent components, including mediators in the sense of [21]. Before that, we formalise the notion of design and propose a mathematical semantics for event publication/subscription and service invocation/execution.

3 A Formal Model for Event-Based Designs

In order to provide a formal model for designs in e-CommUnity, we follow the categorical framework that, in previous papers, we have adopted for defining the syntax and semantics of specification and design languages [8].

3.1 Signatures

We start by formalising the language that we use for defining designs, starting with the data types and data type constructors. As seen in the examples discussed in the previous section, data structures are used for defining the computational aspects of component behaviour as well supporting interactions through parameter passing. In order to remain independent of any specific language for the definition of the data component of designs, we work over a fixed data signature $\Sigma=<D,F>$, where D is a set (of sorts) and F is a $D^* \times D$-indexed family of sets (of operations), and a collection Φ of first-order sentences specifying the functionality of the operations. We refer to this data type specification by Θ. We refrain from expanding on the algebraic aspects of data type specification, the theory of which can be found in textbooks such as [6].

From a mathematical point of view, designs are structures defined over signatures:

Definition 3.1: *A signature is a tuple $Q=<V,E,S,P,T,A,G,H>$ where*

- *V is a D-indexed family of mutually disjoint finite sets (of state variables).*
- *E is a finite set (of events).*
- *S is a finite set (of services).*
- *P assigns a D-indexed family of mutually disjoint finite sets (of parameters) to every service $s \in S$ and to every event $e \in E$.*
- *$T: E \to \{pub,sub,pubsub\}$ is a function classifying events as published (only), subscribed (only) or both published and subscribed. We denote by $Pub(E)$ the set $\{e \in E: T(e) \neq sub\}$ and by $Sub(E)$ the set $\{e \in E: T(e) \neq pub\}$.*
- *$A: S \to 2^V$ is a function returning the write-frame (or domain) of each service.*
- *H is a $Sub(E)$-indexed family of mutually disjoint finite sets (of handlers).*
- *G assigns to every subscribed event $e \in Sub(E)$ and handler $h \in H(e)$, a set $G(h) \subseteq S$ consisting of the services that can be invoked by that event through that handler.*

Every state variable v is typed with a sort – an element of D – that we denote by $sort(v)$; V_d is the set of variables whose type is d. All these sets are mutually disjoint, meaning that variables of different sorts have different names.

The mapping P defines, for every event and service, the name and the type of its parameters, i.e. $P(s)_d$ (resp. $P(e)_d$) is the set of parameters of service s (resp. event e) that are of sort d; like for variables, we use $sort(p)$ to indicate the sort of parameter p. Again, the sets $(P(s)_{d\in D})_{s\in S}$ and $(P(e)_{d\in D})_{e\in E}$ are assumed to be mutually disjoint. This is why, for ease of presentation, we used the "dot-notation" according to which the "official" name of, for instance, parameter *which* of event *inserted* is *inserted.which*.

We also assume that the sets of variables and parameters are mutually disjoint and disjoint from the sets of events, services and handlers. In other words, the same name cannot be used for different entities.

We use T to classify events as *pub* (published only), *sub* (subscribed only) or *pub-sub* (both published and subscribed). For instance, in *SCA* we have:

- $E_{SCA}=\{inserted, deleted, doInsert, doDelete\}$
- $T_{SCA}(inserted)=T_{SCA}(deleted)=pubsub;$
  $T_{SCA}(doInsert)=T_{SCA}(doDeleted)=sub$
- $Sub_{SCA}(E)=\{inserted, deleted, doInsert, doDelete\}$
- $Pub_{SCA}(E)=\{inserted, deleted\}$

And in *Set (S)* we have:

- $E_S=\{inserted, deleted, doInsert, doDelete\}$
- $T_S(inserted)=T_S(deleted)=pub;$
  $T_S(doInsert)=T_S(doDeleted)=sub$
- $Sub_S(E)=\{doInsert, doDelete\}$
- $Pub_S(E)=\{inserted, deleted\}$

For every service s, we define a set $A(s)$ – its *domain* or *write-frame* – that consists of the state variables that can be affected by instances of s. These are the variables indicated under *assignsTo*. For instance, $A_S(insert)=\{elems\}$. We extend the notation to state variables so that $A(v)$ is taken to denote the set of services that have v in their write-frame, i.e. $A(v)=\{s\in S \mid v\in A(s)\}$. Hence, $A_S(elems)=\{insert,delete\}$.

When a notification that a subscribed event has been published is received, a component reacts by invoking services. For every subscribed event e, we denote by $H(e)$ the mechanisms (handlers) through which notifications that e has occurred are handled. Each handler h defines a specific way of the component to react to the notification that e has been published, which may involve the invocation of one or more services declared in $G(h)$. For instance, $H_{SC}(inserted)$ has only one handler, which invokes *inc*, but $H_{SCA}(inserted)$ has two handlers – one invokes *inc* and the other invokes *add*. As for write-frames, we also extend the notation to services so that $G(s)$ for a service s is taken to denote the set of handlers that can invoke s regardless of the way the invocation is actually handled.

Notice that the functions A, G and H just declare the state variables and services that can be changed and invoked, respectively. The events that can be published are those in $Pub(E)$. Nothing in the signature states how state variables are changed, or

how and in which circumstances events are published or services invoked. This is left to the *body* of the design as discussed later in Section 3.3.

3.2 Interpretation structures

Signatures are interpreted over a semantic model based on transition systems in which execution steps are performed by synchronisation sets of services. Such interpretation structures require a model for the underlying data types, which we take as a Σ-algebra $\mathcal{D}$ that validates the specification Θ (see [6] for details):

- *Each data sort $d \in D$ is assigned a set $d_{\mathcal{D}}$ (the data values of sort d).*
- *Each operation $f:d_1,\ldots,d_n \to d'$ is assigned a function $f_{\mathcal{D}}:(d_1)_{\mathcal{D}} \times \ldots \times (d_n)_{\mathcal{D}} \to d'_{\mathcal{D}}$.*

The first step in the definition of our semantic domain is the construction of the language and space of states, events and services. We assume a fixed a signature $Q=<V,E,S,P,T,A,G,H>$ throughout this section.

***Definition 3.2**: A Q-space consists of:*

- *The extension Σ_Q of the data signature Σ with*
  - *For every event $e \in E$, a new sort d^e and, for every parameter $p \in P(e)_d$ of sort d, an operation $d^p:d^e \to d$.*
  - *For every subscribed event $e \in Sub(E)$, handler $h \in H(e)$ and service $s \in G(h)$ that can be invoked, an operation $inv^{h,s}:d^e \to d^s$.*
  - *For every service $s \in S$, a new sort d^s and, for every parameter $p \in P(s)_d$ of sort d, an operation $d^p:d^s \to d$.*
  - *For every service $s \in S$ and published event $e \in Pub(E)$, an operation $pub^{s,e}:d^s \to d^e$.*
- *An algebra $\mathcal{U}$ for Σ_Q that extends the Σ-algebra $\mathcal{D}$ in the sense that $\mathcal{U}$ is the reduct of $\mathcal{D}$ for the inclusion $\Sigma \subseteq \Sigma_Q$ and, in addition, assigns mutually disjoint carrier sets to services, and also to events.*

According to this definition, each event $e \in E$ and service $s \in S$ defines a sort, which we take to consist of their run-time instances. The parameters of events and services define operations that assign data values to every instance – the value that they carry when the corresponding events occur or services are invoked. Further to these operations that return data values, we define two other kinds of operations: *inv* that return the service instances invoked by every event occurrence, and *pub* that return the event instances published by every service execution.

Notice that we are defining a "declarative" or "denotational" semantics, not an operational one. That is to say, we are not saying how parameters are assigned to events/services, or how *pub*/*inv* generate instances of published events and invoked services; parameters are passed at run-time, and the specific event/service instances that *pub*/*inv* return are also determined during execution. However, from a declarative point of view, we can say that these functions exist between the sets of all possible instances that can ever take place.

All these sets and operations extend the algebra that interprets the data type specification. We assume that this extension does not affect the original algebra, i.e. it

does not interfere with the sets of data and the operations on data. In other words, $\mathcal{U}$ and $\mathcal{D}$ coincide in the interpretation that they provide for the data signature Σ.

As already mentioned, the sorts associated with events and services are populated with identifiers of their run-time instances. These are used for the definition of the execution model associated with our approach:

Definition 3.3*: An execution state consists of:*

- *A mapping VAL (valuation) that, to every data sort $d \in D$ and state variable $v \in V_d$, assigns a value $VAL(v) \in d_{\mathcal{D}}$.*
- *A set PND whose elements (pending invocations) are triples <t,h,u> where*
  - *t is an event instance, i.e. an element of $d^e_{\mathcal{U}}$ for some event $e \in Sub(E)$.*
  - *h is a handler for e, i.e. $h \in H(e)$.*
  - *u is a service instance invoked by t through h, i.e. $u = inv^{h,s}_{\mathcal{U}}(t)$ for some $s \in G(h)$.*

The proposed notion of state includes, as usual, the values that the variables take in that state – this is provided by the mapping *VAL*. Further to this, we have also provided states with information on the service invocations that are pending in that state – this is provided by the set *PND*. As discussed below, a service invocation becomes pending, and is added to *PND*, when an event is published that includes the service in its list of invocations. Each pending invocation is a structure that records both the event instance that triggered the invocation and the handler through which the invocation is controlled.

Not all pending invocations need to be selected for actual invocation in a given step; as we shall see, only a subset is chosen according to a policy that depends on the run-time platform. The subsets that can be chosen need satisfy some conditions:

Definition 3.4*: Given an execution state, a subset $INV \subseteq PDN$ of actual service invocations satisfies:*

- *For any invocation $<t,h,u> \in INV$, if $<t,h,u'> \in PND$ then $<t,h,u'> \in INV$, i.e. all pending invocations for the same event and handler must be selected together.*
- *For every pair of services s and s' and different service invocations <t,h,u> and <t',h',u'> in INV with $u \in d^s_{\mathcal{U}}$ and $u' \in d^{s'}_{\mathcal{U}}$, $A(s) \cap A(s') = \emptyset$; this means that there can be no simultaneous invocations of services that have intersecting domains, i.e. that can change the same state variable.*

That is, all the services invoked by the same event instance and controlled by the same handler need to be grouped together; this is because, as already motivated, all such invocations need to be discarded in the same state; service invocations that do not need to be discarded simultaneously should be assigned to different handlers.

Furthermore, instances of services that have intersecting domains cannot be selected together; this is because they cannot both write on the same part of the state within an (atomic) execution step. A particular case is when they are both instances of the same service: it does not make sense to fulfil two pending invocations of *inc*, corresponding to insertions of different elements, by executing the increment only once: clearly, we want the number of elements in the set to be incremented twice.

Further to the notion of execution state, we need to define the state transitions that characterise the way a system can evolve:

Definition 3.5*: An execution step consists of:*

- *Two execution states SRC (the source) and TRG (the target).*
- *A subset $INV \subseteq PDN_{SRC}$ (of actual service invocations)*
- *A set EXC of service instances. These correspond to the actual service invocations that are enabled in SRC.*
- *A set PUB whose elements are the event instances published at that step.*
- *A subset $NXT \subseteq PDN_{TRG}$ of next service invocations.*

satisfying the following properties:

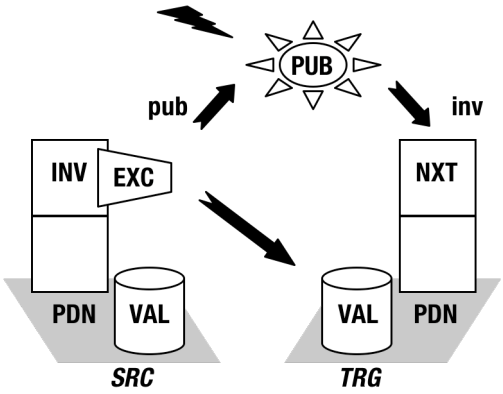
- *For every $u \in EXC$ there is $<t,h,u> \in INV$.*
- *For every $<t,h,u>$ in NXT, $t \in PUB$, i.e. only services invoked by published events can become pending.*
- *$PDN_{TRG} = PDN_{SRC} \setminus INV \cup NXT$, i.e. we discard the invoked services from the set of pending ones, and we add the set of services invoked by handlers of published events.*
- *For every $v \in V$ such that $VAL_{TRG}(v) \neq VAL_{SRC}(v)$, there is $u \in EXC$ with $u \in d^s_{//}$ such that $v \in A(s)$, i.e. a state variable can only change during an execution step if a service in its domain is executed during that step.*

The proposed notion of execution step captures the main aspects of the computational model that we are adopting. On the one hand, a number of event instances are published during an execution step – this is captured by the set *PUB*; these published events add pending invocations to the target state – this is captured by *NXT* and the functions *inv*. On the other hand, each step discards a number of the invocations pending in the source state – this is captured through *INV*. Services belonging to the discarded invocations that are enabled in the source state are executed – this is captured by the "subset" *EXC* of *INV*. The actual service executions in *EXC* are responsible for the publication of events and changes performed on the state variables in ways that are discussed in the next sub-section.

Finally, one generally assumes that the selection process is fair in the sense that invocations cannot remain pending forever; they must be eventually selected and executed if enabled. Notice that this is not a property of any individual execution step but of the global execution model; therefore, it is not captured in the definition above.

The picture below summarises some of the relationships between the entities involved in an execution step.

Note that events may be published that do not result from service execution: these instances are generated by the environment. However, all pending service invocations result necessarily from an event published in *PUB* and one of its handlers; i.e. services of a component cannot be invoked directly from the environment, only as a result of the publication of an event. A similar kind of encapsulation is enforced on the state component: a state variable can only change value if a service has been executed that includes the variable in its domain.



In summary, we are saying that interaction between a component and its environment is reduced to the publication and subscription of events: the state structures and the services that operate on them cannot be acted upon directly from outside the component. More precisely, what these encapsulation mechanisms imply is that state variables and services can only be shared together with the events that manipulate them. We shall discuss this further in later sections.

A model for a signature consists of a set of execution steps that satisfy a number of closure conditions that capture the fact that service execution is deterministic: the effects on the state and the enabling condition of every service is fully determined.

Definition 3.6: *A model of a signature Q is a Q-space $<\Sigma_Q,\mathcal{U}>$ together with a directed acyclic graph <N,R> where N is the set of nodes and R is a set of pairs of nodes (directed arcs) and a labelling function $\mathcal{L}$ that assigns an execution state to every node and an execution step to every arrow, satisfying the following conditions:*

- *For every arrow $r=<n_1,n_2>$, $\mathcal{L}(r)$ is of the form $<\mathcal{L}(n_1),\mathcal{L}(n_2),\_,\_,\_,\_>$.*
- *$VAL_{TRG}(v)=VAL_{TRG'}(v)$ for every state variable $v\in V$ and every pair of arrows $r=<n,m>$ and $r'=<n,m'>$ such that there are $s\in A(v)$ and $u\in d^s_{/\!/}\cap EXC\cap EXC'$, where $\mathcal{L}(r)$ is of the form $<\_,TRG,\_,EXC,\_,\_>$ and $\mathcal{L}(r')$ is of the form $<\_,TRG',\_,EXC',\_,\_>$.*
- *For any two arrows $r=<n,m>$ and $r'=<n,m'>$ where $\mathcal{L}(r)$ is of the form $<\_,\_,INV,EXC,\_,\_>$ and $\mathcal{L}(r')$ is of the form $<\_,\_,INV',EXC',\_,\_>$, and service instance u s.t. $<\_,\_,u>\in INV$ and $<\_,\_,u>\in INV'$, $u\in EXC$ iff $u\in EXC'$.*

Notice that, in order to improve readability, we use underscores "_" in lieu of parameters that do not play a role in the definitions or propositions.

The first condition simply means that the labelling function respects sources and targets of execution steps. The second condition means that the effects on any state variable are fully determined by the execution of an instance of a service that has the variable in its write frame. A particular case is when the execution sets of the two steps are the same, meaning that service instances have a deterministic effect on the state. The same does not apply to the publication of events because we are allowing the environment to publish events as well.

The third condition reflects the fact that the set *EXC* of service executions is fully determined by the selected invocations *INV* and the source state. Intuitively, what determines if an invoked service will be executed in the source state is what we call its enabling condition. In Section 3.3 we discuss how the lower and upper guards are interpreted as requirements on the enabling condition.

As a result, any branching in a model, i.e. the existence of more than one execution step from the same state, reflects a degree of non-determinism that results from the fact that the behaviour of the component is open to the environment.

3.3 Designs and their models

Signatures provide the "syntax" of designs. However, notice that signatures include typing information that, sometimes, is associated with "semantics" such as the encapsulation of state change and service invocation. In brief, as explained later on, signatures need to include all and only the typing information required for establishing interconnections. Hence, for instance, it is important to include in the signature information about which state variables are in the domain of which services but not the way services affect the state variables; it is equally important to know the structure of handlers for each subscribed event but not the way each subscription is handled.

The additional information that pertains to the individual behaviour of components is defined in the *bodies* of designs through three different structures, each of which involves sentences in a different language. We start with the language in which we can specify the guards of services:

Definition 3.7: *Given a signature $Q=<V,E,S,P,T,A,G,H>$ and a service $s \in S$, we define the state language $SL_{Q,s}$ associated with s as the first-order language generated by the data signature $\Sigma=<D,F>$ enriched with:*

- *for every $d \in D$, each parameter $p \in P(s)_d$ as a constant of sort d*
- *for every $d \in D$, each state variable $v \in V_d$ as a constant of sort d*

Given a valuation VAL of the state variables and an instance $u \in d^s_{\mathcal{U}}$, we evaluate the sentences of $SL_{Q,s}$ in the extension of the Σ-algebra $\mathcal{D}$ with

- $p_{\mathcal{D}} = d^p_{\mathcal{U}}(u)$
- $v_{\mathcal{D}} = VAL(v)$

That is, we extend the first-order language associated with the data signature with the parameters of the service and the state variables. We call it a "state" language because it does not concern state transitions – sentences can be evaluated on a single state, which is what is required for determining if a service is enabled. An example is the sentence *delete.lm∈elems* in the state language of *delete*; this sentence involved the parameter *delete.lm* as well as the state variable *elems*.

Consider now the language in which we specify the effects of services:

Definition 3.8: *Given a signature $Q=<V,E,S,P,T,A,G,H>$ and a service $s \in S$, we define the transition language $TL_{Q,s}$ associated with s as the first-order language generated by the data signature $\Sigma=<D,F>$ enriched with:*

- *For every $d \in D$, each parameter $p \in P(s)_d$ as a constant of sort d.*
- *For every $d \in D$, each parameter $p \in P(e)_d$ of every published event $e \in Pub(E)$ as a constant of sort d.*
- *For every $d \in D$, each state variable $v \in V_d$ as a constant of sort d.*
- *For every $d \in D$ and state variable $v \in A(s)_d$, v' as a constant of sort d.*
- *For every published event $e \in Pub(E)$, the atomic proposition $e!$.*

Given an execution step and an instance $u \in d^s_{\mathcal{U}}$, we evaluate the sentences of $TL_{Q,s}$ in the extension of the Σ-algebra $\mathcal{D}$ with:

- $p_{\mathcal{D}} = d^p_{\mathcal{U}}(u)$ *for* $p \in P(s)$.
- $p_{\mathcal{D}} = d^p_{\mathcal{U}}(\, pub^{s,e}_{\mathcal{U}}(u))$ *for* $p \in P(e)$ *and* $e \in Pub(E)$.
- $v_{\mathcal{D}} = VAL_{SRC}(v)$ *for* $v \in V$.
- $v'_{\mathcal{D}} = VAL_{TRG}(v)$ *for* $v \in A(s)$.
- $e!$ *is true iff* $pub^{s,e}_{\mathcal{U}}(u) \in PUB$.

This time, the extension includes not only the state variables and the parameters of the service but also the events that the service can publish (and their parameters) and primed versions of the state variables that belong to the domain of the service. This is because we need to be able to specify the effects of the execution of the service on the state variables, for which we use their primed versions, as well as the circumstances in which events are published, which includes the specification of how parameters are passed. Such sentences specify no longer properties of single execution states but of execution steps; this is why we call it a "transition" language.

An example is the sentence

$$(elems' = \{insert.lm\} \cup elems \wedge inserted! \wedge inserted.which = insert.lm)$$

in the transition language of *insert*. This sentence uses *elems'* to indicate that, when executed, *insert* adds its parameter to the set stored in the state variable *elem*; this is because primed variables are evaluated in the target state *TRG* of the execution step. As already mentioned, *inserted!* is used for indicating that the event *inserted* is published: such propositions specify properties of the set *PUB* associated with the execution step. Indeed, a typical sentence of the form $\psi \supset (e! \wedge \phi)$ in the transition language holds of a step for an instance u of a service s iff, when ψ is true, ϕ is also true and an event publication is added to *PUB* for the instance $pub^{s,e}_{\mathcal{U}}(u)$ of e generated by u. Notice that, typically, ψ – the pre-condition in the sense of the Hoare calculus – involves the state variables, which are evaluated at the source state, and ϕ – the post-condition – involves the primed state variables, which are evaluated in the target state *TRG,* thus establishing how the state changes as a result of the execution of the service. In the event-based approach, the post-condition includes conditions on the parameters of t and $pub^{s,e}_{\mathcal{U}}(t)$, which are evaluated in the algebra $\mathcal{U}$.

When a state sentence determines the value of a primed variable as a function of the state variables and the parameters of the service, we obtain an assignment, in which case we tend to use the notation that is normally found in programming languages – $v := F(s,v)$ for $v' = F(s,v)$.

Finally, the language in which we specify the event handlers:

Definition 3.9: *Given a signature $Q=<V,E,S,P,T,A,G,H>$ and a handler $h \in H(e)$ of an event $e \in E$, we define the handling language $HL_{Q,h}$ associated with h as the first-order language generated by the data signature $\Sigma=<D,F>$ enriched with:*

- *For every $d \in D$, each parameter $p \in P(e)_d$ as a constant of sort d.*
- *For every $d \in D$, each parameter $p \in P(s)_d$ of every service $s \in G(h)$ invoked by h as a constant of sort d.*
- *For every service $s \in G(h)$ invoked by h, the atomic proposition s?.*

Given an execution step, an instance $t \in d^e_{\mathcal{U}}$, we evaluate the sentences of $HL_{Q,h}$ in the extension of the Σ-algebra $\mathcal{D}$ with:

- $p_{\mathcal{D}} = d^p_{\mathcal{U}}(t)$ *for* $p \in P(e)$.
- $p_{\mathcal{D}} = d^p_{\mathcal{U}}(inv^{h,s}_{\mathcal{U}}(t))$ *for* $p \in P(s)$ *and* $s \in G(h)$.
- *s? is true iff* $<t,h,inv^{h,s}_{\mathcal{U}}(t)> \in NXT$.

Handling languages are not associated with services but with events and their handlers; they provide the means for specifying how the publication of the associated events are handled. A typical handling requirement is of the form $\psi \supset (s? \wedge \phi)$ establishing that s is invoked with property ϕ if condition ψ holds on notification that an instance of e has occurred. This involves the circumstances in which services are invoked, including how parameters are passed. An example, in the handling language associated with *doInsert* in *SCA*, is the sentence $(insert? \wedge doInsert.which=add.lm)$; it uses *insert?* to indicate that service *insert* is invoked when *doInsert* is published with the same parameter value as *doInsert*.

Sentences of this form specify properties of the set *NXT* of service invocations associated with the execution step. Indeed, $\psi \supset (s? \wedge \phi)$ holds for an instance t of an event e and handler h for e iff, when ψ is true, ϕ is also true and a service invocation is added to *NXT* for the instance $inv^{h,s}_{\mathcal{U}}(t)$ of s invoked by t through h. Notice that, typically, both ψ and ϕ are properties of the parameters of t and $inv^{h,s}_{\mathcal{U}}(t)$, which are evaluated in the algebra $\mathcal{U}$. This is because handling languages do not include state variables, reflecting the fact that typical publish/subscribe mechanisms do not use state information of the components to decide which services are to be invoked. However, this does not mean that the invoked services will be necessarily executed as they may not be enabled.

We can now define the notion of design:

Definition 3.10: *A design is a pair $<Q,\Delta>$ where Q is a signature and Δ, the body of the design, is a tuple $<\eta,\rho,\gamma>$ where:*

- *η assigns to every handler $h \in H(e)$ of a subscribed event $e \in Sub(E)$, a sentence in the handling language $HL_{Q,h}$ associated with h.*
- *ρ assigns to every service $s \in S$, a sentence in the transition language $TL_{Q,s}$ associated with s.*
- *γ assigns to every service $s \in S$, a pair of sentences $[\gamma^l(s),\gamma^u(s)]$ in the state language $SL_{Q,s}$ associated with s.*

Given this, the body of a design is defined in terms of:

- For every subscribed event e, a set – $H(e)$ – of *handling requirements* expressed through sentences $\eta(h)$ for every handler $h \in H(e)$. Every handling requirement

(handling for short) is enforced when the event is published. Each handler consists of service invocations and other properties that need to be observed on invocation (e.g. for parameter passing) or as a pre-condition for invocation (e.g. in the case of filters for discarding notifications).

- For every service *s*, *an enabling interval* – $[\gamma^l(s),\gamma^u(s)]$ – defining constraints on the states in which the invocation of *s* can be accepted. These are the conditions that we specify under *guardedBy*. The invocation is accepted when $\gamma^u(s)$ holds and is refused when $\gamma^l(s)$ is false.
- For every service *s*, a sentence – $\rho(s)$ – defining the *state changes* that can be observed due to the execution of *s*. As shown in the examples, this sentence may include the publication of events and parameter passing. This is the condition that we specify under *effects*.

This intuitive semantics is formalised as follows:

Definition 3.11: *A model of a design* $<Q,\Delta>$ *where* $\Delta=<\eta,\rho,\gamma>$ *is a model of Q such that any execution step* $<SRC,TRG,INV,EXC,PUB,NXT>$ *that is the label of an arrow of the underlying graph satisfies the following conditions:*

- *For every* $u\in EXC$ *with* $u\in d^s_{/\!/}$, $\gamma^l(s)$ *holds for u at SRC.*
- *For every* $<t,h,u>\in INV$ *and* $u\in d^s_{/\!/}$, *if* $\gamma^u(s)$ *holds for u at SRC then* $u\in EXC$.
- *For every* $u\in EXC$ *with* $u\in d^s_{/\!/}$, $\rho(s)$ *holds for u at that step.*
- *For every* $t\in PUB$ *where* $t\in d^e_{/\!/}$ *and* $h\in H(e)$, $\eta(h)$ *holds for t and h at that step.*

A complete execution in a model is a sequence of steps $\mathcal{F}(<n_i,m_i>)_{i\in\omega}$ *such that* $m_i=n_{i+1}$ *for every* $i\in\omega$. *We say that an execution is fair iff, for every* $i\in\omega$ *and* $<t,h,u>\in PDN_i$, *there is* $k\geq i$ *such that* $<t,h,u>\in INV_k$.

Because each model is fully deterministic apart from the possible interference of the environment, the existence of more than one model for a given design reflects under-specification. In other words, each such model reflects a possible choice of implementation. The degree of under-specification can be reduced by *refining* the design. Refinement supports a stepwise development process in which design decisions are made because requirements are made more specific, e.g. as in product-lines, or knowledge of the target run-time platform becomes more precise. This is the topic discussed in the Section 5.

4 Structuring Event-based Systems

In a categorical approach to software architecture [9,11], the structure of systems is captured through *morphisms*. These are maps between designs that identify ways in which the source is a design of a component of the system described by the target. Morphisms induce operations on models of designs that explain how the behaviour of the component can be restricted by that of the system.

4.1 Identifying components of systems

We start by defining how morphisms act on signatures:

Definition/Proposition 4.1*: A morphism $\sigma: Q_1 \rightarrow Q_2$ for $Q_1=<V_1,E_1,S_1,P_1,T_1,A_1,G_1,H_1>$ and $Q_2=<V_2,E_2,S_2,P_2,T_2,A_2,B_2,G_2,H_2>$ is a tuple $<\sigma_{st},\sigma_{ev},\sigma_{sv},\sigma_{par\text{-}ev},\sigma_{par\text{-}sv},\sigma_{hr\text{-}ev}>$ where*

- $\sigma_{st}: V_1 \rightarrow V_2$ *is a function on state variables.*
- $\sigma_{ev}: E_1 \rightarrow E_2$ *is a function on events.*
- $\sigma_{sv}: S_1 \rightarrow S_2$ *is a function on services.*
- $\sigma_{par\text{-}ev}$ *maps every event e to a function* $\sigma_{par\text{-}ev,e}: P_1(e) \rightarrow P_2(\sigma_{ev}(e))$ *on its parameters.*
- $\sigma_{par\text{-}sv}$ *is like* $\sigma_{par\text{-}ev}$ *but for service parameters, i.e.* $\sigma_{par\text{-}sv,s}: P_1(s) \rightarrow P_2(\sigma_{sv}(s))$.
- $\sigma_{hr\text{-}ev}$ *maps every subscribed event e to a function* $\sigma_{hr\text{-}ev,e}: H_1(e) \rightarrow H_2(\sigma_{ev}(e))$ *on its handlers.*

satisfying the following conditions:

- $sort_2(\sigma_{st}(v))=sort_1(v)$ *for every* $v \in V_1$, *i.e. the sorts of state variables are preserved.*
- σ_{ev} *preserves kinds, i.e.*
  - $\sigma_{ev}(e) \in Pub(E_2)$ *for every* $e \in Pub(E_1)$.
  - $\sigma_{ev}(e) \in Sub(E_2)$ *for every* $e \in Sub(E_1)$.
- $A_2(\sigma_{st}(v))=\sigma_{sv}(A_1(v))$ *for every* $v \in V_1$, *i.e. domains are preserved.*
- $\sigma_{sv}(G_1(h)) \subseteq G_2(\sigma_{hr\text{-}ev}(h))$ *for every* $e \in E_1$ *and* $h \in H_1(e)$, *i.e. services invoked by handlers carry through.*
- $\sigma_{hr\text{-}ev}(G_1(s))=G_2(\sigma_{sv}(s))$ *for every* $s \in S_1$, *i.e. invocation of services is preserved.*
- $sort_2(\sigma_{par\text{-}ev,e}(p))=sort_1(p)$ *for every* $e \in E_1$ *and* $p \in P_1(e)$, *i.e. event parameter sorts are preserved.*
- $sort_2(\sigma_{par\text{-}sv,s}(p))=sort_1(p)$ *for every* $s \in S_1$ *and* $p \in P_1(s)$, *i.e. service parameter sorts are preserved.*

Signatures and their morphisms constitute a category ***SIGN***.

A morphism σ from Q_1 to Q_2 supports the identification of a way in which a component with signature Q_1 is embedded in a larger system with signature Q_2. Morphisms map state variables, services and events of the component to corresponding state variables, services and events of the system, preserving data sorts and kinds. An example is the inclusion of *Set* in *SCA*. All the mappings are inclusions: all names used in *Set* are preserved in *SCA*.

Notice that it is possible that an event that the component subscribes is bound to an event published by some other component in the system, thus becoming *pubsub* in the system. This is why we have $T_S(inserted)=sub$ but $T_{SCA}(inserted)=pubsub$: in *SCA*, the event *inserted* is published by the service *insert*.

The constraints on domains are of the form $A_2(\sigma_{st}(v))=\sigma_{sv}(A_1(v))$ and imply that the domain in Q_2 of an "old" variable, i.e. a variable of the form $\sigma_{st}(v)$, is the image of the domain of that variable in Q_1. Therefore, new services introduced in the system cannot assign to state variables of the component. This is what makes state variables

"private" to components. The same applies to the invocation of services through the constraints $\sigma_{hr\text{-}ev}(G_1(s))=G_2(\sigma_{sv}(s))$: events subscribed by the system but not by the component cannot invoke services of the component; if other parts of the system want to invoke services of the component, they must do so by publishing events to which the component subscribes. Notice that the condition $\sigma_{sv}(G_1(h))\subseteq G_2(\sigma_{hr\text{-}ev}(h))$ allows for a subscribed event to invoke more services in the system through the same handler; however, the previous constraint implies that these new invocations cannot be for services of the component.

As a result of these encapsulation mechanisms, we cannot identify components of a system by grouping state variables, services and events in an arbitrary way; we have to make sure that variables are grouped together with all the services that can assign to them, and we have to group those services with all the events that can invoke them. For instance, we can identify a counter as a component of *SCA* that manages the state variable *value*:

```
design Counter is                store value: nat
subscribe doInc                  provide inc
   invokes inc                      assignsTo value
      handledBy inc?                effects  value'=value+1
subscribe doDec                  provide dec
   invokes dec                      assignsTo value
      handledBy dec?                effects  value'=value-1
```

If we map *doInc* to *inserted* and *doDec* to *deleted*, we do define a morphism between the signatures of *Counter* and *SCA*. Indeed, sorts of state variables are preserved, and so are the kinds of the events. The domain of the state variable *value* is also preserved because the other services available in *SCA – insert, delete, add, sub –* do not assign to it. The same applies to the invocation of its services: *inc* and *dec* are not invoked by the new events subscribed in *SCA – doInsert* and *doDelete*.

Components are meant to be "reusable" in the sense that they are designed without a specific system or class of systems in mind. In particular, it is not necessary that the components that are responsible for publishing events, as well as those that will subscribe published events, are fixed at design time. This is why, in our language, all names are local and morphisms have to account for any renamings that are necessary to establish the *bindings* that may be required. For instance, as already mentioned, the morphism that identifies *Counter* as a component of *SCA* needs to map *doInc* to *inserted* and *doDec* to *deleted*. Do notice that the binding also implies that *inserted* and *deleted* are subscribed within *SCA*. As a result, our components are independent in the sense of [21]: they do not explicitly invoke any component other than themselves.

In order to identify components in systems, the bodies of their designs also have to be taken into account, i.e. the "semantics" of the components have to be preserved. In this sense, morphisms capture relationships between designs that are similar to what in parallel program design languages are known as "superposition" [15].

Definition/Proposition 4.2*: A superposition morphism $\sigma:<Q_1,\Delta_1>\rightarrow<Q_2,\Delta_2>$ consists of a signature morphism $\sigma:Q_1\rightarrow Q_2$ such that, for every model of $<Q_2,\Delta_2>$ and execution step:*

- *Handling requirements are preserved:* $(\eta_2(\sigma_{hr\text{-}ev,e}(h)) \supseteq \underline{\sigma}(\eta_1(h)))$ *holds for every event* $e \in E_1$ *and handling* $h \in H_1(e)$.
- *Effects are preserved:* $(\rho_2(\sigma_{sv}(s)) \supseteq \underline{\sigma}(\rho_1(s)))$ *holds for every* $s \in S_1$.
- *Lower guards are preserved:* $(\gamma^l{}_2(\sigma_{sv}(s)) \supseteq \underline{\sigma}(\gamma^l{}_1(s)))$ *holds for every* $s \in S_1$.
- *Upper guards are preserved:* $(\gamma^u{}_2(\sigma_{sv}(s)) \supseteq \underline{\sigma}(\gamma^u{}_1(s)))$ *holds for every* $s \in S_1$.

Designs and their morphisms constitute a category ***sDSGN****. We denote by* ***sign*** *the forgetful functor from* ***sDSGN*** *to* ***SIGN*** *that forgets everything from designs except their signatures.*

By $\underline{\sigma}$ we denote the translations that the morphism σ induces on the languages that we use in the body of designs. The definition of such translations is quite straightforward (but tedious) by induction in the structure of the terms and sentences. See [8] for examples.

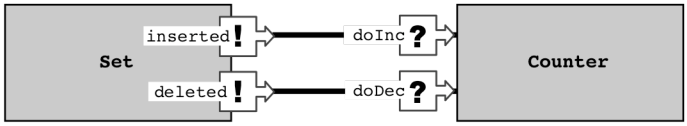
Notice that the first condition allows for more handling requirements to be added and, for each handling, subscription conditions to be strengthened. In other words, as a result of being embedded in a bigger system, a component that publishes a given event may acquire more handling requirements but also more constraints on how to handle previous requirements, for instance on how to pass new parameters.

It is easy to see that these conditions are satisfied by the signature morphisms that identify *Set* and *Counter* as components of *SCA*. However, in general, it may not be trivial to prove that a signature morphism extends to a morphism between designs. After all, such a proof corresponds to recognising a component within a system, which is likely to be a highly complex task unless we have further information on how the system was put together. This is why it is important to support an architectural approach to design through which systems are put together by interconnecting independent components. This is the topic of the Section 4.3.

4.2 Externalising the bindings

As explained in [9,11], one of the advantages of the categorical formalisation is that it allows us to support a design approach based on superposing separate components (or connectors) over independent units. These separate components are called mediators in [21]. Here we take "separate" and "independent" in the same sense as used in [21]: mediators are separate in the sense that they are components in their own right, and they interconnect components that are independent as already explained – they do not explicitly invoke any component other than themselves.

For instance, using a graphical notation for the interfaces of components – the events they publish and subscribe, and the services that they can perform – we are able to start from separate *Set* and *Counter* components and superpose, externally, the bindings through which *Counter* subscribes the events published by *Set:*



Like in [8], we explore the "graphical" nature of Category Theory to model interconnections as "boxes and lines". In our case, the lines need to be accounted for by special components that perform the bindings between the event published by one component and subscribed by the other:

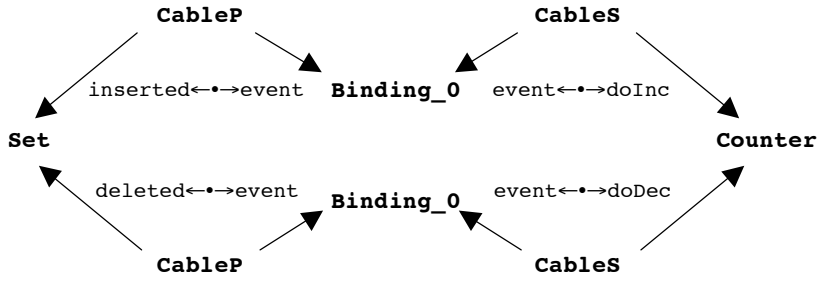
```
design Binding_0 is
publish&subscribe event
```

The binding has a single event that is both published and subscribed. The interconnection between *Set*, *Binding_0* and *Counter* is performed by an even simpler kind of component: cables that attach the bindings to the events of the components. These are of the form:

```
design CableP is
publish •
```

```
design CableS is
subscribe •
```

Because names are local, the identities of events in cables are not relevant: they are just placeholders for the projections to define the relevant bindings. Hence, we represented them through the symbol •. The configuration given above corresponds to the following diagram (labelled graph) in the category ***sDSGN*** of designs:



In Category Theory, diagrams are mathematical objects and, as such, can be manipulated in a formal way. One of the constructs that are available on certain diagrams internalises the connections in a single (composite) component. In the case above, this consists in computing the colimit of the diagram [8], which returns the design *Set&Counter* discussed in Section 2. In fact, the colimit also returns the morphisms that identify both *Set* and *Counter* as components of *Set&Counter*. We discuss these constructions in Section 4.3.

Bindings can be more complex. Just for illustration, consider the case in which we want to count only the even elements that are inserted. Instead of using *Binding_0* to connect directly *Set* and *Counter*, we would use a more elaborate connector (mediator) *Filter* defined as follows:

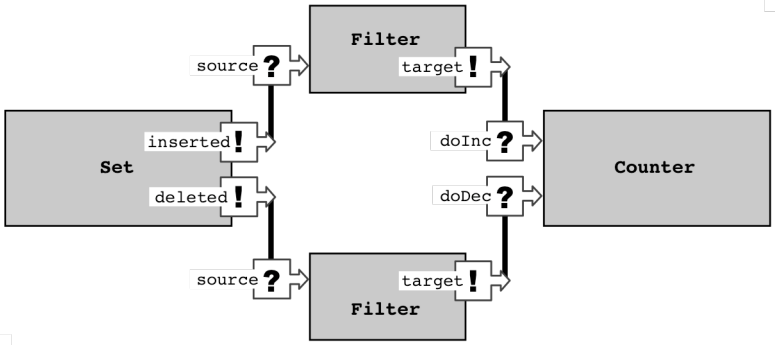
```
design Filter is
publish target
provide service
   effects target!
```

```
subscribe source
   par n:nat
   invokes service
      handledBy
         isEven(n) ≡ service?
```

This is a generic component that subscribes to an event *source* that carries a natural number, and invokes the service *source* when and only when that natural number is even. The effect of executing *source* is to publish an event *target*. That is, what we

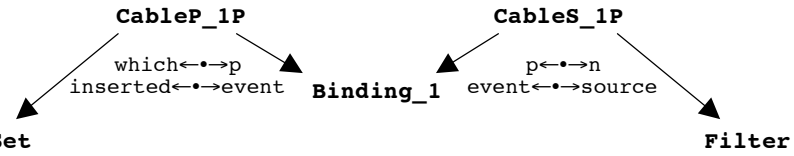
are filtering is *source* events, passing on only those that carry an even parameter. What we want now is for this filter to be connected to *inserted* events at the source, and to *doInc* at the target.

This connector is made explicit in the configuration as a mediator between *Set* and *Counter*, replacing the simple binding:



Notice that the connections between *Filter* and the other two components – *Set* and *Counter* – is still established through bindings, which we have abstracted in the picture through the same solid lines as used before. That is to say, we have, for instance:

However, the connection with *Set* requires a more sophisticated binding to ensure that the parameter is transmitted. We need



where

```
design Binding_1 is
publish&subscribe event
   par p:nat
```

```
design CableP_1P is
publish •
   par •:nat
```

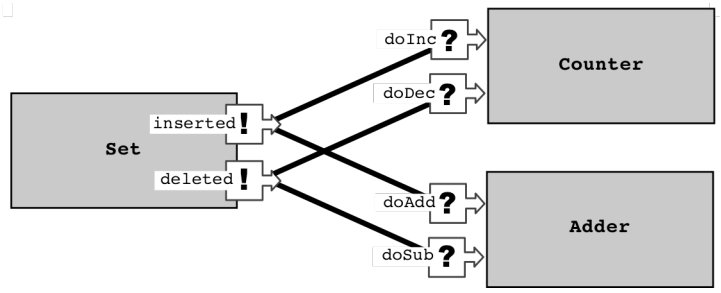
```
design CableS_1P is
subscribe •
   par •:nat
```

The other connections are established in a similar way.

The same design approach can be applied to the addition of an Adder:

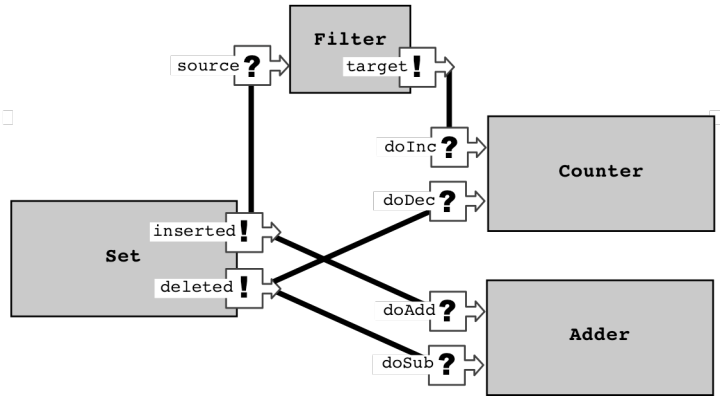
```
design Adder is
   provide add
      par lm:nat
      assignsTo sum
      effects  sum'=sum+lm
   provide sub
      par lm:nat
      assignsTo sum
      effects  sum'=sum-lm
store sum:nat
subscribe doAdd
   par which:nat
   invokes add
      handledBy add? ∧ which=add.lm
subscribe doSub
   par which:nat
   invokes sub
      handledBy sub? ∧ which=sub.lm
```

The required configuration is:



We abstain from translating the configuration to a categorical diagram. The colimit of that diagram returns the design *SCA* discussed in Section 2 and the morphisms that identify *Set, Adder* and *Counter* as components.

Notice that the categorical approach allows for systems to be reconfigured by plugging in and out bindings, components, connectors, mediators, and so on. For instance, we can superpose *Filter* to count only insertions of even numbers, or we could have superposed *Adder* to the previous configuration with *Filter*:



4.3 The universal properties of designs

We have already mentioned that we rely on so-called universal constructions, namely colimits, for giving semantics to configuration diagrams following the same principles that have been used in other areas, including CommUnity. These are operations that, when applied to a (categorical) diagram return an object (a design in our case) that captures the global behaviour of the configured system, together with the morphisms that indicate how the objects of the diagram are now components of the system. For instance, we have already mentioned how the more complex designs of Section 2 result from configurations developed in Section 4.2.

The category of e-CommUnity designs satisfies the property of being coordinated over signatures as defined in Section 3 through the (forgetful) functor that discards the information on event handling and service execution (guards and effects). This property [8] relies on the fact that every signature Q has a "canonical realisation" (a discrete lift) as a design $\boldsymbol{dsgn}(Q)$: the one that is completely under-specified. More precisely, to each service s of the signature we assign *true* to $\rho(s)$ – i.e. we make no commitments on the effects of the execution of the service – and *true* to both guards $\gamma^l(s)$ and $\gamma^u(s)$ – i.e. we make no commitments as to the bounds of the enabling condition of the service. Furthermore, we assign *true* to every handling condition $\eta(h)$ – i.e. we make no requirements on how subscribed events are handled.

This canonical realisation is such that every morphism $\sigma{:}Q \rightarrow \boldsymbol{sign}(\langle Q',\Delta'\rangle)$ is also a morphism of designs $\boldsymbol{dsgn}(Q) \rightarrow \langle Q',\Delta'\rangle$. Hence, the cables in a configuration diagram are, basically, signatures and, indeed, the calculation of a colimit takes place, essentially, in the underlying diagram of signatures: once the signature of the colimit is computed, the body is "lifted" in a canonical way from the body of the components.

The colimit construction operates over signatures by amalgamating the events involved in each pub/sub-interconnection established by the configuration. From a mathematical point of view, these events represent the quotient sets of events defined by the equivalence relation that results from the pub/sub-interconnections. The corresponding sets of parameters are amalgamated in a similar way, and so are services and their parameters.

Lifting the colimit of a diagram of signatures back to a design operates as follows. The transformations performed by an amalgamated service are specified by the conjunction of the specifications of the local effects of each of the services in the quotient set. That is, if we denote by $\{s_1,\ldots,s_n\}$ the quotient set of amalgamated services of the components, we obtain $\rho(\{s_1,\ldots,s_n\})=\underline{\sigma}_{i_1}(\rho_{i_1}(s_1))\wedge\ldots\wedge\underline{\sigma}_{i_n}(\rho_{i_n}(s_n))$ where σ_{i_j} is the signature morphism that identifies the component to which service s_j belongs within the system. Guards operate in same way, $\gamma^l(\{s_1,\ldots,s_n\})=\underline{\sigma}_{i_1}(\gamma^l{}_{i_1}(s_1))\wedge\ldots\wedge\underline{\sigma}_{i_n}(\gamma^l{}_{i_n}(s_n))$ and $\gamma^u(\{s_1,\ldots,s_n\})=\underline{\sigma}_{i_1}(\gamma^u{}_{i_1}(s_1))\wedge\ldots\wedge\underline{\sigma}_{i_n}(\gamma^u{}_{i_n}(s_n))$. The set of handlers of a subscribed event is also obtained through amalgamated sums and the handling requirement of a quotient set of handlers is also a conjunction: $\eta(\{h_1,\ldots,h_n\})=\underline{\sigma}_{i_1}(\eta_{i_1}(h_1))\wedge\ldots\wedge\underline{\sigma}_{i_n}(\eta_{i_n}(h_n))$. This explains the colimits that we have already computed in the paper for various configuration diagrams.

5 Refinement and Compositionality

In this section, we define a formal notion of *refinement* that supports incremental development by removing under-specification. As in [16], we distinguish between composition and refinement as design dimensions and formalise them through different notions of morphism, giving rise to two different but related categories of designs. We also show that this notion of refinement is compositional in the sense that designs may be refined independently of the other components and the way they are interconnected in a configuration.

5.1 Refining designs

We define the notion of refinement in much the same way as in CommUnity, i.e. by defining a notion of morphism between designs through which we can add detail and remove under-specification:

Definition/Proposition 5.1*: A refinement morphism* $\mu:<Q_1,\Delta_1>\rightarrow<Q_2,\Delta_2>$ *consists of a signature morphism* $\mu:Q_1\rightarrow Q_2$ *such that:*

- *The interface with the environment is preserved: the functions* μ_{ev}, μ_{sv}, $\mu_{par\text{-}ev,e}$, $\mu_{par\text{-}sv,s}$, $\mu_{hr\text{-}ev,e}$, *for every* $e\in E_1$ *and* $s\in S_1$, *are injective.*
- *Handling requirements are preserved:* $(\eta_2(\mu_{hr\text{-}ev,e}(h))\supseteq\underline{\mu}(\eta_1(h)))$ *holds for every event* $e\in E_1$ *and handling* $h\in H_1(e)$.
- *Effects are preserved:* $(\rho_2(\mu_{sv}(s))\supseteq\underline{\mu}(\rho_1(s)))$ *holds for every* $s\in S_1$
- *Lower guards are preserved:* $(\gamma^l{}_2(\mu_{sv}(s))\supseteq\underline{\mu}(\gamma^l{}_1(s)))$ *holds for every* $s\in S_1$
- *Upper guards are reflected:* $(\underline{\mu}(\gamma^u{}_1(s))\supseteq\gamma^u{}_2(\mu_{sv}(s)))$ *holds for every* $s\in S_1$

Designs and their refinement morphisms constitute a category ***rDSGN***.

A refinement morphism μ from designs C_1 to C_2 captures the way in which the design C_1 of a given component is refined by a more concrete design C_2 (of the same component). Although refinement morphisms are based on the same signature mappings as superposition morphisms, there are some significant differences.

- Every event and service of C_1 is represented by a distinct event and service in C_2; the same applies to the set of event and service parameters, as well as event handlers. This means that refinement preserves the interface of the component: design decisions may be made that add new events, services, parameters and handlers without collapsing them as this would change the way other components may have been connected through the more abstract design.
- The intervals provided by the guards for the enabling conditions of services are preserved in the sense that the refined interval is included in the abstract one. This means that refinement reduces the degree of under-specification on enabling conditions. Notice that superposition morphisms allow for this interval to be shifted to reflect the fact that a service shared by two components requires that both enabling conditions are true for the service to be executed.

Otherwise, the conditions on the effects of services and handling of events are the same because they reduce the degree of under-specification present in the abstract design. This reflects the fact that superposition identifies ways in which complex components share simpler components; as a result their designs may complement each other where they were under-specified.

As an example, consider the high-level design of a typical *Actuator* that provides a service *action* that can only be invoked through the publication of the event *doAction*, the publication of which guarantees that *action* is indeed invoked.

```
design Actuator is
subscribe doAction                provide action
   invokes action
      handledBy action?
```

Notice that, in this description, we do not provide any detail on what exactly the action does and when it is enabled, i.e. the execution of *action* is totally under-specified. This design can be regarded as an abstract description of *Set*. This is because if we map *doAction* to *doInc* and *action* to *inc* we define a refinement morphism from *Actuator* to *Set*. In fact, there are two ways of identifying *Set* as a refinement of *Actuator* because, if we map *doAction* to *doDec* and *action* to *dec*, we also define a refinement morphism. Similarly, *Counter* and *Adder* also refine *Actuator* in several ways.

A more informative abstract description of *Set* is provided by the design *FBActuator* that refines *Actuator* by including feedback on the execution of *action* in the form of the publication of a new event *actioned*:

```
design FBActuator is
subscribe doAction                publish actioned
   invokes action                 provide action
      handledBy action?              effects actioned!
```

Notice that this design is no longer refined by *Counter*, nor by *Adder*.

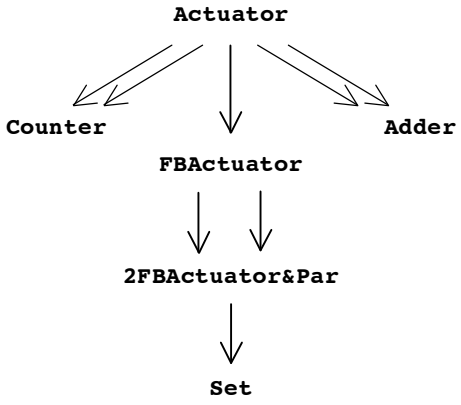
An abstraction of *Set* that is more specific in the way it can relate to other components is as follows:

```
design 2FBActuator&Par is
subscribe doAction1                    publish actioned1
   par par:nat                            par par:nat
   invokes action1                     publish actioned2
      handledBy action1? ∧                par par:nat
         action1.which=par             provide action1
subscribe doAction2                       par which:nat
   par par:nat                            effects actioned1! ∧
   invokes action2                              actioned1.par=which
      handledBy action2? ∧               provide action2
         action2.which=par                  par which:nat
                                            effects actioned2! ∧
                                                  actioned2.par=which
```

Apart from the state variables, this design has the same signature as *Set* up to renaming. As result, if offers the same interactions to the environment as *Set* but is more abstract in the sense that it does not specify its state component.

Refinement morphisms support the definition of hierarchies of "kinds" or classes of components, which is useful for defining architectural connectors as illustrated in

[11]. Notice that, in order to represent refinement morphisms in diagrams, we use an arrow that is different from the one that we use for superposition.



5.2 Reducts

Refinement morphisms act on models of the corresponding designs through what is usually called a *reduct* mapping (or just reduct, for short). The definition of a reduct requires that we are able to relate the semantic structures of both designs. In the sequel, we assume a fixed a refinement morphism $\mu:<Q_1,\Delta_1>\rightarrow<Q_2,\Delta_2>$.

Proposition 5.2*: A data signature morphism $\underline{\mu}: \Sigma_1 \rightarrow \Sigma_2$ is defined by μ between the corresponding extensions of the data signature Σ by mapping:*

- *Every sort and operation of Σ into itself.*
- *d^e into $d^{\mu(e)}$, for every $e \in E_1$.*
- *$d^p:d^e \rightarrow d$ into $d^{\mu(p)}:d^{\mu(e)} \rightarrow d$, for every $e \in E_1$, $p \in P_1(e)_d$ and sort d in Σ.*
- *$inv^{h,s}:d^e \rightarrow d^s$ into $inv^{\mu(h),\mu(s)}:d^{\mu(e)} \rightarrow d^{\mu(s)}$, for every $e \in Sub(E_1)$, $h \in H_1(e)$ and $s \in G_1(h)$.*
- *$d^p:d^s \rightarrow d$ into $d^{\mu(p)}:d^{\mu(s)} \rightarrow d$, for every $s \in S_1$, $p \in P_1(s)_d$ and sort d in Σ.*
- *$pub^{s,e}:d^s \rightarrow d^e$ into $pub^{\mu(s),\mu(e)}:d^{\mu(s)} \rightarrow d^{\mu(e)}$, for every $s \in S_1$ and $e \in Pub(E_1)$.*

Every such signature morphism $\underline{\mu}: \Sigma_1 \rightarrow \Sigma_2$ induces a reduct functor $\_|_\mu$ from the algebras of Σ_2 to the algebras of Σ_1 [6]. Such reducts extend to spaces in the sense that, applied to a Q_2-space $<\Sigma_2, \mathcal{U}>$, we obtain $<\Sigma_2, \mathcal{U}|_\mu>$ as a Q_1-space.

We omit the proof of this result because it is quite simple. However, we would like to notice that the injectivity of the functions μ_{ev} *and* μ_{sv} is necessary to ensure that the reducts of algebras extend to spaces.

From now on, we assume a fixed Q_2-space with an algebra $\mathcal{U}$.

Definition 5.3*: The μ-reduct of a set of invocations INV, which we denote by $INV|_\mu$, is the set of triples <t,h,u> such that*

- *t is an element of $d^e_{\mathcal{U}}$ for some event $e \in \mu_{ev}(Sub(E_1))$*

- $h \in H_1(e)$
- $u = inv_{//}^{\mu(h),s}(t)$ *for some* $s \in \mu_{sv}(G_1(h))$
- $<t, \mu_{hr\text{-}ev,e}(h), u> \in INV$

That is, the set $INV|_\mu$ is obtained by "forgetting" those invocations in *INV* that result from the handling of new events, or invoke new services, or result from a new handler for an "old" event.

Definition 5.4*: The μ-reduct of an execution state* $EST=<VAL,PND>$ *for* Q_2*, which we denote by* $EST|_\mu$*, consists of:*

- *The mapping that, to every data sort* $d \in D$ *and state variable* $v \in V_{1d}$*, assigns the value* $VAL(\mu_{st}(v))$*.*
- *The μ-reduct* $PND|_\mu$ *of PND.*

That is, variables are evaluated in the reduct of a state in the same way that their translations are evaluated in the original state. In what concerns pending invocations, as mentioned before, the reduct "forgets" those that result from the handling of new events, or invoke new services, or result from a new handler for an "old" event. The following results reflects the fact that what we obtain is an execution state for the source signature.

Proposition 5.5*: The μ-reduct of an execution state EST for* Q_2*, in the sense that it satisfies the conditions of Definition 3.3, is an execution state for* Q_1*. Moreover, the μ-reduct of any set of actual service invocations of EST, in the sense that it satisfies the conditions of Definition 3.4, is a set of actual invocations of* $EST|_\mu$*.*

The proof of this result is straightforward. We can now define how reducts act on execution steps:

Definition/Proposition 5.6*: Given an execution step* $<SRC,TRG,INV,EXC,PUB,NXT>$ *of* Q_2*, its μ-reduct is the execution step* $<SRC|_\mu,TRG|_\mu,INV|_\mu,EXC|_\mu,PUB|_\mu,NXT|_\mu>$ *of* Q_1 *where*

- $EXC|_\mu$ *is the set of service instances* $u \in EXC$ *s.t.* $u \in d_{//}^{s}$ *for some* $s \in \mu_{sv}(S_1)$*, i.e. that are instances of services in* Q_1*.*
- $PUB|_\mu$ *is the set of event instances* $t \in PUB$ *s.t.* $t \in d_{//}^{e}$ *for some* $e \in \mu_{ev}(E_1)$*, i.e. that are instances of events in* Q_1*.*

In this case, we simply apply the reduct componentwise to each element of the execution step. The sets $EXC|_\mu$ of executed services and $PUB|_\mu$ of published events are obtained by "forgetting" the services and events that are not generated within Q_1.

Definition/Proposition 5.7*: Given a model* $\mathcal{M}$ *of a signature* Q_2*, its μ-reduct is the model of* Q_1 *obtained by taking the μ-reduct of the* Q_2*-space of* $\mathcal{M}$ *together with the direct acyclic graph of* $\mathcal{M}$ *and the labelling function* $\mathcal{L}|_\mu$ *that results from the application of the reduct to the labels provided by* $\mathcal{L}$ *(i.e.,* $\mathcal{L}|_\mu$ *assigns the execution state* $\mathcal{L}(n)|_\mu$ *to a node n and the execution step* $\mathcal{L}(r)|_\mu$ *to an arrow r).*

In this way, the structure of the original model is preserved. The reduct only affects the labelling of nodes and arrows, which is obtained by applying the corresponding reducts to the labels of the original model.

Finally, we enunciate the result that states that refinement morphisms are model-preserving.

***Proposition 5.8**: Given a model $\mathscr{M}$ of a design $<Q_2,\Delta_2>$ its μ-reduct $\mathscr{M}|_\mu$ is a model the design $<Q_1,\Delta_1>$.*

As required, the refinement of a design may only eliminate models, reflecting that the degree of under-specification is reduced. As a result, any refinement of a design preserves its properties.

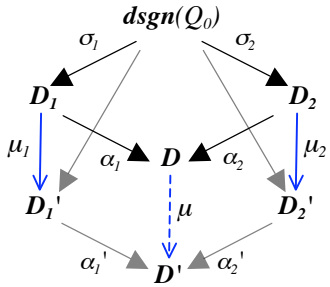
The proofs of propositions 5.6, 5.7 and 5.8 are presented in the Appendix.

5.3 Compositionality

Refinement and composition are handled through different kinds of morphisms but they can be related by a compositionality property according to which it is possible to refine designs that are part of a configuration without interfering with the other components and the interconnections that are in place. We enunciate and prove our results for a special kind of colimits – pushouts – as this simple case generalises to the colimit of any finite diagram [8].

***Proposition 5.9**: Let $<\sigma_1:$**dsgn**$(Q_0)\rightarrow D_1,\sigma_2:$**dsgn**$(Q_0)\rightarrow D_2>$ be a pair of superposition morphisms in **sDSGN** with pushout $<\alpha_1:D_1\rightarrow D,\alpha_2:D_2\rightarrow D>$. Given a pair $<\mu_1:D_1\rightarrow D_1',\mu_2:D_2\rightarrow D_2'>$ of refinement morphisms in **rDSGN**, there exists a unique refinement morphism $\mu:D\rightarrow D'$ satisfying $\alpha_1;\mu=\mu_1;\alpha_1'$ and $\alpha_2;\mu=\mu_2;\alpha_2'$ in the category **SIGN**, where $<\alpha_1':D_1'\rightarrow D',\alpha_2':D_2'\rightarrow D'>$ is the pushout of $<\sigma_1;\mu_1,\sigma_2;\mu_2>$ in **sDSGN** and $(\sigma_i;\mu_i)$ are the morphisms obtained by lifting the composition of the underlying signature morphisms to **sDSGN**.*

Notice that the fact that ***sDSGN*** is coordinated over ***SIGN*** ensures that any interconnections of designs can be established via their signatures, which is why we used ***dsgn***(Q_0) as a middle object in the given configuration. As discussed in Section 4.3, this design is a canonical realisation of a signature. The fact that this simplification does not constitute a limitation is proved in [8].



More information on the relationship between refinement and superposition, and the compositionality results that relate them can be found in [16].

6 Adding Synchronous Interactions

Another advantage of the categorical formalisation of publish/subscribe is that it allows us to use this style in conjunction with other architectural modelling techniques, namely synchronous interactions as in CommUnity. For instance, consider that we are now interested in restricting the insertion of elements in a set to keep the sum below a certain limit *LIM*. Changing the service *add* of *Adder* to:

```
provide add
   par lm:nat
   assignsTo sum
   guardedBy sum+lm<LIM
   effects  sum'=sum+lm
```

does not solve the problem because *inserted*, to which *Adder* subscribes, is published after the element has been inserted in the set. What we need is to change the service *insert* of *Set* so as to strengthen its enabling condition with *sum+lm<LIM*, and ensure that *sum* is updated by *insert* and *delete*. However, to do so within ***sDSGN***, we would have to redesign the whole system. Ideally, we would like to remain within the incremental design approach through which we superpose separate components to induce required behaviour.

One possibility is to use action synchronisation and i/o-communication as in CommUnity. More precisely, the idea is to synchronise *Set* and *Adder* to ensure that *sum* is updated when insertions and deletions are made, and superpose a regulator to check the sum before allowing the insertion invocation to proceed. In CommUnity, actions capture synchronisation sets of service invocations, something that is not intrinsic to implicit invocation as an architectural style and, therefore, cannot be expressed in the formalism presented in the previous sections. Likewise, input and output channels are needed for making sure that data is exchanged in a synchronous way. This is why we extend the notion of design in e-CommUnity with synchronisation constraints and communication channels.

As an example, consider the revision of *SCA* given below. Through the new primitive *synchronise* we provide a sentence that defines the synchronisation sets of service execution that can be observed at run-time. For instance, through the sentence $a \equiv b$, we can specify that two given services *a* and *b* are always executed simultaneously. Hence, in the example, *insert* and *add* are always performed synchronously.

Through *convey* we establish how the output channels relate to the state variables. In the example, we are just making the *sum* directly available to be read by the environment through *mysum*. The idea is that *sum* "belongs" to the adder but needs to be observed by the set in order to determine if insertions are allowed. The output channel *mysum* serves exactly this purpose, i.e. it allows a component to make data available, in a synchronous way, to other components in the same system (as above) or the environment. This is why we can strengthen the guard of *insert* with the condition *lm+mysum<LIM*.

```
design syncSet&Counter&Adder is
   store elems: set(nat),
         value:nat, sum:nat
   output mysum:nat
   publish&subscribe inserted
      par which:nat
      invokes inc
         handledBy inc?
   publish&subscribe deleted
      par which:nat
      invokes dec
         handledBy dec?
   subscribe doInsert
      par which:nat
      invokes insert
         handledBy insert? ∧
           which=insert.lm
   subscribe doDelete
      par which:nat
      invokes delete
         handledBy delete? ∧
            which=delete.lm
   synchronise insert=add ∧
          insert.lm=add.lm ∧
          sub=delete ∧
          sub.lm=delete.lm
   convey mysum=sum
```

```
provide insert
   par lm:nat
   assignsTo elems
   guardedBy
      [lm∉elems∧lm+mysum<LIM,false]
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elems
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
provide dec
   assignsTo value
   effects  value'=value-1
```

We can now formalise the extension, starting with signatures:

Definition 6.1*: We call an extended signature $Q^{I,O}$ a signature Q together with two D-indexed families I and O of mutually disjoint finite sets (of input and output channels, respectively).*

Our next step concerns the semantic model. Basically, we have to provide the structures through which we can interpret channels and synchronisation constraints. This concerns both execution states and steps.

Communication channels are interpreted over execution states by extending the valuation mappings:

Definition 6.2*: An extended execution state for an extended signature $Q^{I,O}$ is an execution state for Q with its valuation mapping VAL extended to I and O, i.e. to every data sort d∈D and channel $c \in I_d \cup O_d$, VAL assigns a value $VAL(c) \in d_{\mathscr{D}}$.*

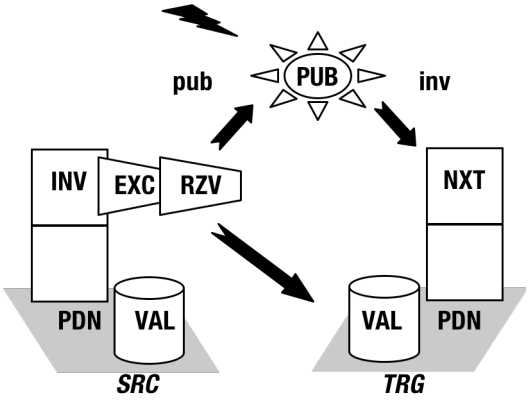
Execution steps are extended with an additional set *RZV* of service instances corresponding to the executions that result from synchronisation constraints. These additional executions have to satisfy the requirement that there can be no more than one execution of an instance of any service at any given step.

Definition 6.3*: An extended execution step for an extended signature $Q^{I,O}$ is an execution step for Q together with a set RZV of service instances such that, for every s∈S, there is at most one $u \in d^s_{\mathscr{U}} \cap RZV$.*

The set *RZV* contains the service instances that are executed during that step. It may exclude some of the instances in *EXC*, i.e. instances that have been invoked and

are enabled. This may happen, for instance, because the excluded services are synchronised with other services that are not enabled. That is, the synchronisation requirements may impose the execution of services that were not directly invoked, but they may also exclude invoked services that would otherwise be executed. However, as discussed in the definition of model, we impose a "maximality" constraint on *RZV* wrt *EXC* that makes sure that only as many enabled invocations are discarded as necessary to satisfy the synchronisation constraints.

The following picture reflects the structure of an extended execution step:



We can now define the languages over which we can specify both observation and synchronisation constraints. The former involves output channels and state variables:

Definition 6.4*: Given an extended signature $Q^{I,O}$ we define the observation language $OL_{Q,I,O}$ associated with Q as the first-order language generated by the data signature $\Sigma=<D,F>$ enriched with:*

- *For every sort $d\in D$, each output channel $o\in O_d$ as a constant of sort d.*
- *For every sort $d\in D$, each state variable $v\in V_d$ as a constant of sort d.*

Given an extended execution state, we evaluate the sentences of $OL_{Q,I,O}$ in the extension of the Σ-algebra $\mathcal{D}$ with:

- $o_{\mathcal{D}}=VAL(o)$.
- $v_{\mathcal{D}}=VAL(v)$.

The synchronisation constraints are expressed in a language that involves services and their parameters:

Definition 6.5*: Given an extended signature $Q^{I,O}$ we define the synchronisation language $SL_{Q,I,O}$ associated with Q as the first-order language generated by the data signature $\Sigma=<D,F>$ enriched with:*

- *For every $s\in S$ and $d\in D$, each parameter $p\in P(s)_d$ as a constant of sort d.*
- *For every service $s\in S$, the atomic proposition s.*

Given an extended execution step, we evaluate the sentences of $SL_{Q,I,O}$ in the extension of the Σ-algebra $\mathcal{D}$ with:

- $p_{\mathcal{D}}= d^{p}_{u}(u)$ *for $s\in S$, $p\in P(s)$ and $u\in d^{s}_{u}$ such that $u\in d^{s}_{u}\cap RZV$ if $d^{s}_{u}\cap RZV\neq\emptyset$.*
- *s is true iff $d^{s}_{u}\cap RZV\neq\emptyset$.*

We use *s* to denote the fact that an instance of service *s* is executed during a step, either in response to an invocation or as a result of a synchronisation. Notice that this is different from the invocation of *s*, which we denoted by *s?*; the invocation is evaluated over *NXT*, whereas the execution refers to *RZV*.

Finally, we extend the state and transition languages defined in Section 3 in order to allow communication channels to be used both in guards and the specification of the effects of services:

Definition 6.6*: The state and transition languages associated with* $Q^{I,O}$ *are those of Q extended with each input channel* $i \in I_d$ *as a constant of sort d. For every execution state, we extend every* Σ*-algebra* $\mathcal{D}$ *with* $i_{\mathcal{D}}=VAL(i)$ *and, for every execution step,* $i_{\mathcal{D}}=VAL_{SRC}(i)$.

Given this, we can define designs in extended signatures:

Definition 6.7*: An extended design over* $Q^{I,O}$ *is a tuple* $<\eta,\rho,\gamma,\beta,\chi>$ *where* $<\eta,\rho,\gamma>$ *is a design for Q in which I and O can be used in the languages of* ρ *and* γ*, and:*

- $\beta \in OL_{Q,I,O}$ *is a sentence establishing what observations of the local state are made available through the output channels.*
- $\chi \in SL_{Q,I,O}$ *is a sentence establishing dependencies between service execution that need to be observed at every step.*

The corresponding notion of model is:

Definition 6.8: *A model of an extended design* $<Q^{I,O},\Delta>$ *where* $\Delta=<\eta,\rho,\gamma,\beta,\chi>$ *is a model of* $Q^{I,O}$ *such that any label <SRC,TRG,INV,EXC,RZV,PUB,NXT> of an arrow of the underlying graph satisfies the following conditions:*

- *For every* $u \in EXC$ *with* $u \in d^s_{\mathcal{U}}$, $\gamma^I(s)$ *holds for u at SRC.*
- *For every* $<t,h,u> \in INV$ *and* $u \in d^s_{\mathcal{U}}$, *if* $\gamma^u(s)$ *holds for u at SRC then* $u \in EXC$.
- *For every* $u \in RZV$ *with* $u \in d^s_{\mathcal{U}}$, $\gamma^I(s)$ *holds for u at SRC.*
- *For every* $u \in RZV$ *with* $u \in d^s_{\mathcal{U}}$, $\rho(s)$ *holds for u at that step.*
- *For every* $t \in PUB$ *where* $t \in d^e_{\mathcal{U}}$ *and* $h \in H(e)$, $\eta(h)$ *holds for t and h at that step.*
- β *is true at TRG.*
- χ *is true at that step.*
- *If* $u \in EXC$ *and* $u \notin RZV$, *there is no step <SRC,_,INV,EXC,RZV',_,_> with* $RZV' \supseteq RZV$ *and* $u \in RZV'$ *such that all the previous conditions hold for that step.*
- *There is no step <SRC,_,INV,EXC,RZV',_,_> with* $RZV' \subset RZV$ *such that all the previous conditions hold for that step.*
- *If* $u \in EXC$ *and* $u \notin RZV$, *and there is a step <SRC,_,INV, EXC,RZV',_,_> such that* $u \in RZV'$ *and all the previous conditions hold for that step, than there is an arrow of the underlying graph that has the same source node and is labelled with that step.*

The first and second condition repeat what we defined for models of the original designs. The third condition is like the first but applied to *RZV*. The fourth condition repeats what is required for models of original designs but applied to *RZV* instead of *EXC*; this is because the services that are executed are those in *RZV*, which may include only some of those in *EXC*. The fifth condition is also as for original designs.

The sixth and seventh conditions are the ones that address the new sets of requirements on observations and synchronisations. The eighth condition captures, in a sense, a notion of "maximality" wrt *EXC*: invoked services that can be executed in spite of synchronisation constraints should be part of a step. The ninth condition captures a notion of "minimality" of *RZV*: no more services should be executed than those necessary for fulfilling the synchronisation constraints. Finally, the tenth condition adds to the maximality property of the eighth the fact that all options should be reflected in the same model.

Notice that, because service synchronisations are specified through a sentence in which services are used as atomic propositions, every model defines a number of sets of services – those that correspond to the propositional models of the synchronisation constraint. For instance, in a language of propositions (services) *{a,b,c}*, the (synchronisation) constraint *(a⊃b)* admits as models the subsets *{}, {c}, {b}, {b,c}, {a,b},* and *{a,b,c}*. In other words, it excludes the sets that contain *a* but not *b*.

These propositional models correspond to the synchronisation sets used for interpreting actions in CommUnity. The difference is that, in e-CommUnity, we are not synchronising actions as sets of service executions, but imposing constraints on the way these service can be executed wrt to each other. In other words, whereas by binding action names, CommUnity offers an "operational" account of synchronisation through its universal constructions, e-CommUnity is "declarative"; the bindings established in e-CommUnity through cables do not synchronise independent services, they identify them. We resume this discussion at the end of this section.

It remains to show how we can externalise the extension in much the same way we did in Section 3. The following design captures the synchronisation:

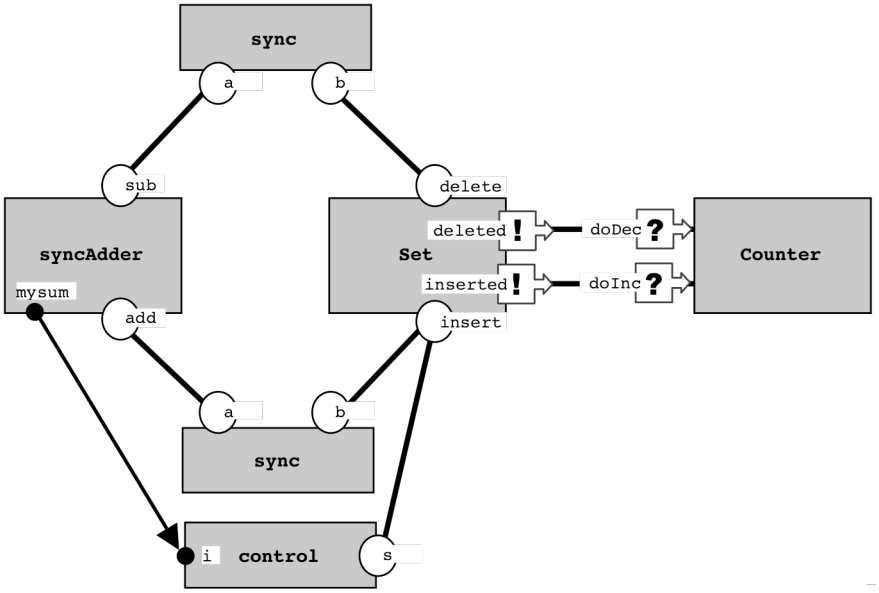
```
design sync is
synchronise a=b
     ∧ a.p=b.p
provide a
   par p:nat
provide b
   par p:nat
```

For strengthening the guard of *insert* we need a component that reads the state of *Adder* to determine if *insert* can proceed:

```
design control is
input i:nat
provide s
   par n:nat
   guardedBy n+i<LIM
```

This leads us to a new configuration in which *syncAdder* is modelled as a component prepared for synchronous interaction:

```
design syncAdder is
provide add
   par lm:nat
   assignsTo sum
   effects  sum'=sum+lm
provide sub
   par lm:nat
   assignsTo sum
   effects  sum'=sum-lm
store sum:nat
output mysum:nat
convey mysum=sum
```



Notice that *sync* and *control* are, like mediators, separate components that interconnect independent components: *syncAdder* and *Set* are unaware that they are being synchronised, and *syncAdder* does not know who is connected to its output channel; we can replace *sync* by another interaction protocol without disturbing *syncAdder* and *Set*. Therefore, we can claim that we have not increased the degree of coupling and compromised the evolutionary properties of systems by adding synchronous interactions to implicit invocations.

The proposed extension of e-CommUnity is supported by the following notion of morphism:

Definition 6.9*: A morphism σ between extended signatures $<V_1,E_1,S_1,P_1,T_1,A_1,G_1,H_1,I_1,O_1>$ and $<V_2,E_2,S_2,P_2,T_2,A_2,G_1,H_2,I_2,O_2>$ is a morphism between signatures $<V_1,E_1,S_1,P_1,T_1,A_1,G_1,H_1>$ and $<V_2,E_2,S_2,P_2,T_2,A_2,G_2,H_2>$ together with $\sigma_{in}:I_1 \rightarrow I_2 \cup O_2$ and $\sigma_{out}:O_1 \rightarrow O_2$.*

That is, as in CommUnity, input channels may become output channels of the system but not the other way around.

Definition 6.11*: A morphism between $<\eta_1,\rho_1,\gamma_1,\beta_1,\chi_1>$ and $<\eta_2,\rho_2,\gamma_2,\beta_2,\chi_2>$ is a morphism between $<\eta_1,\rho_1,\gamma_1>$ and $<\eta_2,\rho_2,\gamma_2>$ such that the observation and synchronisation dependencies are preserved: $\Phi \vdash \beta_2 \supset \underline{\sigma}(\beta_1)$ and $\Phi \vdash \chi_2 \supset \underline{\sigma}(\chi_1)$.*

Notice that this is an extension of the previous notion of morphism, i.e. morphisms between designs that do not involve communication channels and synchronisations are as before. According to this notion of morphism, synchronisation and observation dependencies can be strengthened, i.e. a system may impose new synchronisations among services of the component and new observations of the state of the component.

However, notice that, at the level of synchronisation sets, morphisms operate in a contra-variant way: the inverse image $\sigma_{ac}^{-1}(ss_2)$ of every synchronisation set ss_2 of P_2

is a synchronisation set of P_1. To understand why this is so, consider the case in which the morphism is an inclusion over services. This means that the implication $(\chi_2 \supset \chi_1)$ holds, which implies that every model (synchronisation set) of χ_2 projects to a model of χ_1 by discarding the propositions (services) that are not in the language of S_1. This contra-variant behaviour reflects the way signature morphisms were used in the previous generation of CommUnity.

In terms of the colimit construction, synchronisation dependencies are also composed as a conjunction of the dependencies of the components: $\chi = \underline{\sigma_1}(\chi_1) \wedge \ldots \wedge \underline{\sigma_n}(\chi_n)$. Again, this reflects the fact that colimits operate on synchronisation sets through fibred products: these compute intersections of inverse images of synchronisation sets of components, which is the way actions are synchronised in CommUnity.

Every design in e-CommUnity is also an extended design in a canonical way by considering that the set of communication channels is empty, and including observation and synchronisation constraints that are tautological. However, notice that this relationship does not extend to an adjunction [8]: there is an adjunction between the corresponding categories of signatures, but it does not lift to designs, much in the same way that, in logics, adjunctions between categories of signatures lift to the categories of theories but not of presentations [8].

On the other hand, every model of a design provides a model for its canonical extended design by making *RZV* equal to *EXC*. Furthermore, the maximality and minimality conditions ensure that this is the only possible choice for *RZV*: on the one hand, we cannot exclude enabled invocations from *RZV* because of the eighth condition; on the other hand we cannot add more invocations because of the ninth condition. In other words, every design and its canonical extended design have essentially the same models, meaning that the extension is "conservative" in a model-theoretic sense.

6 Conclusions and Further Work

In this paper, we presented an extended account of the formalisation of the architectural style known as "publish/subscribe" or "implicit invocation" that we started in [10]. Other formal models [e.g., 5,12] exist that abstract away from concrete notions of event and related notification mechanisms. However, they address the computational aspects of the paradigm, which is necessary for supporting, for instance, several forms of analysis. Our work addresses both the computational *and* the architectural properties of the paradigm, i.e. what concerns the way connectors can be defined and superposed over components to coordinate their interactions.

In what concerns the computational model, we proposed a mathematical semantics based on transition systems extended with publication of events and invocation of (atomic) services. As mentioned in Section 2, we are now extending this framework to address the full expressive power of conversational services in the sense of Service-Oriented Architectures [1]. We defined several logics that support high-level specification of different aspects of component behaviour. However, such logics do not support verification of properties as such; we are currently developing a modal logic

that supports analysis of several classes of properties. This modal logic semantics should also give rise to a functor that captures the way properties emerge from interconnections

In what concerns the architectural model, our formalisation allowed us to characterise key structural properties of the architectural style in what concerns the externalisation of bindings and mediators previously claimed in papers like [21]. Terms like "separate" and "independent" were given a precise interpretation in our framework, which we claim is faithful to the one given in [21]: mediators are separate components in the sense that they are defined as first-class citizens that maintain a state and can publish and subscribe events as necessary to coordinate the behaviour of other components; components remain independent in the sense that they do not invoke any other component (including mediators) other than themselves. The ability to support a design approach in which mediators can be superposed, dynamically, over such independent components derives from the externalisation of bindings. From a mathematical point of view, these properties derive from the fact that the (forgetful) functor that maps the category of designs to that of signatures has the strong structural property of being coordinated, as explained in [8].

Furthermore, the proposed categorical semantics allowed us to propose extensions to what is normally available in event-based languages. On the one hand, e-CommUnity supports under-specification and refinement, i.e. the ability to design systems in which components, mediators and their interconnections have been established but not the circumstances in which they actually publish events, how they subscribe events, or how their services operate. Refinement is the process through which we can add detail to the designs of these components in a stepwise way. We proved that this process is compositional wrt superposition, i.e. that the designs of components can be refined independently of the way they are interconnected. We believe that the separation between superposition and refinement as design dimensions is an essential one, and that compositionality results are key for any architectural style to be able to address the complexity of software development [16].

The second extension that we proposed concerns the way in which implicit invocation can be used together with synchronous forms of interconnection as previously formalised through the language CommUnity. More precisely, we added channels for (synchronous) input/output communication, and rendez-vous style of synchronisation of service executions. We showed how these new forms of interaction do not increase the degree of coupling nor compromise the evolutionary properties of implicit-invocation. In particular, we showed how synchronous interactions may themselves be externalised in separate mediators, and how communication channels are not connected through explicit naming but external bindings. Again, the proposed categorical formalisation was key for showing how all these dimensions can be brought together.

Further work is going on towards exploiting this categorical framework to support the integration of several architectural styles. For instance, we should be able to extend e-CommUnity with the primitives that we used for extending CommUnity to capture distribution and mobility [18] as well as context awareness [17]. However, we are still in the initial stages of what could be called "architectural engineering".

Acknowledgements

This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*.

References

1. G. Alonso, F. Casati, H. Kuno, V. Machiraju (2004) *Web Services*. Springer Berlin Heildelberg New York
2. J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, M. Spiteri (2000) Generic support for distributed applications. *IEEE Computer* 33(3):68–76
3. J. Bradbury, J. Dingel (2003) Evaluating and improving the automatic analysis of implicit invocation systems. In: *ESEC/FSE'03*. ACM Press, New York, pp 78–87
4. A. Carzaniga, D. Rosenblum, A. Wolf (2001) Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* 19:283–331
5. J. Dingel, D. Garlan, S. Jha, D. Notkin (1998) Towards a formal treatment of implicit invocation. *Formal Aspects of Computing* 10:193–213
6. H. Ehrig, B. Mahr (1985) *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol 6. Springer, Berlin Heidelberg New York
7. P. Eugster, P. Felber, R. Guerraoui, A-M. Kermarrec (2003) The many faces of publish/subscribe. *ACM Computing Surveys* 35(2):114–131
8. J. L. Fiadeiro (2004) *Categories for Software Engineering*. Springer, Berlin Heidelberg New York
9. J. L. Fiadeiro, A. Lopes (1997) Semantics of architectural connectors. In: M. Bidoit, M. Dauchet (eds) *TAPSOFT: Theory and Practice of Software Development. LNCS, vol 1214*. Springer, Berlin Heidelberg New York, pp 505–519
10. J. L. Fiadeiro, A. Lopes (2006) A formal approach to event-based architectures. In: L. Baresi, R. Heckel (eds) *Fundamental Aspects of Sofware Engineering. LNCS, vol 3922*. Springer, Berlin Heidelberg New York, pp 18–32
11. J. L. Fiadeiro, A. Lopes, M. Wermelinger (2003) A mathematical semantics for architectural connectors. In: R. Backhouse, J. Gibbons (eds) *Generic Programming. LNCS, vol 2793*. Springer, Berlin Heidelberg New York, pp 190–234
12. D. Garlan, S. Khersonsky, J. S. Kim (2003) Model checking publish-subscribe systems. In: T. Ball, S. Rajamani (eds) *Model Checking Software. LNCS, vol 2648*. Springer, Berlin Heidelberg New York, pp 166–180
13. D. Garlan, D. Notkin (1991) Formalizing design spaces: Implicit invocation mechanisms. In: S. Prehn, W. J. Toetenel (eds.) *VDM'91: Formal Software Development Methods. LNCS, vol 551*. Springer, Berlin Heidelberg New York, pp 31–44
14. J. Goguen (1973) Categorical foundations for general systems theory. In: F. Pichler, R. Trappl (eds) *Advances in Cybernetics and Systems Research*. Transcripta Books, New York, pp 121–130
15. S. Katz (1993) A superimposition control construct for distributed systems. *ACM TOPLAS* 15(2):337–35
16. A. Lopes, J. L. Fiadeiro (2004) Superposition: composition versus refinement of non-deterministic action-based systems. *Formal Aspects of Computing* 16(1), pp 5–18

17. A. Lopes, J. L. Fiadeiro (2005) Algebraic semantics of design abstractions for context-awareness. In: J. L. Fiadeiro, P. Mosses, F. Orejas (eds) *Algebraic Development Techniques*, *LNCS, vol 3423*, Springer, Berlin Heidelberg New York, pp 79–93
18. A. Lopes, J. L. Fiadeiro (2006) Adding mobility to software architectures. *Science of Computer Programming*. In print
19. R. Meier, V. Cahill (2002) Taxonomy of distributed event-based programming systems. In: *Proceedings of the International Workshop on Distributed Event-Based Systems*. IEEE Computer Society, Silver Spring, MD, pp 585–588
20. J. Misra, W. Cook (2006) Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modelling*. In print
21. K. Sullivan, D. Notkin (1992) Reconciling environment integration and software evolution. *ACM TOSEM* 1(3):229–268

Notation

Signatures

$A(s)$	The write-frame (or domain) of service s, i.e. the state variables that any execution of s can change. This set is declared under *assignsTo*.
$A(v)$	The set of services that can change the state variable v, $s \in A(v)$ *iff* $v \in A(s)$.
D	The set of data sorts.
E	The set of all the events either published or subscribed by a component.
F	Family of operations on data.
$G(h)$	The set of services that can be invoked through handler h. This set is declared under *invokes*.
$G(s)$	The set of handlers that can invoke s, $h \in G(s)$ *iff* $s \in G(h)$.
$H(e)$	The set of handlers that react to the notifications that e has occurred. Each handler h declares, under *invokes,* the set $G(h)$ of services that it can invoke and, under *handledBy*, the condition $\eta(h)$ that specifies how such services are invoked.
I_d	The set of input channels of sort d. A channel $i \in I_d$ is declared under *input i:d*.
O_d	The set of output channels of sort d. A channel $o \in O_d$ is declared under *output o:d*.
$P(e)_d$	The set of parameters of event e that are of sort d. A parameter $p \in P(e)_d$ is declared under *par p:d*.
$P(s)_d$	The set of parameters of service s that are of sort d. A parameter $p \in P(e)_d$ is declared under *par p:d*.
S	The set of all services of a component. Each service is declared under *publishes*.
$T(e)$	The type of event e: *pub* (published only), *sub* (subscribed only) or *pub-sub* (published and subscribed).
V_d	The set of state variables of sort d. A variable $v \in V_d$ is declared under *store v:d*.

Design bodies

β	A sentence that establishes what observations of the local state are made available through the output channels. This sentence is declared under *convey*.
$\gamma^{l}(s)$	The lower guard of service *s*, i.e. a sentence that, when false, implies that the execution of *s* is not enabled. This sentence is declared under *guardedBy* as part of a pair $[\gamma^{l}(s), \gamma^{u}(s)]$.
$\gamma^{u}(s)$	The upper guard of service *s*, i.e. a sentence that, when true, implies that the execution of *s* is enabled. This sentence is declared under *guardedBy* as part of a pair $[\gamma^{l}(s), \gamma^{u}(s)]$.
$\eta(h)$	A sentence that specifies how the services in *G(h)* are invoked by *h*. This sentence is declared under *handledBy*.
$\rho(s)$	A sentence that specifies how the execution of service *s* changes the state variables declared in *A(s)* and publishes the events declared in *B(s)*. This sentence is declared under *effects*.
χ	A sentence that establishes synchronisation dependencies on the execution of services. This sentence is declared under *synchronise*.

Semantic models

EXC	Invoked service instances that are enabled.
$inv_{//}^{h,s}(t)$	The instance of *s* invoked by handler *h* for the event instance *t*.
INV	Service invocations selected for an execution step.
NXT	Service invocations generated by an execution step.
PDN	Service invocations pending in a given state.
$pub_{//}^{s,e}(u)$	The instance of *e* published when the instance *u* of service *s* is executed.
PUB	Event instances published during an execution step.
RZV	Service instances that result from the synchronisation constraints applied to *EXC*.
SRC	Source state of an execution step.
TRG	Target state of an execution step.
VAL(v)	Value of state variable *v* in a given state.

Appendix

Definition/Proposition 5.6*: Given an execution step STP=<SRC,TRG,INV,EXC,PUB, NXT> of Q_2, its μ-reduct is the execution step <$SRC|_\mu$,$TRG|_\mu$,$INV|_\mu$,$EXC|_\mu$,$PUB|_\mu$, $NXT|_\mu$> of Q_1 where*

- *$EXC|_\mu$ is the set of service instances $u \in EXC$ s.t. $u \in d^s_{/\!/}$ for some $s \in \mu_{sv}(S_1)$, i.e. that are instances of services in Q_1.*
- *$PUB|_\mu$ is the set of event instances $t \in PUB$ s.t. $t \in d^e_{/\!/}$ for some $e \in \mu_{ev}(E_1)$, i.e. that are instances of events in Q_1.*

Proof.

We prove that $STP|_\mu$ is indeed an execution step of Q_1. This requires to prove that:

- *For every $u \in EXC|_\mu$ there is $<t,h',u> \in INV|_\mu$:*

  If $u \in EXC|_\mu$ then $u \in d^s_{/\!/}$ for some $s \in \mu_{sv}(S_1)$ and $u \in EXC$. Supose that $s' \in S_1$ and $\mu(s')=s$. Because *STP* is an execution step of Q_2, there exists $<t,h,u> \in INV$ such that:

  - $t \in d^e_{/\!/}$ for some $e \in Sub(E_2)$
  - $h \in H_2(e)$
  - $u = inv^{h,s}_{/\!/}(t)$ and $s \in G_2(h)$; this is because algebras of spaces assign disjoint carrier sets to different services and $u \in d^s_{/\!/}$, i.e. the service that invokes e must be an instance of s.

  On the one hand, from $\mu(s')=s \in G_2(h)$ and the fact that μ is a refinement morphism it follows that there is $h' \in G_1(s')$ s.t. $\mu(h')=h$. This implies that $s' \in G_1(h')$. On the other hand, $\mu(h')=h \in H_2(e)$ and the fact that μ is a refinement morphism implies that $e \in \mu(Sub(E_1))$ and, hence, there is $e' \in E_1$ s.t. $\mu(e')=e$ and $h' \in H_1(e')$. It then follows that $<t,h',u> \in INV|_\mu$.

- *For every $<t,h',u> \in NXT|_\mu$, $t \in PUB|_\mu$:*

  If $<t,h',u> \in NXT|_\mu$ then $<t,\mu(h'),u> \in NXT$ and $t \in d^e_{/\!/}$ for some $e \in \mu(Sub(E_1))$. Because *STP* is an execution step of Q_2, $t \in PUB$ and, hence, $t \in PUB|_\mu$.

- *$PDN_{TRG|\mu} = PDN_{SRC|\mu} \setminus INV|_\mu \cup NXT|_\mu$:*

  This is a simple consequence of a general result*:* for every pair A,B of sets of invocations, $(A \cup B)|_\mu = A|_\mu \cup B|_\mu$ and $(A \setminus B)|_\mu = A|_\mu \setminus B|_\mu$.

- *For every $v \in V_1$ s.t. $VAL_{TRG|\mu}(v) \neq VAL_{SRC|\mu}(v)$, there is $u \in EXC|_\mu$ with $u \in d^s_{/\!/|_\mu}$ such that $v \in A_1(s)$:*

  If $VAL_{TRG|\mu}(v) \neq VAL_{SRC|\mu}(v)$ then $VAL_{TRG}(\mu(v)) \neq VAL_{SRC}(\mu(v))$. Because *STP* is an execution step of Q_2, there is $u \in EXC$ with $u \in d^{s'}_{/\!/}$ such that $\mu(v) \in A_2(s')$. Because μ is a refinement morphism, $\mu(v) \in A_2(s')$ implies that there is $s \in S_1$ s.t. $\mu(s')=s$ and $v \in A_1(s)$. This also implies that $u \in EXC|_\mu$.

Definition/Proposition 5.7*: Given a model $\mathcal{M}$ of a signature Q_2, its μ-reduct is the model of Q_1 obtained by considering the μ-reduct of the Q_2-space of $\mathcal{M}$ together with the direct acyclic graph of $\mathcal{M}$ and the labelling function $\mathcal{A}_\mu$ that results from the application of the reduct to the labels provided by $\mathcal{A}$ (i.e., $\mathcal{A}_\mu$ assigns the execution state $\mathcal{A}(n)|_\mu$ to a node n and the execution step $\mathcal{A}(r)|_\mu$ to an arrow r).*

Proof.

We prove that $\mathcal{M}|_\mu$ is indeed a model of Q_1. This requires the proof of the following:

- *For every arrow $r=<n_1,n_2>$, $\mathcal{A}_\mu(r)$ is of the form $<\mathcal{A}_\mu(n_1),\mathcal{A}_\mu(n_2),\_,\_,\_,\_>$:*
  This follows trivially from the definition of $\mathcal{A}_\mu$ and the fact that, because $\mathcal{M}$ is a model of Q_2, $\mathcal{A}(r)$ is of the form $<\mathcal{A}(n_1),\mathcal{A}(n_2),\_,\_,\_,\_>$.
- *$VAL_{TRG|\mu}(v)=VAL_{TRG'|\mu}(v)$ for every state variable $v\in V_1$ and every pair of arrows $r=<n,m>$ and $r'=<n,m'>$ s.t. there are $s\in A_1(v)$, $u\in d^s_{\mathcal{U}|_\mu} \cap EXC|_\mu \cap EXC'|_\mu$, where $\mathcal{A}_\mu(r)$ and $\mathcal{A}_\mu(r')$ are of the form $<\_,TRG|_\mu,\_,EXC|_\mu,\_,\_>$ and $<\_,TRG'|_\mu,\_,EXC'|_\mu,\_,\_>$, respectively:*

  From the hypothesis and the fact that μ is a refinement morphism, it follows that there are $\mu(s)\in A_2(\mu(v))$, $u\in d^{\mu(s)}_{\mathcal{U}} \cap EXC \cap EXC'$, where $\mathcal{A}(r)$ and $\mathcal{A}(r')$ are of the form $<\_,TRG,\_,EXC,\_,\_>$ and $<\_,TRG',\_,EXC',\_,\_>$, respectively. Given that $\mathcal{M}$ is a model of Q_2, we have that $VAL_{TRG}(\mu(v))=VAL_{TRG}(\mu(v))$, which implies that $VAL_{TRG|\mu}(v)=VAL_{TRG'|\mu}(v)$.
- *For any two arrows $r=<n,m>$ and $r'=<n,m'>$ where $\mathcal{A}_\mu(r)$ and $\mathcal{A}_\mu(r')$ are of the form $<\_,\_,INV|_\mu,EXC|_\mu,\_,\_>$ and $<\_,\_,INV'|_\mu,EXC'|_\mu,\_,\_>$, respectively, and service instance u s.t. $<\_,\_,u>\in INV|_\mu$ and $<\_,\_,u>\in INV'|_\mu$, we have that $u\in EXC|_\mu$ iff $u\in EXC'|_\mu$:*

  The hypothesis implies that $\mathcal{A}(r)$ and $\mathcal{A}(r')$ are of the form $<\_,\_,INV,EXC,\_,\_>$ and $<\_,\_,INV',EXC',\_,\_>$, respectively. Furthermore, $<\_,\_,u>\in INV$ and $u\in d^s_{\mathcal{U}}$ for some $s\in\mu_{sv}(S_1)$. Given that $\mathcal{M}$ is a model of Q_2, $u\in EXC$ iff $u\in EXC'$. Given that $u\in d^s_{\mathcal{U}}$ for some $s\in\mu_{sv}(S_1)$, we conclude that $u\in EXC|_\mu$ iff $u\in EXC'|_\mu$.

Proposition 5.8*: Given a model $\mathcal{M}$ of a design $<Q_2,\Delta_2>$ its μ-reduct $\mathcal{M}|_\mu$ is a model of the design $<Q_1,\Delta_1>$.*

Proof.

We start by enunciating some auxiliary results related to the satisfiability of the translation of sentences in the languages $HL_{Q_1,h}$, $SL_{Q_1,s}$ and $TL_{Q_1,s}$ induced by refinement morphisms in the corresponding interpretation structures and the satisfiability of the original formulas in the corresponding reducts.

Let h be an handler of an event e in Q_1, $t\in d^e_{\mathcal{U}|_\mu}$, s a service in Q_1, $u\in d^s_{\mathcal{U}|_\mu}$, and $STP=<SRC,TRG,INV,EXC,PUB,NXT>$ an execution step for Q_2.

- *Every sentence ϕ in $HL_{Q_1,h}$ holds for t and h at $STP|_\mu$ iff $\underline{\mu}(\phi)$ holds for t and $\mu(h)$ at STP.*
- *Every sentence ϕ in $SL_{Q_1,s}$ holds for u at $SRC|_\mu$ iff $\underline{\mu}(\phi)$ holds for u at SRC.*
- *Every sentence ϕ in $TL_{Q_1,s}$ holds for u at $STP|_\mu$ iff $\underline{\mu}(\phi)$ holds for u at STP.*

In order to prove that $\mathcal{M}|_\mu$ is indeed a model of $<Q_1,\Delta_1>$, we must prove that for every execution step $STP|_\mu=<SRC|_\mu,TRG|_\mu,INV|_\mu,EXC|_\mu,PUB|_\mu,NXT|_\mu>$ that is the label of an arrow of the underlying graph, the following properties hold:

- *For every $u\in EXC|_\mu$ with $u\in d^s_{\mathcal{U}|_\mu}$, $\gamma^l{}_1(s)$ holds for u at $SRC|_\mu$:*
  If $u\in EXC|_\mu$ then $u\in d^s_{\mathcal{U}}$ for some $s\in\mu_{sv}(S_1)$ and $u\in EXC$. Given that $\mathcal{M}$ is a model of $<Q_2,\Delta_2>$ and $d^s_{\mathcal{U}|_\mu}=d^{\mu(s)}_{\mathcal{U}}$, $\gamma^l{}_2(\mu(s))$ holds for u at SRC. Given that μ is a refinement morphism, $\gamma^l{}_2(\mu_{sv}(s))\supseteq\underline{\mu}(\gamma^l{}_1(s))$ holds and, hence, $\underline{\mu}(\gamma^l{}_1(s))$ holds for u at SRC. As a consequence of the auxiliary result enunciated above, $\gamma^l{}_1(s)$ holds for u at $SRC|_\mu$.
- *For every $<t,h,u>\in INV|_\mu$ and $u\in d^s_{\mathcal{U}|_\mu}$, if $\gamma^u{}_1(s)$ holds for u at $SRC|_\mu$ then $u\in EXC|_\mu$:*
  On the one hand, if $<t,h,u>\in INV|_\mu$ and $u\in d^s_{\mathcal{U}|_\mu}$ then $<t,\mu(h),u>\in INV$ and $u\in d^s_{\mathcal{U}}$ for some $s\in\mu_{sv}(S_1)$. On the other hand, as a consequence of the auxiliary result enunciated above, if $\gamma^u{}_1(s)$ holds for u at $SRC|_\mu$ then $\underline{\mu}(\gamma^u{}_1(s))$ holds for u at SRC. Given that μ is a refinement morphism, $\underline{\mu}(\gamma^u{}_1(s))\supseteq\gamma^u{}_2(\mu_{sv}(s))$ holds and, hence, $\gamma^u{}_2(\mu_{sv}(s))$ holds for u at SRC. Given that $\mathcal{M}$ is a model of $<Q_2,\Delta_2>$ and $d^s_{\mathcal{U}|_\mu}=d^{\mu(s)}_{\mathcal{U}}$, we know that $u\in EXC$. Because $u\in d^s_{\mathcal{U}}$ for some $s\in\mu_{sv}(S_1)$, we conclude that $u\in EXC|_\mu$.
- *For every $u\in EXC|_\mu$ with $u\in d^s_{\mathcal{U}|_\mu}$, $\rho_1(s)$ holds for u at $STP|_\mu$:*
  If $u\in EXC|_\mu$ then $u\in d^s_{\mathcal{U}}$ for some $s\in\mu_{sv}(S_1)$ and $u\in EXC$. Given that $\mathcal{M}$ is a model of $<Q_2,\Delta_2>$ and $d^s_{\mathcal{U}|_\mu}=d^{\mu(s)}_{\mathcal{U}}$, $\rho_2(\mu(s))$ holds for u at STP. Given that μ is a refinement morphism, $\rho_2(\mu_{sv}(s))\supseteq\underline{\mu}(\rho_1(s))$ holds and, hence, $\underline{\mu}(\rho_1(s))$ holds for u at STP. As a consequence of the auxiliary result enunciated above, $\rho_1(s)$ holds for u at $STP|_\mu$.
- *For every $t\in PUB|_\mu$ where $t\in d^e_{\mathcal{U}|_\mu}$ and $h\in H_1(e)$, $\eta_1(h)$ holds for t and h at $STP|_\mu$:*
  If $t\in PUB|_\mu$ then $t\in d^e_{\mathcal{U}}$ for some $e\in\mu_{ev}(E_1)$ and $t\in PUB$. Moreover, because μ is a refinement morphism, $h\in H_1(e)$ implies that $\mu(h)\in H_2(\mu(e))$. Given that $\mathcal{M}$ is a model of $<Q_2,\Delta_2>$ and $d^e_{\mathcal{U}|_\mu}=d^{\mu(e)}_{\mathcal{U}}$, $\eta_2(\mu(h))$ holds for t and $\mu(h)$ at STP. Again because μ is a refinement morphism, $\eta_2(\mu(h))\supseteq\underline{\mu}(\eta_1(h))$ holds and, hence, $\underline{\mu}(\eta_1(h))$ holds for t and $\mu(h)$ at STP. As a consequence of the auxiliary result enunciated above, $\eta_1(h)$ holds for t and h at $STP|_\mu$.

***Proposition 5.9**: Let $<\sigma_1:$**dsgn**$(Q_0)\rightarrow D_1,\sigma_2:$ **dsgn**$(Q_0)\rightarrow D_2>$ be morphisms in **sDSGN** with pushout $<\alpha_1:D_1\rightarrow D,\alpha_2:D_2\rightarrow D>$. Given a pair $<\mu_1:D_1\rightarrow D_1',\mu_2:D_2\rightarrow D_2'>$ of refinement morphisms in **rDSGN**, there exists a unique refinement morphism $\mu:D\rightarrow D'$ in **rDSGN** satisfying $\alpha_1;\mu=\mu_1;\alpha_1'$ and $\alpha_2;\mu=\mu_2;\alpha_2'$ in the category **SIGN**, where $<\alpha_1':D_1'\rightarrow D',\alpha_2':D_2'\rightarrow D'>$ is the pushout of $<\sigma_1;\mu_1:$**dsgn**$(Q_0)\rightarrow D_1',\sigma_2;\mu_2:$ **dsgn**$(Q_0)\rightarrow D_2'>$ in **sDSGN** and $(\sigma_i;\mu_i)$ are the morphisms obtained by lifting the composition of the underlying signature morphisms to **sDSGN**.*

Proof

We start by noticing that there is a forgetful functor ***rsign*** from ***rDSGN*** to ***SIGN*** that forgets everything from designs except their signatures. The fact that ***dsgn***(Q_0) is a

discrete lift ensures that signature morphisms $\boldsymbol{sign}(\sigma_i);\boldsymbol{rsign}(\mu_i)$ give rise to morphisms $\sigma_i;\mu_i:\boldsymbol{dsgn}(Q_0)\rightarrow D_i'$ *in* ***sDSGN***.

Given that ***sign*** preserves pushouts, we have that $<\boldsymbol{sign}(\alpha_1),\boldsymbol{sign}(\alpha_2)>$ is a pushout of $<\boldsymbol{sign}(\sigma_1),\boldsymbol{sign}(\sigma_2)>$ in ***SIGN***. Because $<\boldsymbol{sign}(\alpha_1'),\boldsymbol{sign}(\alpha_2')>$ is a candidate for being a different pushout, from the universal property of pushouts it follows that there exists a unique morphism $\mu:\boldsymbol{sign}(D)\rightarrow\boldsymbol{sign}(D')$ in ***SIGN*** satisfying $\alpha_1;\mu=\mu_1;\alpha_1'$ and $\alpha_2;\mu=\mu_2;\alpha_2'$. It remains to prove that μ also defines a morphism $\mu:D\rightarrow D'$ *in* ***sDSGN***. We only prove that μ satisfies the conditions of refinement morphisms that do not necessarily hold for morphisms in ***sDSGN***; the other conditions follow straightforwardly.

- *The functions* μ_{ev}, μ_{sv}, $\mu_{par\text{-}ev,e}$, $\mu_{par\text{-}sv,s}$, $\mu_{hr\text{-}ev,e}$, *for every* $e\in E$ *and* $s\in S$, *are injective*:

  This is just a simple consequence of a general result about pushouts in the category of sets and functions.

- *Upper guards are reflected –* $(\underline{\mu}(\gamma^u(s))\supset\gamma^u{}'(\mu_{sv}(s)))$ *holds for every* $s\in S$:

  As explained at the end of Section 4, if we denote by $\{s_1,\ldots,s_n\}$ a quotient set of amalgamated services, we have that $\gamma^u(\{s_1,\ldots,s_n\})=\underline{\alpha}_{i_1}(\gamma^u{}_{i_1}(s_1))\wedge\ldots\wedge\underline{\alpha}_{i_n}(\gamma^u{}_{i_n}(s_n))$ where α_{i_j}' is either α_1 or α_2, depending whether s_j belongs to S_1 or S_2. Similarly, if we denote by $\{s_1',\ldots,s_m'\}$ a quotient set of amalgamated services of designs D_1' and D_2', we have that $\gamma^u{}'(\{s_1',\ldots,s_n'\})=\underline{\alpha}_{i_1}'(\gamma^u{}_{i_1}(s_1'))\wedge\ldots\wedge\underline{\alpha}_{i_n}'(\gamma^u{}_{i_n}(s_n'))$ where α_{i_j}' is either α_1' or α_2', depending whether s_j' belongs to S_1' or S_2'. We prove that, for every s_j' in the quotient set corresponding to $\mu(s)$ (i.e., s_j' in D_1' or D_2' s.t. $\underline{\alpha}_{j}(s_j')=\mu_{sv}(s)$), $\underline{\mu}(\underline{\alpha}_{i_1}(\gamma^u{}_{i_1}(s_1)))\wedge\ldots\wedge\underline{\mu}(\underline{\alpha}_{i_n}(\gamma^u{}_{i_n}(s_n)))\supset\underline{\alpha}_{j}'(\gamma^u{}_{j}'(s_j'))$ holds.
  It is not difficult to conclude that, for every s_j' in the quotient set corresponding to $\mu(s)$, there exists a s_j in the quotient set corresponding to s s.t. $\mu_{i_j}(s_j)=s_j'$. Because μ_{i_j} is a refinement morphism, we have that $\underline{\mu}_{j}(\gamma^u{}_{i_j}(s_j))\supset\gamma^u{}_{i_j}'(\mu_{i_j}(s_j))$ holds, i.e., $\underline{\mu}_{j}(\gamma^u{}_{i_j}(s_j))\supset\gamma^u{}_{i_j}'(s_j')$ holds. Then, $\underline{\alpha}_{i_j}'(\underline{\mu}_{j}(\gamma^u{}_{i_j}(s_j)))\supset\underline{\alpha}_{i_j}'(\gamma^u{}_{i_j}'(s_j'))$ also holds. The result follows trivially from the equalities $\alpha_i;\mu=\mu_i;\alpha_i'$.