A Formal Approach to 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@fiadeiro.org
[2] Department of Informatics, Faculty of Sciences, University of Lisbon,
Campo Grande, Lisboa 1749-016, Portugal
mal@di.fc.ul.pt

Abstract. We develop a formal approach to 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 CommUnity approach to architectural modelling with "implicit invocation", or "publish/subscribe" interactions. This is a first step towards a formal integration of architectural styles.

1 Introduction

Event-based interactions are now established as a major paradigm for large-scale distributed applications (e.g. [1,3,5,10,12]). 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 [14] focuses on tool integration and software evolution, the paradigm is much more general: components can be all sorts of runtime 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.

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

L. Baresi and R. Heckel (Eds.): FASE 2006, LNCS 3922, pp. 18 – 32, 2006.

acknowledge the merit of several efforts towards providing methodological principles and formal semantics (e.g. [14]), including recent incursions on using model-checking techniques for reasoning about such systems [2,9]. 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 makes another contribution in this direction by investigating how event-based interactions can be formalised in a categorical setting similar to the one that we developed in [7] for i/o-communication and action synchronisation (rendez-vous) around the language CommUnity. Our formalisation addresses the architectural properties, i.e. the discipline of decomposition and interconnection, not the notification and subscription mechanisms that support them. More precisely, it serves two main purposes. On the one hand, to 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 [14] 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. On the other hand, we wish to further validate and refine the categorical approach that we have been developing to support architectural modelling by investigating how the "implicit invocation" architectural style can be captured as a coordinated category [6]. This is a first step towards a formal approach to the integration of architectural styles.

In section 2, we introduce our primitives for modelling publish/subscribe interactions using a minimal language in the style of CommUnity [7]. In section 3, we define the category over which we formalise our approach. 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. In section 4, we show how event bindings can be externalised and made explicit in configuration diagrams. In section 5, we give a necessarily brief account of how we can use the categorical formalisation to bring several architectural styles together.

2 Event-Based Designs

We model components that keep a local state and subscribe to a number of events. Upon notification that one such event has taken place, a component invokes one or more services. If, when invoked, 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. We start with 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∉elms
   publishes inserted
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elms
   publishes deleted
   effects  elems'=elems\{lm} ∧
      deleted! ∧ deleted.which=lm
```

Even if the notation is self-explanatory, we need to discuss some of its features:

- When declaring the events that a component *subscribes*, we identify under *invokes* the services that may be invoked when a notification is received. Under *handledBy*, we specify the different ways in which a notification is handled, using *s?* to denote the invocation of service *s*.
- Parameter passing is made explicit through expressions within specifications. 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*.
- Under *store* we identify the state variables 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 and, through *publishes*, we identify the events that a service may publish.
- When specifying the *effects* of a service, *v'* denotes the value that state variable *v* takes after it is executed, and *e!* denotes the publication of event *e*.
- Through *guardedBy* we identify the enabling condition of a service, i.e. the set of states in which its invocation is accepted and the service is executed.
- Designs can be underspecifed, 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.

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∉elms
   publishes inserted
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elms
   publishes deleted
   effects  elems'=elems\{lm} ∧
     deleted! ∧ deleted.which=lm
provide inc
   assignsTo value
   effects  value'=value+1
provide dec
   assignsTo value
   effects  value'=value-1
```

We can keep extending the design 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*.

```
design Set&Counter&Adder is
store elems: set(nat),
     value:nat, sum:nat
publish&subscribe inserted
   par which:nat
   invokes inc, add
   handledBy inc?
   handledBy add? ∧
     which=add.lm
publish&subscribe deleted
   par which:nat
   invokes dec,sub
   handledBy dec?
   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∉elms
   publishes inserted
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   guardedBy lm∈elms
   publishes deleted
   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 we can declare more than one handler for a given event subscription. For instance, the event *inserted* has two handlers: one invokes *add*

and the other invokes *inc*. Both invocations are independent in the sense that they can take place at different times. This is different from declaring just one handler of the form $inc? \wedge add? \wedge which=add.lm$; such a handler would require synchronous invocation of both services. The latter is useful when one wants to make sure that separate state components are updated simultaneously, say to ensure that the values of *sum* and *count* apply to the same set of elements.

As a design of a system, *Set&Counter&Add* seems to be highly unstructured: we seem to have lost the original *Set*; and where is the *Counter*? and the *Adder*? In the next section, we show how *Set&Counter&Add* can be designed by interconnecting separate and independent components, including mediators in the sense of [14].

3 Structuring Event-Based Designs

In order to discuss the structuring of event-based designs, we adopt the categorical approach that we have been developing for architectural modelling [6,7]. In Category Theory, the structure of objects such as the designs introduced in the previous section is formalised in terms of *morphisms*. A morphism is simply a mechanism for recognising a component within a larger system.

In the examples discussed in the previous section, we used a number of data types and data type constructors. In order to remain independent of any specific language for the definition of the data component of designs, we assume a data signature $\Sigma=<D,\Omega>$, where D is a set (of sorts) and Ω is a $D^* \times D$-indexed family of sets (of operations), to be given together with a collection Φ of first-order sentences specifying the functionality of the operations. We refer to this data type specification by Θ.

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

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

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

The mapping P defines, for every event and service, the name and the type of its parameters. Every variable and parameter v is typed with a sort $sort(v) \in D$. The sets $V_{d \in D}$, E, S, $P_{s \in S}$ and $P_{e \in E}$ are assumed to be mutually disjoint. This is why the "official" name of, for instance, parameter *which* of event *inserted* is *inserted.which*.

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

- 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}

Events are published by services. We declare the events that each service may publish through the mapping B. For instance,

- $B_S(insert)=B_{SC}(insert)=B_{SCA}(insert)$={inserted}
- $B_S(delete)=B_{SC}(delete)=B_{SCA}(delete)$={deleted}

For every service s, another set $A(s)$ is defined 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. 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 $G(e)$ the set of services that may be invoked. For instance,

- $G_S(doInsert)=G_{SC}(insert)=G_{SCA}(insert)$={insert}
- $G_{SC}(inserted)$={inc}
- $G_{SCA}(inserted)$={inc,add}

Notice that the functions A, B, and G just declare the state variables, events and services that can be changed, published, and invoked, respectively. Nothing in a signature states how state variables are changed, or how and in which circumstances events are published or services invoked. In brief, 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. This additional information that pertains to the individual behaviour of components is defined in the *bodies* of designs:

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

- *η assigns to every handler $h \in H(e)$ of a subscribed event $e \in Sub(E)$, a proposition in the language of V (state variables), the parameters of e, the services declared in G(e) and their parameters.*

- *ρ assigns to every service $s \in S$, a proposition in the language of V, the parameters of s, the primed variables in the domain of s, as well as the events – $B(e)$ – that may be published by the service and their parameters.*
- *γ assigns to every service $s \in S$, a proposition in the language of V (state variables) and the parameters of s.*

By "the language of X" we mean the first-order language generated by using X as atomic terms. 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 propositions $\eta(h)$ for every handler $h \in H(e)$. For instance, in *Set&Counter&Adder*, we have $H_{SCA}(inserted)$ given by two handlers whose requirements are *inc?* and $(add? \wedge inserted.which=add.lm)$. Every handling requirement (handling for short) is enforced when the event is published. Each handling consists of service invocations and other properties that need to be observed on invocation (e.g. for parameter passing) or as a precondition for invocation (e.g. in the case of filters for discarding notifications). A typical handling is of the form $\psi \supset (s? \wedge \phi)$ establishing that *s* is invoked with property ϕ if condition ψ holds on notification.
- for every service *s*, *an enabling condition* – $\gamma(s)$ – defining the states in which the invocation of *s* can be accepted. This is the condition that we specify under *guardedBy*.
- for every service *s*, a proposition – $\rho(s)$ – defining the *state changes* that can be observed due to the execution of *s*. As shown in the examples, this proposition may include the publication of events and parameter passing. This is the condition that we specify under *effects*.

The language over which propositions used in η, γ and ρ are written extends that used for the data type specification with state variables (and their primed versions in the case of ρ) as nullary operators. Qualified parameters of events and services are also taken as nullary operators. In the case of $\rho(s)$ this extension also comprises the events of $B(s)$ as nullary operators that represent the publication of the corresponding event. This is why $\rho_{SCA}(insert)$ includes the expression *inserted!* indicating the publication of the event *inserted*. In the case of $\eta(e)$ the extension includes services $a \in G(e)$ as nullary operators that represent their invocation, what we denote with *a?*.

As already mentioned, the structure of designs is captured through *morphisms*. These are maps between designs that identify ways in which the source is a component of the target. We define first how morphisms act on signatures:

Definition/Proposition: *A morphism $\sigma: Q_1 \to Q_2$ for $Q_1=<V_1,E_1,S_1,P_1,T_1,A_1,B_1,G_1,H_1>$ 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*

- *σ_{st}: $V_1 \to V_2$ is a function on state variables that preserves their sorts, i.e. $sort_2(\sigma_{st}(v))=sort_1(v)$ for every $v \in V_1$*
- *σ_{ev}: $E_1 \to E_2$ is a function on events that preserves kinds, i.e. $\sigma_{ev}(e) \in Pub(E_2)$ for every $e \in Pub(E_1)$ and $\sigma_{ev}(e) \in Sub(E_2)$ for every $e \in Sub(E_1)$, as well as invoked services, i.e. $\sigma_{sv}(G_1(e)) \subseteq G_2(\sigma_{ev}(e))$ for every $e \in Sub(E_1)$.*

- σ_{sv}: $S_1 \to S_2$ *is a function that preserves domains, i.e* $A_2(\sigma_{st}(v))=\sigma_{sv}(A_1(v))$ *for every* $v \in V_1$, *as well as published events, i.e.* $\sigma_{ev}(B_1(s)) \subseteq B_2(\sigma_{sv}(s))$
- $\sigma_{par\text{-}ev}$ *maps every event e to a function* $\sigma_{par\text{-}ev,e}$: $P_1(e) \to P_2(\sigma_{ev}(e))$ *that preserves the sorts of parameters, i.e.* $sort_2(\sigma_{par\text{-}ev,e}(p))=sort_1(p)$ *for* $p \in P_1(e)$
- $\sigma_{par\text{-}sv}$ *operates like* $\sigma_{par\text{-}ev}$ *but on service parameters*
- $\sigma_{hr\text{-}ev}$ *maps every subscribed event e to a function* $\sigma_{hr\text{-}ev,e}$: $H_1(e) \to H_2(\sigma_{ev}(e))$.

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

A morphism σ from Q_1 to Q_2 is intended to support 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 *Set&Counter&Adder*.

Notice that 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 $T_{SCA}(inserted)=pubsub$.

The constraints on domains imply that new services of the system cannot assign to variables of the component. This is what makes state variables "private" to components. As a result, we cannot identify components of a system by grouping state variables, services and events in an arbitrary way. For instance, we can identify a counter as a component of *Set&Counter&Adder* as follows. Consider the following design:

```
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
```

It we map *doInc* to *inserted* and *doDec* to *deleted*, we do define a morphism between the signatures of *Counter* and *Set&Counter&Adder*. 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 *Set&Counter&Adder* do not assign to it.

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, the components that are responsible for publishing events, as well as those that will subscribe published events, are not 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, the morphism that identifies *Counter* as a component of *Set&Counter&Adder* is not just an injection. Do notice that the binding also implies that *inserted* and *deleted* are subscribed within *Set&Counter&Adder*. As a result, our components are independent in the sense of [14]: 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. We recall that we denote by Φ the specification of the data sorts and operations.

Definition/Proposition: *A superposition morphism* $\sigma:\langle Q_1,\Delta_1\rangle \to \langle Q_2,\Delta_2\rangle$ *consists of a signature morphism* $\sigma:Q_1 \to Q_2$ *such that:*

1. *Handling requirements are preserved: for every event* $e \in E_1$ *and handling* $h \in H_1(e)$, $\Phi \vdash \eta_2(\sigma_{hr\text{-}ev,e}(h)) \supset \underline{\sigma}(\eta_1(h))$
2. *Effects are preserved:* $\Phi \vdash (\rho_2(\sigma_{sv}(s)) \supset \underline{\sigma}(\rho_1(s))$ *for every* $s \in S_1$
3. *Guards are preserved:* $\Phi \vdash (\gamma_2(\sigma_{sv}(s)) \supset \underline{\sigma}(\gamma_1(s))$ *for every* $s \in S_1$

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

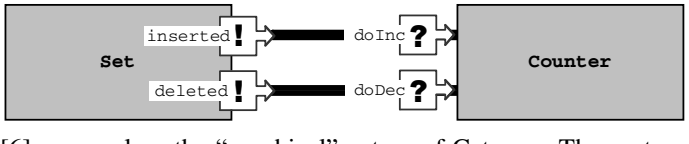
Notice that the first condition allows for more handling requirements to be added and, for each handling, subscription conditions to be strengthened. In other words, as a result of being embedded in a bigger system, a component that publishes a given event may acquire more handling requirements but also 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 *Set&Counter&Adder*. 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 next section.

4 Externalising the Bindings

As explained in [7], one of the advantages of the categorical formalisation is that it supports a design approach based on superposing separate components (connectors) over independent units. These separate components are called mediators in [14]: for instance, *Set* as used for connecting a *Counter* and independent components that publish insertions and deletions. Morphisms, as defined in the previous section, enable the definition of such a design approach by supporting the externalisation of bindings.

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 [6], 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:

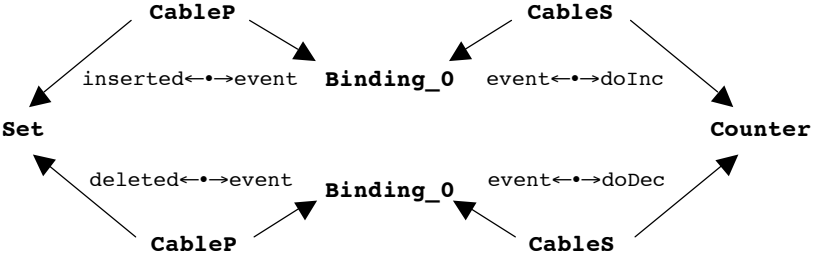
```
design Binding_0 is
publish&subscribe event
```

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.

```
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 ***DSGN*** of designs:



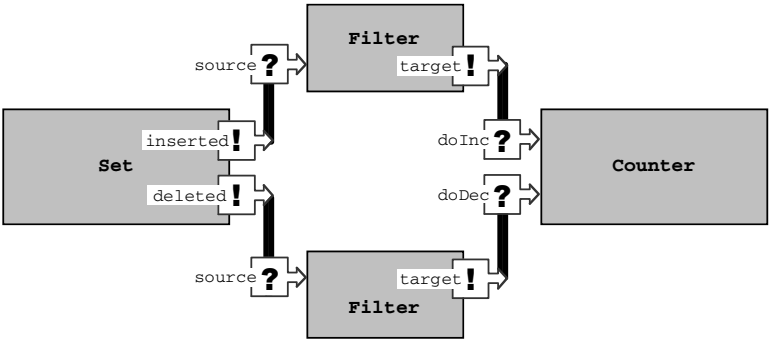
In Category Theory, diagrams are mathematical objects and, as such, can be manipulated in a formal way. One of the constructs that are available on certain diagrams internalises the connections in a single (composite) component. In the case above, this consists in computing the colimit of the diagram [6], which returns the design *Set&Counter* discussed in section 2. In fact, the colimit returns the morphisms that identify both *Set* and *Counter* as components of *Set&Counter*.

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* we would use a more elaborate connector *Filter* defined as follows:

```
design Filter is
publish&subscribe target
provide service
   publishes target
   effects target!
```

```
publish&subscribe source
   par n:nat
   invokes service
   handledBy iseven(n) ⊃ service?
```

This connector is made explicit in the configuration as a mediator:

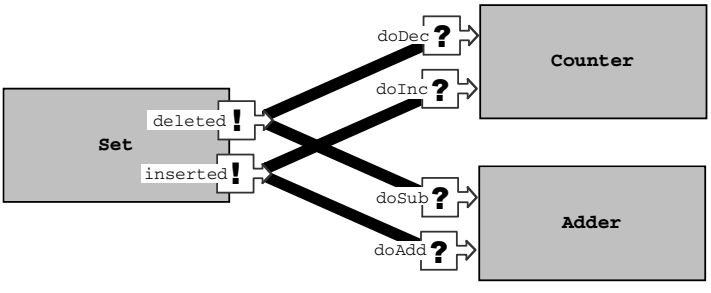


The same design approach can be applied to the addition of an Adder:

```
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 *Set&Counter&Adder* discussed in section 2 and the morphisms that identify *Set, Adder* and *Counter* as components.

5 Combining Architectural Styles

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 [6]. 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 *Adder* subscribes to *inserted* which is published after the element has been inserted in the set. What we need is to strengthen the enabling condition of *insert* in *Set* with *sum+lm<LIM* and ensure that *sum* is updated by *insert* and *delete*. However, to do so within ***DSGN*** we would have to redesign the whole system. Ideally, we would like to remain within the incremental 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 [6]. 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.

Consider the synchronisation of *Set* and *Adder* first. 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. Our first step is to extend the notion of design with synchronisation constraints and communication channels.

Definition: *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). 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 can be used in the languages of ρ and γ, and:*

- *β is a proposition establishing what observations of the local state (variables) are made available through the output channels.*
- *χ is a proposition in the language of services and their parameters establishing dependencies that need to be observed on execution.*

As an example, consider the following revision of *Set&Counter&Adder:*

```
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
   publishes inserted
   guardedBy lm∉elms ∧ lm+sum<LIM
   effects  elems'={lm}∪elems ∧
     inserted! ∧ inserted.which=lm
provide delete
   par lm:nat
   assignsTo elems
   publishes deleted
   guardedBy lm∈elms
   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
```

Through *synchronise* we provide a proposition that defines the synchronisation sets of service activation that can be observed during execution. For instance, through $a \equiv b$, we can specify that two given services *a* and *b* are always activated 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 envi-

ronment through *mysum*. Notice that we have also strengthened the guard of *insert* with the condition *lm+sum<LIM*.

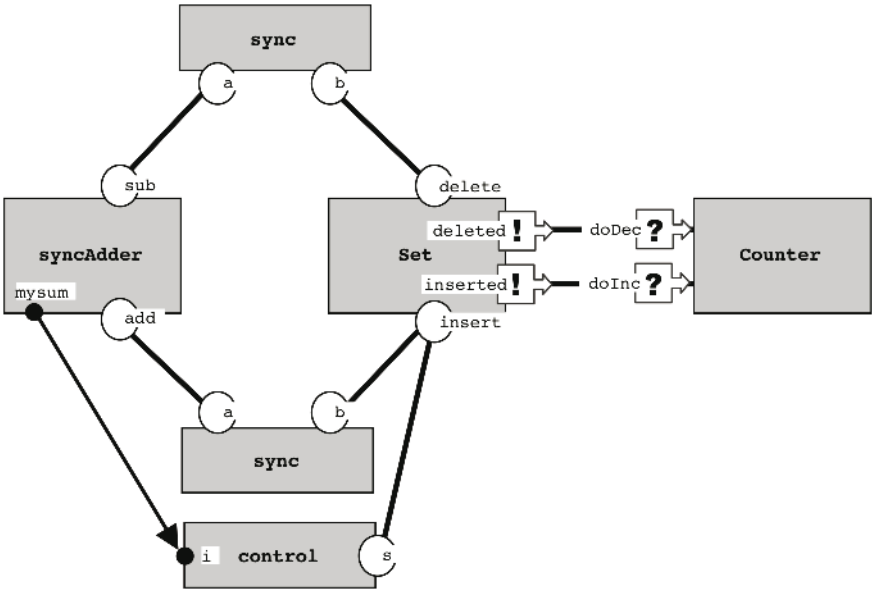
It remains to show how we can externalise the extension. The following design captures the synchronisation:

```
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 the following configuration:



Notice that *syncAdder* is given by the following design:

```
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
```

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

Definition: *A morphism* σ *between extended signatures* $<V_1,E_1,S_1,P_1,T_1,H_1,I_1,O_1>$ *and* $<V_2,E_2,S_2,P_2,T_2,H_2,I_2,O_2>$ *is a morphism between signatures* $<V_1,E_1,S_1,P_1,T_1,H_1>$ *and* $<V_2,E_2,S_2,P_2,T_2,H_2>$ *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 [6], input channels may become output channels of the system but not the other way around.

Definition: *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, i.e. morphisms between designs that do not involve communication channels and synchronisations are as before. Further details on this extension, including the way it relates to CommUnity, can be found in a companion paper.

6 Conclusions and Further Work

In this paper, we presented a formalisation of the architectural style known as "publish/subscribe" or "implicit invocation". Full details on the mathematics involved as well as the semantics of publication and notification can be found in a companion paper. This formalisation allowed us to further validate the approach to software architecture introduced in [7].

Other formal models [e.g., 4,9] 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 primarily 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 particular, 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 [14]. 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 [6]. We should stress that these structural properties result from the nature of the morphisms that we defined in section 3, which may leave some readers who are not aware of the complexity of the mathematics involved somewhat disappointed and wishing to have seen more results… It is true that, in this paper, we have "only" defined a category and a (forgetful) functor, but both satisfy very strong properties that can be used for further exploring implicit invocation as an architectural style.

Furthermore, the proposed categorical semantics allows us to investigate how this style can be used in conjunction with other architectural techniques. In section 5, we addressed the way implicit invocation can be used together with synchronous forms of interconnection as previously formalised through the language CommUnity [6]. CommUnity itself has been extended in other ways, for instance with primitives that capture distribution and mobility [8] as well as context awareness [11]. Further work

is going on towards exploiting this categorical framework to support the integration of several architectural styles.

Acknowledgements

This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*. We would like to thank the reviewers for having provided so much feedback.

References

1. 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
2. J. Bradbury, J. Dingel (2003) Evaluating and improving the automatic analysis of implicit invocation systems. In: *ESEC/FSE'03*. ACM Press, pp 78–87
3. 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
4. J. Dingel, D. Garlan, S. Jha, D. Notkin (1998) Towards a formal treatment of implicit invocation. *Formal Aspects of Computing* 10:193–213
5. P. Eugster, P. Felber, R. Guerraoui, A-M. Kermarrec (2003) The many faces of publish/subscribe. *ACM Computing Surveys* 35(2):114–131
6. J. L. Fiadeiro (2004) *Categories for Software Engineering*. Springer, Berlin Heidelberg New York
7. 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
8. J. L. Fiadeiro, A. Lopes (2004) CommUnity on the move: architectures for distribution and mobility. In: M. Bonsangue et al (eds) *Formal Methods for Objects and Components. LNCS, vol 3188*. Springer, Berlin Heidelberg New York, pp 177–196
9. 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
10. 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
11. A. Lopes, J. L. Fiadeiro (2005) Context-awareness in software architectures. In: R. Morrison, F. Oquendo (eds) *Software Architecture*. *LNCS, vol 3527*, Springer, Berlin Heidelberg New York, pp 146–161
12. 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
13. D. Notkin, D. Garlan, W. Griswold, K. Sullivan (1993) Adding implicit invocation to languages: three approaches. In: S. Nishio, A. Yonezawa (eds) *Object Technologies for Advanced Software*. *LNCS, vol. 742*, Springer, Berlin Heidelberg New York, pp 489–510
14. K. Sullivan, D. Notkin (1992) Reconciling environment integration and software evolution. *ACM TOSEM* 1(3):229–268

.