Math. Struct. in Comp. Science (2007), *vol.* 17, *pp.* 1029–1073.
doi:10.1017/S0960129507006299 Printed in the United Kingdom

An algebraic semantics of event-based architectures

JOSÉ LUIZ FIADEIRO[†] and ANTÓNIA LOPES[‡]

[†]*Department of Computer Science, University of Leicester, University Road, Leicester LE1 7RH, U.K.*
Email: jose@mcs.le.ac.uk

[‡]*Department of Informatics, Faculty of Sciences, University of Lisbon, Campo Grande, 1749-016 Lisboa, Portugal*
Email: mal@di.fc.ul.pt

Received 2 May 2006; revised 11 December 2006; first published online 21 September 2007

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', also known as '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 (see, for example, Bacon *et al.* (2000), Carzaniga *et al.* (2001), Eugster *et al.* (2003), Garlan and Notkin (1991) and Meier and Cahill (2002)). 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 (Sullivan and Notkin 1992) 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 any events they subscribe to, or the subscribers of any events that they publish. In particular, event notification and propagation are performed asynchronously, that is, 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, that is, 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.

Precisely these claims can be found in Sullivan and Notkin (1992) 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. Despite the merits of several efforts towards providing methodological principles and formal semantics such as Dingel *et al.* (1998), including recent incursions in the use of model-checking techniques for reasoning about such systems (Bradbury and Dingel 2003; Garlan *et al.* 2003), we are still far from an accepted 'canonical' semantic model over which all these efforts can be brought together to provide effective support and to formulate methodological principles that can steer development independently of specific choices of middleware.

This paper is an extended version of Fiadeiro and Lopes (2006) 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 Fiadeiro and Lopes (1997) around the language CommUnity. As with 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 (Fiadeiro 2004) 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 such as superposition (Katz 1993).

We take this formalisation effort a step further in this paper 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, that is, 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 Sullivan and Notkin (1992) 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 to investigate extensions of event-based interactions with i/o-communication and action synchronisation (rendezvous) as available in CommUnity. We see this as a first step towards a formal approach to the engineering of architectural styles.

Organisation of the paper

In Section 2, we introduce our primitives for modelling publish/subscribe interactions using a minimal language in the style of CommUnity, which 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 in which 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 with respect to superposition as captured through the morphisms studied in Section 4. 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. Appendix A provides a glossary collecting together the different terms used in the definition of the syntax and semantics of e-CommUnity. Finally, the proofs of the main results of Section 5 are given in Appendix B.

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 are addressing service-oriented architectures with similar mathematical techniques (for preliminary work in this direction, see Fiadeiro *et al.* (2006; 2007)), this paper is concerned only with event-based interactions in general. Therefore, we will use the term service in a rather loose way, that is, without committing to a full service-oriented approach in the sense of, say, web-services (Alonso *et al.* 2004) or wide-area computing (Misra and Cook 2006).

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 begin the discussion of 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 presented in Figure 1. This is 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.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 1. The design of *Set*

We formalise the languages that are used for specifying component behaviour in Section 3, together with a denotational semantics for the underlying computational model. Meanwhile, we will just provide an informal overview of all the 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 use *s?* to denote its invocation when specifying how the notification that the event has taken place is handled, which we do under *handledBy*.
— Parameter passing is made explicit through the 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, that is, what 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.
— We use *assignsTo* to 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. Note that because designs can be under specified, the write frame of a service cannot be inferred

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 2. The design of *Set&Counter*

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.

— We use *guardedBy* to specify the enabling condition of a service, that is, 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 $lm \notin elems$ 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 for *delete*, which is specified to be enabled if and only if the element whose deletion is being requested belongs to the set.

Consider now the design presented in Figure 2. This is the design of a system in which a counter subscribes *inserted* and *deleted* to count the number of elements in the set. 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

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 3. The design of *Set&Counter&Adder*

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 design presented in Figure 3.

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.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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 also where are 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 Sullivan and Notkin (1992). However, before we do that, we will 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 we have adopted in previous papers for defining the syntax and semantics of specification and design languages (Fiadeiro 2004).

3.1. *Signatures*

We begin by formalising the language that we use for defining designs, starting with the data types and data type constructors. As can be seen in the examples discussed in the previous section, data structures are used for defining the computational aspects of component behaviour as well as for 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 $\Sigma = \langle D, F \rangle$, 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 will refrain from expanding further on the algebraic aspects of data type specification, the theory of which can be found in textbooks such as Ehrig and Mahr (1985).

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

Definition 3.1. A *signature* is a tuple $Q = \langle V, E, S, P, T, A, G, H \rangle$ 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 use $Pub(E)$ to denote the set $\{e \in E : T(e) \neq sub\}$ and $Sub(E)$ for 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 $G(e, h) \subseteq S$ consisting of the services that can be invoked by that event through that handler. Given that the sets $H(e)$ are mutually disjoint, we simplify the notation and use $G(h)$ instead of $G(e, h)$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

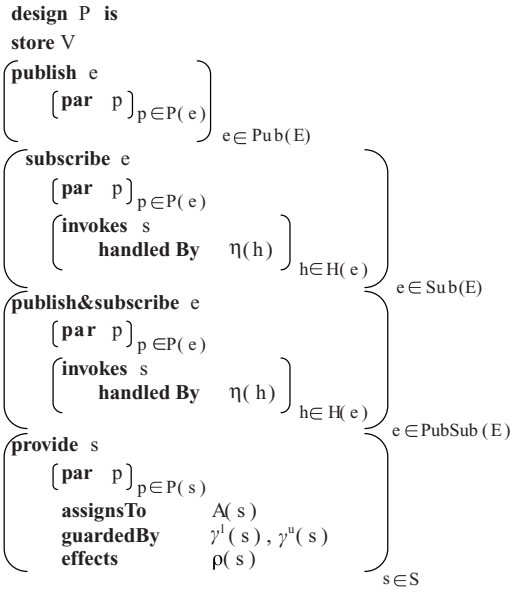

Fig. 4. An e-CommUnity design.

Every state variable v is typed with a sort, which is an element of D and which we denote by $sort(v)$; the set of variables whose type is d is V_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, that is, $P(s)_d$ (respectively, $P(e)_d$) is the set of parameters of service s (respectively, event e) that are of sort d; as 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 have used the 'dot-notation' according to which the 'official' name of, for instance, the 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:

— $E_S = \{inserted, deleted, doInsert, doDelete\}$
— $T_S(inserted) = T_S(deleted) = pub$

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

— $T_S(doInsert) = T_S(doDeleted) = sub$
— $Sub_S(E) = \{doInsert, doDelete\}$
— $Pub_S(E) = \{inserted, deleted\}$.

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, that is, $A(v) = \{s \in S | v \in A(s)\}$. 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 use $H(e)$ to denote the mechanisms (handlers) through which notifications that e has occurred are handled. Each handler h defines a specific way that the component can 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.

Note 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 what 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 $\mathscr{D}$ that validates the specification Θ (see Ehrig and Mahr (1985) for details):

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

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 signature $Q = \langle V, E, S, P, T, A, G, H \rangle$ 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 service $s \in S$, a new sort d^s and, for every parameter $p \in P(s)_d$ of sort d, an operation $d^p : d^s \to d$;

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

- 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 $s \in S$ and published event $e \in Pub(E)$, an operation $pub^{s,e} : d^s \to d^e$.

— An algebra $\mathscr{U}$ for Σ_Q that extends the Σ-algebra $\mathscr{D}$ in the sense that $\mathscr{D}$ is the reduct of $\mathscr{U}$ 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. We require that these sorts are assigned mutually disjoint domains so that reducts, as defined in Section 5.2, can forget invocations and pending events.

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. In aaddition to these operations that return data values, we define two other kinds of operations: *inv*, which return the service instances invoked by every event occurrence; and *pub*, which return the event instances published by every service execution.

Notice that we are defining a 'declarative' or 'denotational' semantics, not an operational one: 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, that is, it does not interfere with the sets of data and the operations on data. In other words, $\mathscr{U}$ and $\mathscr{D}$ 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. For the rest of this section, we consider a fixed Q-space.

Definition 3.3. An *execution state* consists of a pair $\langle VAL, PDN \rangle$ where:

— VAL (valuation) is a mapping that, to every data sort $d \in D$ and state variable $v \in V_d$, assigns a value $VAL(v) \in d_{\mathscr{D}}$.

— PDN is a set whose elements (pending invocations) are triples $\langle t, h, u \rangle$ where:

- t is an event instance, that is, an element of for some event $e \in Sub(E)$.
- h is a handler for e, that is, $h \in H(e)$.
- u is a service instance invoked by t through h, that is, $u = inv^{h,s}_{\mathscr{U}}(t)$ 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. In addition 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 PDN. As discussed below, a service invocation becomes pending, and is added to PDN, when an event is published that includes the service in its list of

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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 to satisfy some conditions.

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

— For any invocation $\langle t, h, u\rangle \in INV$, if $\langle t, h, u'\rangle \in PDN$, then $\langle t, h, u'\rangle \in INV$.
— For every pair of services s and s' and different service invocations $\langle t, h, u\rangle$ and $\langle t', h', u'\rangle$ in INV with $u \in d^s_{\mathscr{U}}$ and $u' \in d^{s'}_{\mathscr{U}}$, we have $A(s) \cap A(s') = \varnothing$.

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.

In addition 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* is a tuple $\langle SRC, TRG, INV, EXC, PUB, NXT\rangle$ where:

— SRC (the source) and TRG (the target) are two execution states;
— INV is a subset of PDN_{SRC} (the set of actual service invocations);
— EXC is a set of service instances (these correspond to the actual service invocations that are enabled in SRC);
— PUB is a set whose elements are the event instances published at that step;
— NXT is a subset of PDN_{TRG} (the set of next service invocations);

satisfying the following properties:

— for every $u \in EXC$ there is $\langle t, h, u\rangle \in INV$;
— for every $\langle t, h, u\rangle \in NXT$, $t \in PUB$, that is, only services invoked by published events can become pending;
— $PDN_{TRG} = PDN_{SRC} \backslash INV \cup NXT$, that is, 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 $u \in d^s_{\mathscr{U}}$ such that $v \in A(s)$, that is, 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

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

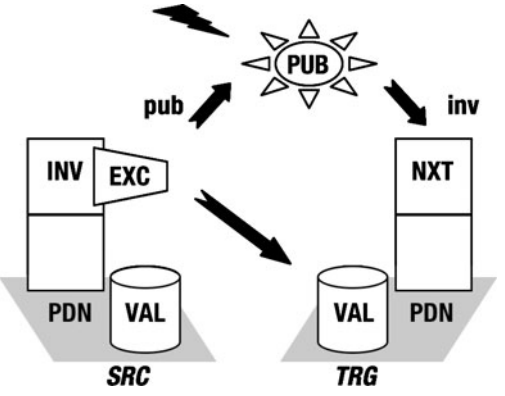

Fig. 5. Representation of the elements involved in the execution steps.

during an execution step, which is captured by the set PUB; and these published events add pending invocations to the target state, which 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, which is captured through INV. Services belonging to the discarded invocations that are enabled in the source state are executed, which 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 eventually be selected and executed if enabled. Notice that this is not a property of any individual execution step but of the global execution model; therefore, this is not captured in the above definition.

The picture presented in Figure 5 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; that is, 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 that includes the variable in its domain has been executed.

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.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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

— For every arrow $r = \langle n_1, n_2 \rangle$, $\mathscr{L}(r)$ is of the form $\langle \mathscr{L}(n_1), \mathscr{L}(n_2), \_, \_, \_, \_ \rangle$.
— $VAL_{TRG}(v) = VAL_{TRG'}(v)$ for every state variable $v \in V$ and every pair of arrows $r = \langle n, m \rangle$ and $r' = \langle n, m' \rangle$ such that there are $s \in A(v)$ and $u \in d^s_{\mathscr{U}} \cap EXC \cap EXC'$, where $\mathscr{L}(r)$ is of the form $\langle \_, TRG, \_, EXC, \_, \_ \rangle$ and $\mathscr{L}(r')$ is of the form $\langle \_, TRG', \_, EXC', \_, \_ \rangle$.
— For any two arrows $r = \langle n, m \rangle$ and $r' = \langle n, m' \rangle$ where $\mathscr{L}(r)$ is of the form $\langle \_, \_, INV, EXC, \_, \_ \rangle$ and $\mathscr{L}(r')$ is of the form $\langle \_, \_, INV', EXC', \_, \_ \rangle$, and service instance u such that $\langle \_, \_, u \rangle \in INV$ and $\langle \_, \_, u \rangle \in INV'$, $u \in EXC$ if and only if $u \in EXC'$.

Note 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 allow 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, branching in a model, that is, 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, note that signatures include typing information that is sometimes associated with the 'semantics' such as the encapsulation of state change and service invocation. In brief, as we will explain later in the paper, 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 begin with the language we use to specify the guards of services.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

Definition 3.7. Given a signature $Q = \langle V, E, S, P, T, A, G, H \rangle$ 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 $\Sigma = \langle D, F \rangle$ 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 $u \in D$, we evaluate the sentences of $SL_{Q,s}$ in the extension of the Σ-algebra $\mathscr{D}$ with

— $p_{\mathscr{D}} = d^p_{\mathscr{U}}(u)$;
— $v_{\mathscr{D}} = VAL(v)$.

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 we require for determining if a service is enabled. An example is the sentence $delete.lm \in elems$ in the state language of *delete*; this sentence involves the parameter *delete.lm* as well as the state variable *elems*.

Consider now the language we use to specify the effects of services.

Definition 3.8. Given a signature $Q = \langle V, E, S, P, T, A, G, H \rangle$ and a service $s \in S$, we define *the transition language* $TL_{Q,s}$ associated with s as the first-order language (with equality) generated by the data signature $\Sigma = \langle D, F \rangle$ 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 $u \in d^s_{\mathscr{U}}$, we evaluate the sentences of $TL_{Q,s}$ in the extension of the Σ-algebra $\mathscr{D}$ with:

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

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 no longer specify properties of single execution states but of execution steps; this is why we call it a 'transition' language.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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 *elems*; this is because primed variables are evaluated in the target state *TRG* of the execution step. As already mentioned, *inserted!* is used to indicate 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 if and only if, when ψ is true, ϕ is also true and an event publication is added to *PUB* for the instance $pub^{s,e}_{\mathscr{U}}(u)$ 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 $pub^{s,e}_{\mathscr{U}}(t)$, which are evaluated in the algebra $\mathscr{U}$.

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 common programming language notation $v := F(s, v)$ for $v' = F(s, v)$.

Finally, we define the language we use to specify the event handlers.

Definition 3.9. Given a signature $Q = \langle V, E, S, P, T, A, G, H \rangle$ 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 $\Sigma = \langle D, F \rangle$ 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 and an instance $t \in d^e_{\mathscr{U}}$, we evaluate the sentences of $HL_{Q,h}$ in the extension of the Σ-algebra $\mathscr{D}$ with:

— $p_{\mathscr{D}} = d^p_{\mathscr{U}}(t)$ for $p \in P(e)$.
— $p_{\mathscr{D}} = d^p_{\mathscr{U}}(inv^{h,s}_{\mathscr{U}}(t))$ for $p \in P(s)$ and $s \in G(h)$.
— $s?$ is true if and only if $\langle t, h, inv^{h,s}_{\mathscr{U}}(t) \rangle \in NXT$.

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 for an event e is of the form $\psi \supset (s? \wedge \phi)$, which establishes the fact that s is invoked with property ϕ if condition ψ holds on notification that an instance of e has occurred. This describes 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 $(insert? \wedge doInsert.which = insert.lm)$; this sentence uses *insert*? to indicate that the service *insert* is invoked when *doInsert* is published; furthermore, the parameter *lm* of this invocation of *insert* has the same value as the parameter *which* of *doInsert*.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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 if and only if when ψ is true, ϕ is also true and a service invocation is added to NXT for the instance $inv^{h,s}_{\mathscr{U}}(t)$ of s invoked by t through h. Notice that, typically, both ψ and ϕ are properties of the parameters of t and $inv^{h,s}_{\mathscr{U}}(t)$, which are evaluated in the algebra $\mathscr{U}$. 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 necessarily be executed as they may not be enabled.

We can now define the notion of a design.

Definition 3.10. A *design* is a pair $\langle Q, \Delta \rangle$ where Q is a signature and Δ, the body of the design, is a triple $\langle \eta, \rho, \gamma \rangle$ 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 $\rho(h)$ 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 (for example, for parameter passing) or as a pre-condition for invocation (for example, 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* $\langle Q, \Delta \rangle$ where $\Delta = \langle \eta, \rho, \gamma \rangle$ is a model of Q such that any execution step $\langle SRC, TRG, INV, EXC, PUB, NXT \rangle$ that is the label of an arrow of the underlying graph satisfies the following conditions:

— For every $u \in EXC$ with $u \in d^s_{\mathscr{U}}$, we have $\gamma^l(s)$ holds for u at SRC.
— For every $\langle t, h, u \rangle \in INV$ and $u \in d^s_{\mathscr{U}}$, if $\gamma^l(s)$ holds for u at SRC, then $u \in EXC$.
— For every $u \in EXC$ with $u \in d^s_{\mathscr{U}}$, we have $\rho(s)$ holds for u at that step.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

— For every $t \in PUB$ where $t \in d^e_{\mathscr{U}}$ and $h \in H(e)$, we have $\eta(h)$ holds for t and h at that step.

A complete execution in a model is a sequence of steps $\mathscr{L}(\langle n_i, m_i \rangle)_{i \in \omega}$ such that $m_i = n_{i+1}$ for every $i \in \omega$. We say that an execution is *fair* if and only if, for every $i \in \omega$ and $\langle t, h, u \rangle \in PDN_i$, there is $k \geqslant i$ such that $\langle t, h, u \rangle \in INV_k$.

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, for example, as in product-lines, or as knowledge of the target run-time platform becomes more precise. This topic is discussed further in Section 5.

4. Structuring event-based systems

In a categorical approach to software architecture (Fiadeiro and Lopes 1997; Fiadeiro *et al.* 2003), 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* $\sigma : Q_1 \to Q_2$ for

$$Q_1 = \langle V_1, E_1, S_1, P_1, T_1, A_1, G_1, H_1 \rangle$$

and

$$Q_2 = \langle V_2, E_2, S_2, P_2, T_2, A_2, B_2, G_2, H_2 \rangle$$

is a tuple $\langle \sigma_{st}, \sigma_{ev}, \sigma_{sv}, \sigma_{par-ev}, \sigma_{par-sv}, \sigma_{hr-ev} \rangle$, where:

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

satisfying the following conditions:

— $sort_2(\sigma_{st}(v)) = sort_1(v)$ for every $v \in V_1$, that is, the sorts of state variables are preserved;
— σ_{ev} preserves kinds, that is:
  - $\sigma_{ev}(e) \in Pub(E_2)$ for every $e \in Pub(E_1)$;
  - $\sigma_{ev}(e) \in Sub(E_2)$ for every $e \in Sub(E_1)$;

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 6. The design of *Counter*

— $A_2(\sigma_{st}(v)) = \sigma_{sv}(A_1(v))$ for every $v \in V_1$, that is, domains are preserved;
— $\sigma_{sv}(G_1(h)) \subseteq G_2(\sigma_{hr-ev}(h))$ for every $e \in E_1$ and $h \in H_1(e)$, that is, services invoked by handlers carry through;
— $\sigma_{hr-ev}(G_1(s)) = G_2(\sigma_{sv}(s))$ for every $s \in S_1$, that is, invocation of services is preserved;
— $sort_2(\sigma_{par-ev,e}(p)) = sort_1(p)$ for every $e \in E_1$ and $p \in P_1(e)$, that is, event parameter sorts are preserved;
— $sort_2(\sigma_{par-sv,s}(p)) = sort_1(p)$ for every $s \in S_1$ and $p \in P_1(s)$, that is, 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, that is, 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 $\sigma_{hr-ev}(G_1(s)) = G_2(\sigma_{sv}(s))$: 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 $\sigma_{sv}(G_1(h)) \subseteq G_2(\sigma_{hr-ev}(h))$ allows 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 (see Figure 6).

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

If we map *doInc* to *inserted* and *doDec* to *deleted*, we 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 or 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 Sullivan and Notkin (1992): 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, that is, 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 is known as 'superposition' (Lopes and Fiadeiro 2004).

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

— Handling requirements are preserved: $(\eta_2(\sigma_{hr-ev,e}(h)) \supset \underline{\sigma}(\eta_1(h)))$ holds for every event $e \in E_1$ and handling $h \in H_1(e)$.
— Effects are preserved: $(\rho_2(\sigma_{sv}(s)) \supset \underline{\sigma}(\rho_1(s)))$ holds for every $s \in S_1$.
— Lower guards are preserved: $(\gamma_2^l(\sigma_{sv}(s)) \supset \underline{\sigma}(\gamma_1^l(s)))$ holds for every $s \in S_1$.
— Upper guards are preserved: $(\gamma_2^u(\sigma_{sv}(s)) \supset \underline{\sigma}(\gamma_1^u(s)))$ holds for every $s \in S_1$.

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

We use $\underline{\sigma}$ to 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) using induction on the structure of the terms and sentences. See Fiadeiro (2004) for examples.

Note 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

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

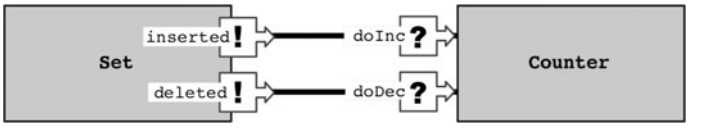

Fig. 7. A simple configuration of *set–counter*.

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 Section 4.3.

4.2. *Externalising the bindings*

As explained in Fiadeiro and Lopes (1997) and Fiadeiro *et al.* (2003), one of the advantages of a categorical formalisation of architectural design 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 Sullivan and Notkin (1992). Here we take 'separate' and 'independent' in the same sense as used in Sullivan and Notkin (1992): 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 externally superpose the bindings through which *Counter* subscribes the events published by *Set* (see Figure 7).

As in Fiadeiro (2004), 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          design CableS is
publish •                 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. This is why we represent them through the symbol •. The configuration presented in Figure 7 corresponds to the diagram (labelled graph) in the category **sDSGN** of designs presented in Figure 8.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

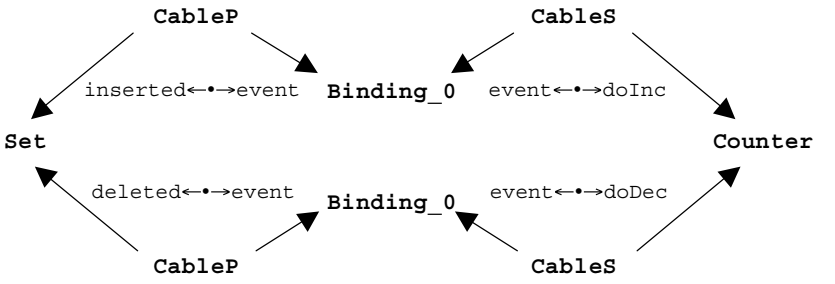

Fig. 8. A categorical diagram expressing the system configuration presented in Figure 7.

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

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

Fig. 9. The design of *Filter*.

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 above case this consists of computing the colimit of the diagram (Fiadeiro 2004), 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 will 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 *Set* and *Counter* directly, we would use the more elaborate connector (mediator) *Filter* presented in Figure 9. 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 service 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, which is presented in Figure 10, 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 we used before. The categorical diagram corresponding to this configuration is presented in Figure 11.

The connection to *Set* requires a more sophisticated binding to ensure that the parameter is transmitted. We need the interconnection presented in Figure 12 with the binding defined in Figure 13 and the cables defined in Figure 14. The other connections are established in a similar way.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

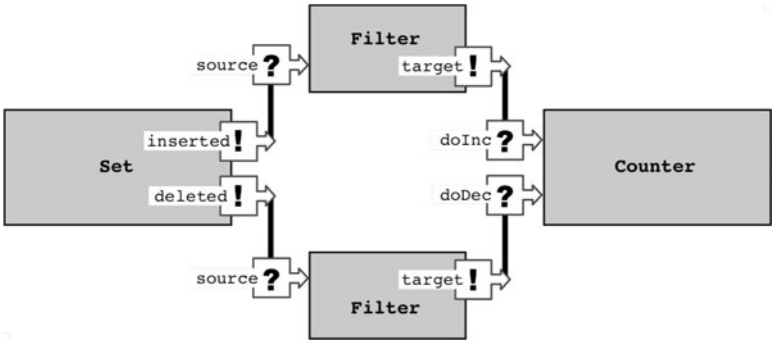

Fig. 10. A configuration involving *Filter*: only the insertions of even numbers are counted.

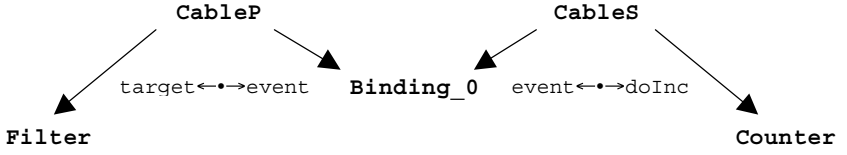

Fig. 11. Interconnection of *Filter* and *Counter*.

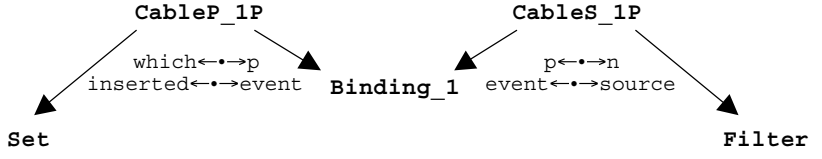

Fig. 12. Interconnection of *Set* and *Filter*.

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

Fig. 13. The binding involved in Figure 12.

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

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

Fig. 14. The cables involved in Figure 12

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 15. The design of *Adder*.

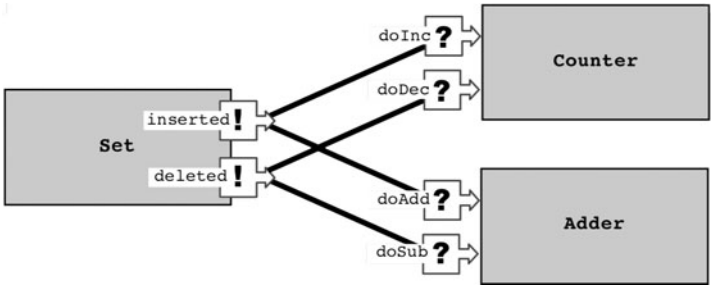

Fig. 16. A configuration of *set–counter* involving *Adder*.

The same design approach can be applied to the addition of an *Adder*, defined in Figure 15. The required configuration is shown in Figure 16. We will 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.

Note 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*, presented in Figure 17.

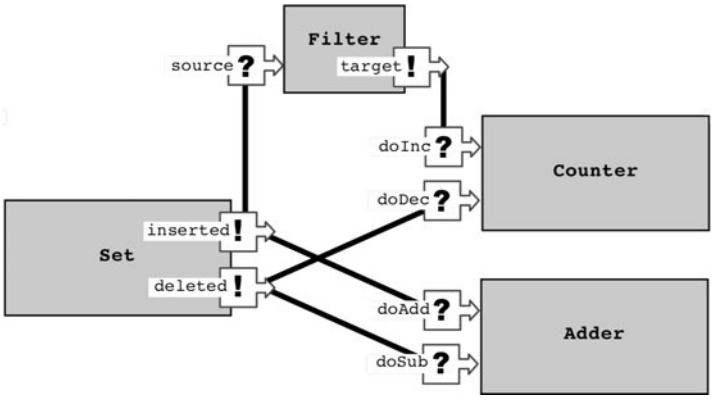

Fig. 17. A configuration of *set–counter* involving *Filter* and *Adder*.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

4.3. *The universal properties of designs*

We have already mentioned that we rely on so-called universal constructions, namely colimits, to give semantics to configuration diagrams following the same principles as 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 the configurations developed in Section 4.2.

Proposition 4.3. The functor **sign** defines **sDSGN** as a coordinated category, that is, it is faithful, lifts colimits and has discrete structures.

The proof of this property is too long to be included in this paper. However, it is useful to explain what being coordinated means and why it is meaningful in this context. The fact that e-CommUnity has discrete structures means that every signature Q has a 'canonical realisation' (a discrete lift) as a design $\mathbf{dsgn}(Q) = \langle Q, \Delta \rangle$ where the body Δ is the tuple $\langle \eta, \rho, \gamma \rangle$ with:

— For every service s, $\rho(s)$ is the proposition *true*: that is, we make no commitments about the effects of the execution of the service.
— For every service s, both guards $\gamma^l(s)$ and $\gamma^u(s)$ are the proposition *true*: that is, we make no commitments about the bounds of the enabling condition of the service.
— For every event handler h, $\eta(h)$ is the proposition *true*: that is, we make no requirements about how subscribed events are handled.

In other words, $\mathbf{dsgn}(Q)$ is completely under specified. This canonical realisation is such that every morphism $\sigma : Q \to \mathbf{sign}(\langle Q', \Delta' \rangle)$ is also a morphism of designs $\mathbf{dsgn}(Q) \to \langle Q', \Delta' \rangle$. 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, as are services and their parameters. Lifting the colimit of a diagram of signatures back to a design operates as follows. Let $\{s_1, \ldots, s_n\}$ be the quotient set of amalgamated services of the components of a system, and σ_{i_j} be the signature morphism that identifies the component to which service s_j belongs within the system. Then:

— 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, we have $\rho(\{s_1, \ldots, s_n\}) = \underline{\sigma}_{i_1}(\rho_{i_1}(S_1)) \wedge \ldots \wedge \underline{\sigma}_{i_n}(\rho_{i_n}(s_n))$.
— Guards operate in the same way, that is, $\gamma^l(\{s_1, \ldots, s_n\}) = \underline{\sigma}_{i_1}(\gamma^l_{i_1}(s_1)) \wedge \ldots \wedge \underline{\sigma}_{i_n}(\gamma^l_{i_n}(s_n))$ and $\gamma^u_{i_1}(\{s_1, \ldots, s_n\}) = \underline{\sigma}_{i_1}(\gamma^u_{i_1}(s_1)) \wedge \ldots \wedge \underline{\sigma}_{i_n}(\gamma^u_{i_n}(s_n))$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

— 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 Lopes and Fiadeiro (2004), 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, that is, 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* $\mu : \langle Q_1, \Delta_1 \rangle \rightarrow \langle Q_2, \Delta_2 \rangle$ 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}, $\mu_{par-ev,e}$, $\mu_{par-sv,s}$, $\mu_{hr-ev,e}$, for every $e \in E_1$ and $s \in S_1$, are injective.
— Handling requirements are preserved: $(\eta_2(\mu_{hr-ev,e}(h)) \supset \underline{\mu}(\eta_1(h)))$ holds for every event $e \in E_1$ and handling $h \in H_1(e)$.
— Effects are preserved: $(\rho_2(\mu_{sv}(s)) \supset \underline{\mu}(\rho_1(s)))$ holds for every $s \in S_1$.
— Lower guards are preserved: $(\gamma_2^l(\mu_{sv}(s)) \supset \underline{\mu}(\gamma_1^l(s)))$ holds for every $s \in S_1$.
— Upper guards are reflected: $(\underline{\mu}(\gamma_1^u(s)) \supset \gamma_2^u(\mu_{sv}(s)))$ 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 since this would change the way other components may have been connected through the more abstract design.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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

Fig. 18. The design of *Actuator*.

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

Fig. 19. The design of *FBActuator*.

— 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. Note 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 the 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*, defined in Figure 18, 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.

Note that in this description we do not provide any details of what exactly the action does or when it is enabled, that is, 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* presented in Figure 19. This design refines *Actuator* by including feedback on the execution of *action* in the form of the publication of a new event actioned.

Notice that this design is no longer refined by either *Counter* or *Adder*.

An abstraction of *Set* that is more specific in the way it can relate to other components is presented in Figure 20. Apart from the state variables, this design has the same signature as *Set* up to renaming. As result, it offers the same interactions with 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 Fiadeiro *et al.* (2003). Figure 21 presents an example with the components involved in our running

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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

Fig. 20. The design of *2FBActuator&Par*.

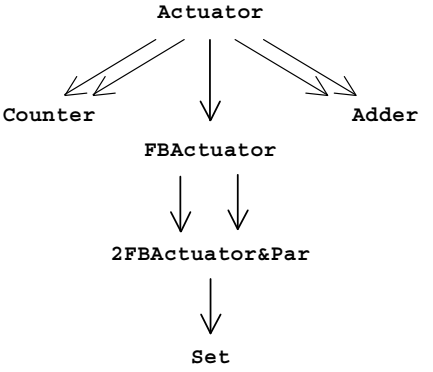

Fig. 21. Hierarchy of components of our running example.

example. Notice that, in order to represent refinement morphisms in diagrams, we use a different arrow from the one 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 rest of the paper, we assume a fixed a refinement morphism $\mu : \langle Q_1, \Delta_1 \rangle \to \langle Q_2, \Delta_2 \rangle$.

Proposition 5.2. A data signature morphism $\underline{\mu} : \Sigma_1 \to \Sigma_2$ is defined by the morphism 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 $e \in E_1$;
— $d^p : d^e \to d$ into $d^{\mu(p)} : d^{\mu(e)} \to d$, for every $e \in E_1$, $p \in P_1(e)_d$ and sort d in Σ;
— $inv^{h,s} : d^e \to d^s$ into $inv^{\mu(h),\mu(s)} : d^{\mu(e)} \to d^{\mu(s)}$, for every $e \in Sub(E_1)$, $h \in H_1(e)$ and $s \in G_1(h)$;
— $d^p : d^s \to d$ into $d^{\mu(p)} : d^{\mu(s)} \to d$, for every $s \in S_1$, $p \in P_1(s)_d$ and sort d in Σ;
— $pub^{s,e} : d^s \to d^e$ into $pub^{\mu(s),\mu(e)} : d^{\mu(s)} \to d^{\mu(e)}$, for every $s \in S_1$ and $e \in Pub(E_1)$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

Every such signature morphism $\mu : \Sigma_1 \to \Sigma_2$ induces a reduct functor $\_|_\mu$ from the algebras of Σ_2 to the algebras of Σ_1 (Ehrig and Mahr 1985). Such reducts extend to spaces in the sense that, applied to a Q_2-space $\langle \Sigma_2, \mathscr{U} \rangle$, we get $\langle \Sigma_2, \mathscr{U}|_\mu \rangle$ as a Q_1-space.

We omit the proof of this result because it is quite simple. However, we would like to point out 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 $\mathscr{U}$.

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

— t is an element of $d^e_{\mathscr{U}}$ for some event $e \in \mu_{ev}(Sub(E_1))$;
— $h \in H_1(e)$;
— $u = inv^{\mu(h),s}_{\mathscr{U}}(t)$ for some $s \in \mu_{sv}(G_1(h))$;
— $\langle t, \mu_{hr-ev,e}(h), u \rangle \in INV$.

That is, we get the set $INV|_\mu$ 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* $EST = \langle VAL, PDN \rangle$ for Q_2, which we denote by $EST|_\mu$, consists of:

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

That is, variables are evaluated in the reduct of a state in the same way that their translations are evaluated in the original state. With regard to pending invocations, as mentioned earlier, 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 reflect 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 $\langle SRC, TRG, INV, EXC, PUB, NXT \rangle$ of Q_2, its μ-reduct is the execution step $\langle SRC|_\mu, TRG|_\mu, INV|_\mu, EXC|_\mu, PUB|_\mu, NXT|_\mu \rangle$ of Q_1 where:

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

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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 $\mathscr{M}$ 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 $\mathscr{M}$ together with the direct acyclic graph of $\mathscr{M}$ and the labelling function $\mathscr{L}|_\mu$ that results from the application of the reduct to the labels provided by $\mathscr{L}$ (that is, $\mathscr{L}|_\mu$ assigns the execution state $\mathscr{L}(n)|_\mu$ to a node n and the execution step $\mathscr{L}(r)|_\mu$ 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 state the result that states that refinement morphisms are model preserving.

Definition/Proposition 5.8. Given a model $\mathscr{M}$ of a design $\langle Q_2, \Delta_2 \rangle$, its *$\mu$-reduct* $\mathscr{M}|_\mu$ is a model of the design $\langle Q_1, \Delta_1 \rangle$.

As required, the refinement of a design may only eliminate models, reflecting the fact that the degree of under specification is reduced. As a result, any refinement of a design preserves its properties. The proofs of Definitions/Propositions 5.6, 5.7 and 5.8 are given 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 either the other components or the interconnections that are in place. We state and prove our results for a special kind of colimits – pushouts – as this simple case generalises to the colimit of any finite diagram (Fiadeiro 2004).

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

Note 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 $\mathbf{dsgn}(Q_0)$ as a middle object in the given configuration (see Figure 22). 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 Fiadeiro (2004).

More information on the relationship between refinement and superposition, and the compositionality results that relate them can be found in Lopes and Fiadeiro (2004).

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

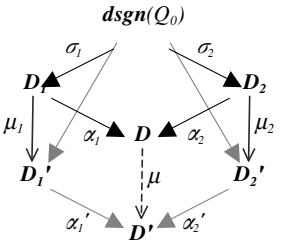

Fig. 22. Compositionality of refinement with respect to superposition.

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, suppose 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, in order to do this 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. Similarly, input and output channels are needed to make sure that data is exchanged synchronously. This is why we will now extend the notion of design in e-CommUnity with synchronisation constraints and communication channels.

As an example, consider the revision of *SCA* given in Figure 23. 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.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 23. The design of *syncSet&Counter&Adder*.

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* does exactly this, that is, it allows a component to make data available synchronously 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 $lm + mysum < LIM$.

We can now formalise the extension, starting with signatures.

Definition 6.1. We define an *extended signature* $Q^{I,O}$ to be 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 deals with 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, that is, to every data sort $d \in D$ and channel $c \in I_d \cap O_d$, VAL assigns a value $VAL(c) \in d_{\mathscr{D}}$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

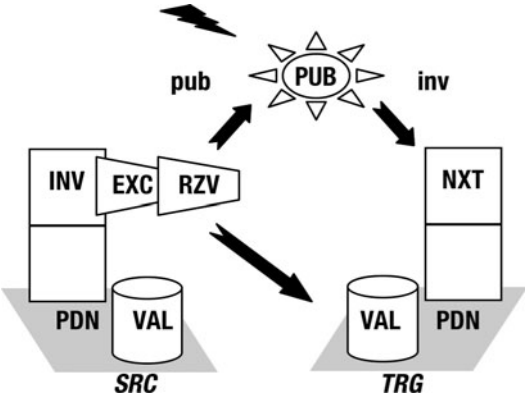

Fig. 24. Representation of the elements involved in extended execution steps.

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 $s \in S$, there is at most one $u \in d^s_{\mathcal{U}} \cap RZV$.

The set RZV contains the service instances that are executed during that step. It may exclude some of the instances in EXC, that is, 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 a model, we impose a 'maximality' constraint on RZV with respect to EXC that makes sure that only as many enabled invocations are discarded as necessary to satisfy the synchronisation constraints.

The Figure 24 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 $\Sigma = \langle D, F \rangle$ 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.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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

— $o_{\mathscr{D}} = VAL(o)$.
— $v_{\mathscr{D}} = VAL(v)$.

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 $\Sigma = \langle D, F \rangle$ 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 $\mathscr{D}$ with:

— $p_{\mathscr{D}} = d^p_{\mathscr{U}}(u)$ for $s \in S$, $p \in P(s)$ and $u \in d^s_{\mathscr{U}}$ such that $u \in d^s_{\mathscr{U}} \cap RZV$ if $d^s_{\mathscr{U}} \cap RZV \neq \varnothing$.
— s is true if and only if $d^s_{\mathscr{U}} \cap RZV \neq \varnothing$.

We use s as a proposition 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. Note 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 in 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, and every output channel $o \in O_d$ as a state variable of sort d. For every execution state, we extend every Σ-algebra $\mathscr{D}$ with $i_{\mathscr{D}} = VAL(i)$ and, for every execution step, $o_{\mathscr{D}} = VAL_{SRC}(o)$ and $o'_{\mathscr{D}} = VAL_{TRG}(o)$.

Given this, we can define designs in extended signatures.

Definition 6.7. An *extended design over* $Q^{I,O}$ is a tuple $\langle \eta, \rho, \gamma, \beta, \chi \rangle$ where $\langle \eta, \rho, \gamma \rangle$ 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 as follows.

Definition 6.8. A *model of an extended design* $\langle Q^{I,O}, \Delta \rangle$ where $\Delta = \langle \eta, \rho, \gamma, \beta, \chi \rangle$ is a model of $Q^{I,O}$ such that any label $\langle SRC, TRG, INV, EXC, RZV, PUB, NXT \rangle$ of an arrow of

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

the underlying graph satisfies the following conditions:

— For every $u \in EXC$ with $u \in d^s_{\mathscr{U}}$, we have $\gamma^l(s)$ holds for u at SRC.
— For every $\langle t, h, u\rangle \in INV$ and $u \in d^s_{\mathscr{U}}$, if $\gamma^u(s)$ holds for u at SRC, then $u \in EXC$.
— For every $u \in RZV$ with $u \in d^s_{\mathscr{U}}$, we have $\gamma^l(s)$ holds for u at SRC.
— For every $u \in RZV$ with $u \in d^s_{\mathscr{U}}$, we have $\rho(s)$ holds for u at that step.
— For every $t \in PUB$ where $t \in d^e_{\mathscr{U}}$ and $h \in H(e)$, we have $\eta(h)$ 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 $\langle SRC, \_, INV, EXC, RZV', \_, \_\rangle$ with $RZV' \supseteq RZV$ and $u \in RZV'$ such that all the previous conditions hold for that step.
— There is no step $\langle SRC, \_, INV, EXC, RZV', \_, \_\rangle$ 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 $\langle SRC, \_, INV, EXC, RZV', \_, \_\rangle$ such that $u \in RZV'$ and all the previous conditions hold for that step, then 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 the requirements for models of the 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 the original designs. The sixth and seventh conditions address the new sets of requirements on observations and synchronisations. The eighth condition captures, in a sense, a notion of 'maximality' with respect to 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 given by the eighth condition the fact that all options should be reflected in the same model.

Note 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 $(a \supset b)$ 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 with respect 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 will resume this discussion at the end of this section.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

```
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
```

Fig. 25. The design of *syncAdder*.

It remains to show how we can externalise the extension in much the same way as we did in Section 4. 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
```

In order to strengthen 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, presented in Figure 26, in which *syncAdder* is modelled as a component prepared for synchronous interaction (see Figure 25).

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*

$$\langle V_1, E_1, S_1, P_1, T_1, A_1, G_1, H_1, I_1, O_1 \rangle$$

and

$$\langle V_2, E_2, S_2, P_2, T_2, A_2, G_1, H_2, I_2, O_2 \rangle$$

is a morphism between signatures

$$\langle V_1, E_1, S_1, P_1, T_1, A_1, G_1, H_1 \rangle$$

and

$$\langle V_2, E_2, S_2, P_2, T_2, A_2, G_2, H_2 \rangle$$

together with $\sigma_{in} : I_1 \to I_2 \cap O_2$ and $\sigma_{out} : O_1 \to O_2$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

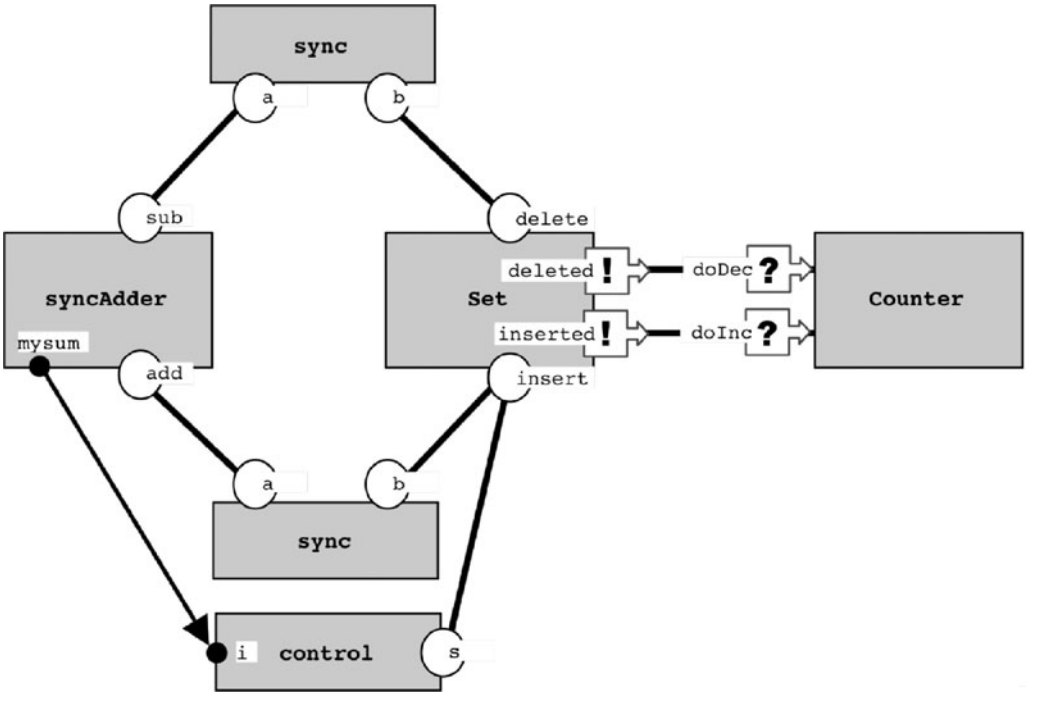

Fig. 26. A new configuration of *set–counter*.

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

Definition 6.10. A *morphism between* $\langle \eta_1, \rho_1, \gamma_1, \beta_1, \chi_1 \rangle$ and $\langle \eta_2, \rho_2, \gamma_2, \beta_2, \chi_2 \rangle$ is a morphism between $\langle \eta_1, \rho_1, \gamma_1 \rangle$ and $\langle \eta_2, \rho_2, \gamma_2 \rangle$ 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, that is, 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, that is, a system may impose new synchronisations among services of the component and new observations of the state of the component.

However, note that, at the level of synchronisation sets, morphisms operate in a contravariant 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 contravariant behaviour reflects the way signature morphisms were used in the previous generation of CommUnity.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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, note that this relationship does not extend to an adjunction: 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 (Fiadeiro 2004).

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.

7. Conclusions and further work

In this paper, we have presented an extended account of the formalisation of the architectural style known as 'publish/subscribe' or 'implicit invocation' that we started in Fiadeiro and Lopes (2006). Other formal models (see, for example, Dingel *et al.* (1998) and Garlan *et al.* (2003)) exist that abstract away from concrete notions of event and related notification mechanisms, but they just 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, that is, how connectors can be defined and superposed over components to coordinate their interactions.

For the computational model, we have proposed a mathematical semantics based on transition systems extended with the 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 (Alonso *et al.* 2004). See Fiadeiro *et al.* (2006; 2007) for a preliminary account of this approach.

We have defined several logics that support the 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 the 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.

For the architectural model, our formalisation has allowed us to characterise key structural properties of the architectural style for the externalisation of bindings and

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

mediators previously claimed in papers such as Sullivan and Notkin (1992). Terms like 'separate' and 'independent' were given a precise interpretation in our framework, which we claim is faithful to the one given in Sullivan and Notkin (1992): 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 required 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 dynamically superposed 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 Fiadeiro (2004).

Furthermore, the proposed categorical semantics has 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, that is, 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 fashion. We have proved that this process is compositional with respect to superposition, that is, 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 (Lopes and Fiadeiro 2004).

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 have added channels for (synchronous) input/output communication, and a rendez vous style of synchronisation of service executions. We have shown how these new forms of interaction do not increase the degree of coupling nor compromise the evolutionary properties of implicit invocation. In particular, we have shown how synchronous interactions may themselves be externalised in separate mediators, and how communication channels are not connected through explicit naming but through 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 (Lopes and Fiadeiro 2006) as well as context-awareness (Lopes and Fiadeiro 2005). However, we are still in the initial stages of what could be called 'architectural engineering', by which we mean the ability to identify, characterise and compose architectural 'aspects' to define an architectural style for a particular class of applications. Our current work on providing an algebraic approach to service-oriented architecture (Fiadeiro *et al.* 2007) should provide us with more insight into the engineering of architectural styles.

Appendix A. Notation

	Signatures
$A(s)$	The write frame (or domain) of service s, that is, 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. $s \in A(v)$ if and only if $v \in A(s)$.
D	The set of data sorts.
E	The set of all the events either published or subscribed to 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. $h \in G(s)$ if and only if $s \in G(h)$.
$H(e)$	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 $p \in P(s)_d$ is declared under *par p:d*.
S	The set of all services of a component. Each service is declared under *provide*.
$T(e)$	The type of event *e: pub* (published only), *sub* (subscribed only) or *pubsub* (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, that is, 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 $[\gamma^l(s), \gamma^u(s)]$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

$\gamma^u(s)$	The upper guard of service s, that is, 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.
$inv^{h,s}_{\mathscr{U}}(t)$	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.
$pub^{s,e}_{\mathscr{U}}(u)$	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 B. Proofs

Definition 5.6 Given an execution step

$$STP = \langle SRC, TRG, INV, EXC, PUB, NXT \rangle$$

of Q_2, its μ-reduct is the execution step

$$\langle SRC|_\mu, TRG|_\mu, INV|_\mu, EXC|_\mu, PUB|_\mu, NXT|_\mu \rangle$$

of Q_1 where

— $EXC|_\mu$ is the set of service instances $u \in EXC$ such that $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$, that is, that are instances of services in Q_1.

— $PUB|_\mu$ is the set of event instances $t \in PUB$ such that $t \in d^e_{\mathscr{U}}$ for some $e \in \mu_{ev}(E_1)$, that is, that are instances of events in Q_1.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

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

— For every $u \in EXC\,|_\mu$ there is $\langle t, h', u\rangle \in INV\,|_\mu$: If $u \in EXC\,|_\mu$, then $u \in$ 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 $\langle t, h, u\rangle \in INV$ such that:

- $t \in d^e_{\mathscr{U}}$ for some $e \in Sub(E_2)$.
- $h \in H_2(e)$.
- $u = inv^{h,s}_{\mathscr{U}}(t)$ and $s \in G_2(h)$; this is because algebras of spaces assign disjoint carrier sets to different services and $u \in d^s_{\mathscr{U}}$, that is, 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 $h' \in G_1(s')$ such that $\mu(h') = h$. 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 $e' \in E_1$ such that $\mu(e') = e$ and $h' \in H_1(e')$. It then follows that $\langle t, h', u\rangle \in INV\,|_\mu$.

— For every $\langle t, h', u\rangle \in NXT\,|_\mu$, $t \in PUB\,|_\mu$.

If $\langle t, h', u\rangle \in NXT\,|_\mu$, then $\langle t, \mu(h'), u\rangle \in NXT$ and $t \in d^s_{\mathscr{U}}$ for some $e \in \mu(Sub(E_1))$. Because STP is an execution step of Q_2, we have $t \in PUB$, so $t \in PUB\,|_\mu$.

— $PDN_{TRG}\,|_\mu = PDN_{SRC}\,|_\mu \setminus INV\,|_\mu \cup NXT\,|_\mu$.

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 $v \in V_1$ such that $VAL_{TRG}\,|_\mu\,(v) \neq VAL_{SRC}\,|_\mu\,(v)$, there is $u \in EXC\,|_\mu$ with $u \in d^s_{\mathscr{U}_{|_\mu}}$ 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 $u \in d^s_{\mathscr{U}}$ such that $\mu(v) \in A_2(s')$. Because μ is a refinement morphism, $\mu(v) \in A_2(s')$ implies that there is $s \in S_1$ such that $\mu(s') = s$ and $v \in A_1(s)$. This also implies that $u \in EXC\,|_\mu$. □

Definition 5.7 Given a model $\mathscr{M}$ 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 $\mathscr{M}$ together with the direct acyclic graph of $\mathscr{M}$ and the labelling function $\mathscr{L}|_\mu$ that results from the application of the reduct to the labels provided by $\mathscr{L}$ (that is, $\mathscr{L}|_\mu$ assigns the execution state $\mathscr{L}(n)\,|_\mu$ to a node n and the execution step $\mathscr{L}(r)\,|_\mu$ to an arrow r).

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

— For every arrow $r = \langle n_1, n_2\rangle$, $\mathscr{L}|_\mu\,(r)$ is of the form $\langle \mathscr{L}|_\mu\,(n_1), \mathscr{L}|_\mu\,(n_2), \_, \_, \_, \_\rangle$.

This follows trivially from the definition of $\mathscr{L}|_\mu$ and the fact that, because $\mathscr{M}$ is a model of Q_2, $\mathscr{L}(r)$ is of the form $\langle \mathscr{L}(n_1), \mathscr{L}(n_2), \_, \_, \_, \_\rangle$.

— $VAL_{TRG}\,|_\mu\,(v) = VAL_{TRG'}\,|_\mu\,(v)$ for every state variable $v \in V_1$ and every pair of arrows $r = \langle n, m\rangle$ and $r' = \langle n, m'\rangle$ such that there are $s \in A_1(v), u \in d^s_{\mathscr{U}_{|_\mu}} \cap EXC\,|_\mu$

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

$\cap EXC'|_\mu$, where $\mathscr{L}|_\mu(r)$ and $\mathscr{L}|_\mu(r')$ are of the form $\langle \_, TRG|_\mu, \_, EXC|_\mu, \_, \_\rangle$ and $\langle \_, TRG'|_\mu, \_, EXC'|_\mu, \_, \_\rangle$, respectively.

It follows from the hypothesis and the fact that μ is a refinement morphism that there are $\mu(s) \in A_2(\mu(v))$, $u \in d^{\mu(s)}_{\mathscr{U}} \cap EXC \cap EXC'$, where $\mathscr{L}(r)$ and $\mathscr{L}(r')$ are of the form $\langle \_, TRG, \_, EXC, \_, \_\rangle$ and $\langle \_, TRG', \_, EXC', \_, \_\rangle$, respectively. Given that $\mathscr{M}$ 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 $r = \langle n, m\rangle$ and $r' = \langle n, m'\rangle$ where $\mathscr{L}|_\mu(r)$ and $\mathscr{L}|_\mu(r')$ are of the form $\langle \_, \_, INV|_\mu, EXC|_\mu, \_, \_\rangle$ and $\langle \_, \_, INV'|_\mu, EXC'|_\mu, \_, \_\rangle$, respectively, and service instance u such that $\langle \_, \_, u\rangle \in INV|_\mu$ and $\langle \_, \_, u\rangle \in INV'|_\mu$, we have $u \in EXC|_\mu$ if and only if $u \in EXC'|_\mu$.

The hypothesis implies that $\mathscr{L}(r)$ and $\mathscr{L}(r')$ are of the form $\langle \_, \_, INV, EXC, \_, \_\rangle$ and $\langle \_, \_, INV', EXC', \_, \_\rangle$, respectively. Furthermore, $\langle \_, \_, u\rangle \in INV$ and $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$. Given that $\mathscr{M}$ is a model of Q_2, $u \in EXC$ if and only if $u \in EXC'$. Given that $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$, we conclude that $u \in EXC|_\mu$ if and only if $u \in EXC'|_\mu$. □

Definition 5.8 Given a model $\mathscr{M}$ of a design $\langle Q_2, \Delta_2\rangle$, its μ-reduct $\mathscr{M}|_\mu$ is a model of the design $\langle Q_1, \Delta 1\rangle$.

Proof. We begin by stating 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 a handler of an event e in Q_1, $t \in d^s_{\mathscr{U}|_\mu}$, s be a service in Q_1, $u \in d^e_{\mathscr{U}|_\mu}$, and $STP = \langle SRC, TRG, INV, EXC, PUB, NXT\rangle$ be an execution step for Q_2.

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

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

— For every $u \in EXC|_\mu$ with $u \in d^s_{\mathscr{U}|_\mu}$, we have $\gamma^l_1(s)$ holds for u at $SRC|_\mu$.

If $u \in EXC|_\mu$, then $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$ and $u \in EXC$. Given that $\mathscr{M}$ is a model of $\langle Q_2, \Delta_2\rangle$ and $d^s_{\mathscr{U}|_\mu} = d^{\mu(s)}_{\mathscr{U}}$, we have $\gamma^l_2(\mu(s))$ holds for u at SRC. Given that μ is a refinement morphism, $\gamma^l_2(\mu_{sv}(s)) \supset \mu(\gamma^l_1(s))$ holds and, hence, $\mu(\gamma^l_1(s))$ holds for u at SRC. As a consequence of the auxiliary result enunciated above, $\gamma^l_1(s)$ holds for u at $SRC|_\mu$.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

— For every $\langle t, h, u\rangle \in INV|_\mu$ and $u \in d^s_{\mathscr{U}_{|_\mu}}$, if $\gamma_1^u(s)$ holds for u at $SRC|_\mu$, then $u \in EXC|_\mu$.

On the one hand, if $\langle t, h, u\rangle \in INV|_\mu$ and $u \in d^s_{\mathscr{U}_{|_\mu}}$, then $\langle t, \mu(h), u\rangle \in INV$ and $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$. On the other hand, as a consequence of the auxiliary result enunciated above, if $\gamma_1^u(s)$ holds for u at $SRC|_\mu$, then $\mu(\gamma_1^u(s))$ holds for u at SRC. Given that μ is a refinement morphism, $\mu(\gamma_1^u(s)) \supset \gamma_2^u(\mu_{sv}(s))$ holds, so $\gamma_2^u(\mu_{sv}(s))$ holds for u at SRC. Given that $\mathscr{M}$ is a model of $\langle Q_2, \Delta_2\rangle$ and $d^s_{\mathscr{U}_{|_\mu}} = d^{\mu(s)}_{\mathscr{U}}$, we know that $u \in EXC$. Because $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$, we conclude that $u \in EXC|_\mu$.

— For every $u \in EXC|_\mu$ with $u \in d^s_{\mathscr{U}_{|_\mu}}$, we have $\rho_1(s)$ holds for u at $STP|_\mu$.

If $u \in EXC|_\mu$, then $u \in d^s_{\mathscr{U}}$ for some $s \in \mu_{sv}(S_1)$ and $u \in EXC$. Given that $\mathscr{M}$ is a model of $\langle Q_2, \Delta_2\rangle$ and $d^s_{\mathscr{U}_{|_\mu}} = d^{\mu(s)}_{\mathscr{U}}$, we have $\rho_2(\mu(s))$ holds for u at STP. Given that μ is a refinement morphism, $\rho_2(\mu_{sv}(s)) \supset \mu(\rho_1(s))$ holds, so $\mu(\rho_1(s))$ holds for u at STP. As a consequence of the auxiliary result stated above, $\rho_1(s)$ holds for u at $STP|_\mu$.

— For every $t \in PUB|_\mu$ where $t \in d^e_{\mathscr{U}_{|_\mu}}$ and $h \in H_1(e)$, we have $\eta_1(h)$ holds for t and h at $STP|_\mu$.

If $t \in PUB|_\mu$, then $t \in d^e_{\mathscr{U}}$ 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 $\mathscr{M}$ is a model of $\langle Q_2, \Delta_2\rangle$ and $d^e_{\mathscr{U}_{|_\mu}} = d^{\mu(e)}_{\mathscr{U}}$, we have $\eta_2(\mu(h))$ holds for t and $\mu(h)$ at STP. Again because μ is a refinement morphism, $\eta_2(\mu(h)) \supset \mu(\eta_1(h))$ holds, so $\mu(\eta_1(h))$ holds for t and $\mu(h)$ at STP. As a consequence of the auxiliary result stated above, $\eta_1(h)$ holds for t and h at $STP|_\mu$. □

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

Proof. We begin by noting that there is a forgetful functor rsign from **rDSGN** to **SIGN** that forgets everything from designs except their signatures. The fact that $\mathbf{dsgn}(Q_0)$ is a discrete lift ensures that signature morphisms $\mathbf{sign}(\sigma_i); \mathbf{rsign}(\mu_i)$ give rise to morphisms $\sigma_i; \mu_i : \mathbf{dsgn}(Q_0) \to D_i'$ in **sDSGN**.

Given that **sign** preserves pushouts, we have that $\langle \mathbf{sign}(\alpha_1), \mathbf{sign}(\alpha_2)\rangle$ is a pushout of $\langle \mathbf{sign}(\sigma_1), \mathbf{sign}(\sigma_2)\rangle$ in **SIGN**. Because $\langle \mathbf{sign}(\alpha_1'), \mathbf{sign}(\alpha_2')\rangle$ is a candidate for being a different pushout, from the universal property of pushouts, it follows that there exists a unique morphism $\mu : \mathbf{sign}(D) \to \mathbf{sign}(D')$ 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 $\mu : D \to D'$ in **sDSGN**. We will 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}, $\mu_{par-ev,e}$, $\mu_{par-sv,s}$, $\mu_{hr-ev,e}$, 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.

https://doi.org/10.1017/S0960129507006299 Published online by Cambridge University Press

— Upper guards are reflected, that is, $(\mu(\gamma^u(s)) \supset \gamma'^u(\mu_{sv}(s)))$ holds for every $s \in S$.

As explained at the end of Section 4, if we use $\{s_1, \ldots, s_n\}$ to denote a quotient set of amalgamated services, we have $\gamma^u(\{s_1, \ldots, s_n\}) = \alpha_{i_1}(\gamma^u_{i_1}(s_1)) \wedge \ldots \wedge \alpha_{i_n}(\gamma^u_{i_n}(s_n))$ where α'_{i_j} is either α_1 or α_2, depending on whether s_j belongs to S_1 or S_2. Similarly, if we use $\{s'_1, \ldots, s'_m\}$ to denote a quotient set of amalgamated services of designs D'_1 and D'_2, we have $\gamma'^u(s'_1, \ldots, s'_n) = \alpha'_{i_1}(\gamma^u_{i_1}(s'_1)) \wedge \ldots \wedge \alpha'_{i_n}(\gamma^u_{i_n}(s'_n))$ where α'_{i_j} is either α'_1 or α'_2, depending on whether s'_j belongs to S'_1 or S'_2. We prove that, for every s'_j in the quotient set corresponding to $\mu(s)$ (that is, s'_j in D'_1 or D'_2 such that $\alpha_{i_j}(s'_j) = \mu_{sv}(s)$), $\mu(\alpha_{i_1}(\gamma^u_{i_1}(s_1))) \wedge \ldots \wedge \mu(\alpha_{i_n}(\gamma^u_{i_n}(s_n))) \supset \alpha'_{i_j}(\gamma'^u_{i_j}(s'_j))$ holds.

It is not difficult to conclude that for every s'_j in the quotient set corresponding to $\mu(s)$ there exists an s_j in the quotient set corresponding to s such that $\mu_{i_j}(s_j) = s'_j$. Because μ_{i_j} is a refinement morphism, we have that $\mu_{i_j}(\gamma^u_{i_j}(s_j)) \supset \gamma'^u_{i_j}(\mu_{i_j}(s_j)))$ holds, that is, $\mu_{i_j}(\gamma^u_{i_j}(s_j)) \supset \gamma'^u_{i_j}(s'_j)$ holds. Then, $\alpha'_{i_j}(\mu_{i_j}(\gamma^u_{i_j}(s_j))) \supset \alpha'_{i_j}(\gamma'^u_{i_j}(s'_j))$ also holds. The result follows trivially from the equalities $\alpha_i; \mu = \mu_i; \alpha'_i$. □

Acknowledgements

This work was partially supported through the IST-2005-16004 Integrated Project *SENSORIA: Software Engineering for Service-Oriented Overlay Computers*. Antónia Lopes was partially supported by a grant from *Fundação para a Ciência e Tecnologia* during an extended stay at the University of Leicester. This work was developed while José Fiadeiro was on study leave.

References

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004) *Web Services*, Springer-Verlag.

Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O. and Spiteri, M. (2000) Generic support for distributed applications. *IEEE Computer* **33** (3) 68–76.

Bradbury, J. and Dingel, J. (2003) Evaluating and improving the automatic analysis of implicit invocation systems. In: *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference*, ACM Press 78–87.

Carzaniga, A., Rosenblum, D. and Wolf, A. (2001) Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems* **19** 283–331.

Dingel, J., Garlan, D., Jha, S. and Notkin, D. (1998) Towards a formal treatment of implicit invocation. *Formal Aspects of Computing* **10** 193–213.

Ehrig, H. and Mahr, B. (1985) *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, EATCS Monographs on Theoretical Computer Science **6**.

Eugster, P., Felber, P., Guerraoui, R. and Kermarrec, A.-M. (2003) The many faces of publish/subscribe. *ACM Computing Surveys* **35** (2) 114–131.

Fiadeiro, J. L. (2004) *Categories for Software Engineering*, Springer-Verlag.

Fiadeiro, J. L. and Lopes, A. (1997) Semantics of architectural connectors. In: Bidoit, M. and Dauchet, M. (eds.) TAPSOFT: Theory and Practice of Software Development. *Springer-Verlag Lecture Notes in Computer Science* **1214** 505–519.

Fiadeiro, J. L. and Lopes, A. (2006) A formal approach to event-based architectures. In: Baresi, L. and Heckel, R. (eds.) Fundamental Aspects of Software Engineering. *Springer-Verlag Lecture Notes in Computer Science* **3922** 18–32.

Fiadeiro, J. L., Lopes, A. and Bocchi, L. (2006) A Formal Approach to Service Component Architecture. In: Bravetti, M., Nez, M. and Zavattaro, G. (eds.) Web Services and Formal Methods, Third International Workshop. *Springer-Verlag Lecture Notes in Computer Science* **4184** 193–213.

Fiadeiro, J. L., Lopes, A. and Bocchi, L. (2007) Modules for service-component architectures. In: Fiadeiro, J. and Schobbens, P-Y. (eds.) Current Trends in Algebraic Development Techniques. *Springer-Verlag Lecture Notes in Computer Science* **4409** 37–55.

Fiadeiro, J. L., Lopes, A. and Wermelinger, M. (2003) A mathematical semantics for architectural connectors. In: Backhouse, R. and Gibbons, J. (eds.) Generic Programming. *Springer-Verlag Lecture Notes in Computer Science* **2793** 190–234.

Garlan, D., Khersonsky, S. and Kim, J. (2003) Model checking publish-subscribe systems. In: Ball, T. and Rajamani, S. (eds.) Model Checking Software. *Springer-Verlag Lecture Notes in Computer Science* **2648** 166–180.

Garlan, D. and Notkin, D. (1991) Formalizing design spaces: Implicit invocation mechanisms. In: Prehn, S. and Toetenel, W. J. (eds.) VDM'91: Formal Software Development Methods. *Springer-Verlag Lecture Notes in Computer Science* **551** 31–44.

Goguen, J. (1973) Categorical foundations for general systems theory. In: Pichler, F. and Trappl, R. (eds.) *Advances in Cybernetics and Systems Research*, Transcripta Books 121–130.

Katz, S. (1993) A superimposition control construct for distributed systems. *ACM TOPLAS* **15** (2) 337–35.

Lopes, A. and Fiadeiro, J. L. (2004) Superposition: composition versus refinement of non-deterministic action-based systems. *Formal Aspects of Computing* **16** (1) 5–18.

Lopes, A. and Fiadeiro, J. L. (2005) Algebraic semantics of design abstractions for context-awareness. In: Fiadeiro, J. L., Mosses, P. and Orejas, F. (eds.) Algebraic Development Techniques. *Springer-Verlag Lecture Notes in Computer Science* **3423** 79–93.

Lopes, A. and Fiadeiro, J. L. (2006) Adding mobility to software architectures. *Science of Computer Programming* **61** (2) 114–135.

Meier, R. and Cahill, V. (2002) Taxonomy of distributed event-based programming systems. In: *Proceedings of the International Workshop on Distributed Event-Based Systems*, IEEE Computer Society 585–588.

Misra, J. and Cook, W. (2006) Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modelling* **6** (1) 83–110.

Sullivan, K. and Notkin, D. (1992) Reconciling environment integration and software evolution. *ACM TOSEM* **1** (3) 229–268.