Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications

John Field[1], Maria-Cristina Marinescu[1], and Christian Stefansen[2]

[1] IBM T.J. Watson Research Center
{jfield,mariacm}@us.ibm.com
[2] Department of Computer Science, University of Copenhagen
cstef@diku.dk

Abstract. Our aim is to define the kernel of a simple and uniform programming model—the *reactor* model—suitable for building and evolving internet-scale programs. A reactor consists of two principal components: mutable state, in the form of a fixed collection of *relations*, and code, in the form of a fixed collection of *rules* in the style of datalog. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's state. As in classical process calculi, the reactor model accommodates collections of distributed, concurrently executing processes. However, unlike classical process calculi, our observable behaviors are sequences of states, rather than sequences of messages. Similarly, the interface to a reactor is simply its state, rather than a collection of message channels, ports, or methods. One novel feature of our model is the ability to compose behaviors both synchronously and asynchronously. Also, our use of datalog-style rules allows aspect-like composition of separately-specified functional concerns in a natural way .

1 Introduction

In modern web applications, the traditional boundaries between browser-side presentation logic, server-side "business" logic, and logic for persistent data access and query are rapidly blurring. This is particularly true for so-called web mash-ups, which bring a variety of data sources and presentation components together in a browser, often using asynchronous ("AJAX") logic. Such applications must currently be programmed using an agglomeration of data access languages, server-side programming models, and client-side scripting models; as a consequence, programs have to be entirely rewritten or significantly updated to be shifted between tiers. The large variety of languages involved also means that components do not compose well without painful amounts of scaffolding. Our ultimate goal is thus to design a uniform programming language for web applications, other human-driven distributed applications, and distributed business processes or web services which can express application logic, user interaction, and application logic using the same basic programming constructs. Ideally, such a language should also simplify composition, evolution, and maintenance of distributed applications. In this paper, we define a kernel programming model which is intended to address these issues and serve as a foundation for future work on programming languages for internet applications.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 76–95, 2007.

The reactor model is a synthesis and extension of key ideas from three linguistic foundations: synchronous languages [2, 4, 7], datalog [15], and the actor model [1]. From datalog, we get an expressive, declarative, and readily composable language for data query. From synchronous languages, we get a well-defined notion of "event" and atomic event handling. From actors, we get a simple model for dynamic creation of processes and asynchronous process interaction.

A reactor consists of two principal components: mutable state, in the form of a fixed collection of *relations*, and code, in the form of a fixed collection of *rules* in the style of datalog [15]. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's state. When a stimulus occurs, the reactor's rules are applied concurrently and atomically in a *reaction* to yield a *response* state, which becomes the initial state for the next reaction. In addition to determining the response state, evaluation of rules in a reaction may spawn new reactors, or generate new stimuli for the currently executing reactor or for other reactors. Importantly, newly-generated stimuli are processed *asynchronously*, in later reactions. However, we provide a simple mechanism to allow collections of reactors to react together as a unit when appropriate, thus providing a form of distributed atomic transaction.

As in classical process calculi, e.g., pi [12], the reactor model accommodates collections of distributed, concurrently executing processes. However, unlike classical process calculi, our observable behaviors are sequences of *states*, rather than sequences of *messages*. Similarly, the interface to a reactor is simply its state ("REST" style [6]), rather than a collection of message channels, ports, or methods. We accommodate information hiding by preventing certain relations in a reactor's state from being externally accessible, and by allowing the public relations to serve as abstractions of more detailed private state (as in database views). A significant advantage of using data as the interface to a component, and datalog as a basis for defining program logic, is that the combination is highly "declarative": it allows separately-specified state updates (written as rules) to be composed with minimal concern for control- and data-dependence or evaluation order.

Contributions. We believe that the reactor model is unique in combining the following attributes harmoniously in a single language: (1) data, rather than ports or channels as the interface to a component; (2) synchronous and asynchronous interaction in the same model, with the ability to generate processes dynamically; (3) expressive data query and transformation constructs; (4) the ability to specify constraints/assertions as a natural part of the core language; (5) distributed atomic transactions; and (6) declarative, compositional specification of functionality in an "aspect-like" manner.

2 Reactor Basics

A reactor consists of a collection of *relations* and *rules*, which together constitute a reactive, atomic, stateful unit of distribution. The full reactor syntax is given in Fig. 2.

Consider the declaration for `OrderEntryA` in Fig. 1. `OrderEntryA` defines a class of reactors that are intended to log orders—say, for an on-line catalog application. Reactor *instances* are created dynamically, using a mechanism we will describe shortly.

```
def OrderEntryA = {
  (* orders: id, itemid, qty *)
  public orders: (int, int, int).
  (* log: id, itemid, qty *)
  log: (int, int, int).

  log(id, itemid, qty) <-
    orders(id, itemid, qty).
}

def OrderEntryA' = {
  (* ... same as OrderEntryA ... *)

  not log(id, itemid, qty) <-
    not orders(id, itemid, qty).
}
```

```
def OrderEntryB = {
  (* ... same as OrderEntryA ... *)

  (* orderIsNew is true if order has
     not previously been logged  *)
  ephemeral orderIsNew: ().
  orderIsNew() <-
    orders(id, itemid, qty),
    not -log(id, itemid, qty)
}

def OrderEntryC = {
  (* ... same as OrderEntryB ... *)

  (* delete any new order whose id is
     duplicate of a prev. logged id  *)
  not orders(id, itemid, qty) <-
    ^orders(id, itemid, qty),
    not -log(id, itemid, qty),
    -log(id, _, _).
}
```

Fig. 1. Order entry: variations

Relations. The state of a reactor is embodied in a fixed collection of *persistent relations*. Relations are sets of $(\tau_1, ..., \tau_n)$ tuples, where each τ_i is one of the types `int`, `string`, *enum-type-name*, or `ref` *reactor-type-name*. The primitive types have the usual meanings. Enumerations introduce a new type ranging over a finite set of constants. *Reactor references*, of the form `ref` *reactor-type-name*, are described in Section 4. Relations are empty when a reactor is instantiated. In addition to persistent relations, whose values persist between reactions, a reactor can declare *ephemeral* relations. These relations can be written and read in the same manner as persistent relations, but they are re-initialized with every reaction.

In the case of `OrderEntryA`, its state consists of two persistent relations, `orders` and `log`, each of which is a collection of 3-tuples of integer values. Relation `orders` has *access annotation* `public`, which means that the contents of `orders` may be read or updated by any client. By "update", we simply mean that tuples may be added to or deleted from `orders`; no other form of update is possible. Relation `log`, lacking any access annotation, is *private*, the default, and may thus only be read or updated by the reactor that contains `log`.

```
        REACTOR ::= def reactor-type-name = { {DECL .}* }
           ENUM ::= enum enum-type-name = { {atom-name ,}+ }
           DECL ::= ( RELATION-DECL | RULE-DECL ) .
  RELATION-DECL ::= [public | public write | publiac read][ephemeral] rel-name : ( {TYPE ,}* )
      RULE-DECL ::= HEAD-CLAUSE <- BODY
           BODY ::= {BODY-CLAUSE ,}+
    HEAD-CLAUSE ::= [not] [var-name .]rel-name[^] ( {(_ | var-name | new reactor-name) ,}* )
    BODY-CLAUSE ::= [not] [var-name .][^|-]rel-name ( {(_ | var-name) ,}* ) | BASIC-PREDICATE
BASIC-PREDICATE ::= EXP (< | > | <> | =) EXP
           TYPE ::= int | string | enum-type-name | ref reactor-type-name
            EXP ::= var-name | NUMERIC-LITERAL | STRING-LITERAL | EXP (+ | - | * | / | %) EXP | self
```

Fig. 2. Reactor syntax

The reaction process. A reaction begins when a reactor receives an *update bundle* from an external source. An update bundle is a total map from the set of relations of the recipient to pairs of sets (Δ^+, Δ^-), where Δ^+ and Δ^- are sets of tuples to be added and deleted, respectively, from the target relation, and $\Delta^+ \cap \Delta^- = \emptyset$. An update bundle should contain at least one non-empty set, i.e. completely empty update bundles are not well-formed. In the examples that follow, an update bundle will typically contain an update to a single relation, usually adding or deleting only a single tuple. However, an update bundle can in general update any of the public relations of a reactor, and add and delete arbitrary number of tuples at a time.

The state of a reactor before an update bundle is received is called its *pre-state*. A reaction begins when an update bundle is applied atomically to the pre-state of a reactor, yielding its *stimulus state*. The stimulus state of a reaction is (conceptually) a copy of each relation of the reactor with the corresponding updates from the update bundle applied. So, for example, in the case of `OrderEntryA`, if relation `orders` contained the single tuple $(0, 1234, 3)$ prior to a reaction, and a reaction is initiated by applying an update bundle with $\Delta^+ = \{(1, 5667, 2)\}$ and $\Delta^- = \emptyset$, then the stimulus value of `orders` at the beginning of the reaction will be the relation $\{(0, 1234, 3), (1, 5667, 2)\}$. We will refer to the "value of relation r in the stimulus state" and "the stimulus value of r" interchangeably.

If a reactor contains no rules, the state of its relations at the end of a reaction—its *response state*—is the same as the stimulus state, and the reaction stores the stimulus values back to the corresponding persistent relations. Hence in its simplest form, a reaction is simply a state update. However, most interesting reactors have one or more rules which compute a response state distinct from the reactor's stimulus state (Section 3). Rule evaluation can also define sets of additions and deletions to/from the *future state* of either local relations or—via reactor references—relations of other reactors. These sets form the update bundles—one bundle per reactor instance referenced in a reaction—that initiate subsequent reactions in the same or other reactors. Update bundles thus play a role similar to messages in message-passing models of asynchronous computation.

It is important to note that from the point of view of an external observer, a reaction occurs *atomically*, that is, no intermediate states of the evaluation process are externally observable, and no additional update bundles may be applied to a reactor until the current reaction is complete.

Fig. 3 illustrates the life cycle of a typical collection of reactors, both from the point of view of an external observer (the top half of the figure) and internally (the bottom half of the figure schematically depicts reactor M during reaction i). The pre-state of reactor M during reaction i is labeled $-S_i$, its stimulus state is labeled $\hat{}S_i$, its future state is labeled $S\hat{}_i$, and its response state is labeled S_i. The terms pre-state, stimulus state, response state, and future state are meaningful only *relative* to a particular reaction, because one reaction's response state becomes the next reaction's pre-state, and references to a reactor's future state are used (along with the pre-state) to define the stimulus state for a subsequent reaction. The only "true" state, which persists between reactions, is the response state. An external observer therefore sees only a sequence of response states (more specifically, the response values of public relations). Each rule of a reactor can refer programmatically to relation values in all four states: it can read the

pre-state of a relation (schematically depicted as `-r` in Fig. 3), the stimulus state (`^r`), and the response state (`r`); it can write the response state and the future state (`r^`).

3 Rules

Basic rule evaluation. Reactor rules (Fig. 2) are written in the style of datalog [15, 14]. The single rule of `OrderEntryA` can be read as "ensure that `log` contains whatever tuples are in `orders`". The right-hand side, or *body* of a reactor rule consists of one or more *body clauses*. In `OrderEntryA`, there is only one body clause, a *match predicate* of the form `orders(id, itemid, qty)`. A match predicate is a pattern which binds instances of elements of tuples in the relation named by the pattern (here, `orders`) to variables (here, `id`, `itemid`, and `qty`). As usual, we use '_' to represent a unique, anonymous variable. Evaluation of the rule causes the body clause to be matched to *each* tuple of `orders` and binds variables to corresponding tuple elements. Since the *head clause* on the left side of the rule contains the same variables as the body clause, it ensures that `log` will contain every tuple in `orders`.

In general, a RULE-DECL can be read "for every combination of tuples that satisfy BODY, ensure that the HEAD-CLAUSE is satisfied" (by adding or deleting tuples to the relation in HEAD-CLAUSE). The semantics of datalog rule evaluation ensures that no change is made to any relation unless necessary to satisfy a rule, and—for our chosen semantics—that rule evaluation yields a unique fixpoint result in which all rules are satisfied. Although our rule evaluation semantics is consistent with standard datalog semantics, we have made several significant extensions, including head negation, reference creation, and the ability to refer to remote reactor relations via reactor references.

Returning to reactor `OrderEntryA`, let us consider the case where the pre-state values of `orders` and `log` are, respectively, $\{(0, 1234, 3)\}$ and$\{(0, 1234, 3)\}$, and an update bundle has $\Delta^+ = \{(1, 5667, 2)\}$ and $\Delta^- = \emptyset$. Then the stimulus value of `orders` will be equal to $\{(0, 1234, 3), (1, 5667, 2)\}$. No rule affects the value of `orders`, so the response value of `orders` will be the same as the stimulus value. In the case of `log`, rule evaluation yields the response state $\{(0, 1234, 3), (1, 5667, 2)\}$, i.e., the least change to `log` consistent with the rule.

Now, starting with the result of the previous `OrderEntryA` reaction described above, consider the effect of applying another update bundle such that $\Delta^+ = \emptyset$ and $\Delta^- = \{(0, 1234, 3)\}$. This reaction will begin by deleting $(0, 1234, 3)$ from `orders`, yielding the stimulus state `orders` $= \{(1, 5667, 2)\}$, `log` $= \{(0, 1234, 3), (1, 5667, 2)\}$. Evaluating the rule after the deletion has no net effect on `log` (since the only remaining tuple in `orders` is already in `log`), hence we get the response state `orders` $= \{(0, 1234, 3)\}$ and `log` $= \{(0, 1234, 3), (1, 5667, 2)\}$. We thus see that the effect of this rule is to ensure that `log` contains every orderid ever seen in `orders`. If we wanted to ensure that `log` is maintained as an *exact* copy of the current value of `orders` (which would mean that it is no longer a log at all), we could add the additional *negative* rule depicted in definition `OrderEntryA'` in Fig. 1. The negative rule of `OrderEntryA'` has the effect of ensuring that if an orderid is not present in `orders`, it will also be absent from `log`; i.e., it encodes tuple *deletion*. While negation is commonly allowed in body clauses for most datalog dialects, negation on the head of a rule is much less common (though not unheard of, see, e.g., [16]).

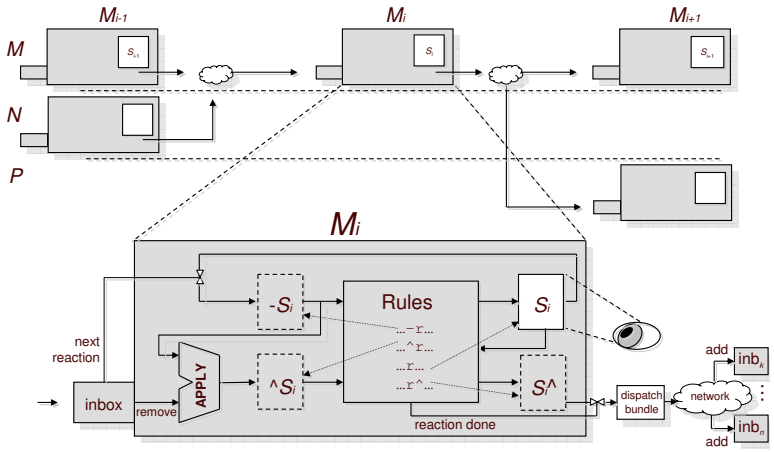

Fig. 3. Reaction schematic

Pre-state value and stimulus values of relations. Reactor definitions `OrderEntryB` and `OrderEntryC` of Fig. 1 add additional rules that further refine the behavior of `OrderEntryA`, using references to both pre-state and stimulus values of relations. `OrderEntryB` defines an *ephemeral* nullary relation `orderIsNew`, which functions as a boolean variable, initially false. The new rule in `OrderEntryB` sets `orderIsNew` to true (i.e., adds a nullary tuple) if `orders` contains a value not found in `log` prior to the reaction (i.e., in `log`'s pre-state value). The definition of `OrderEntryC` further refines `OrderEntryB` by causing any new order whose orderid is a duplicate of a previously logged orderid to be deleted. The new rule in `OrderEntryC` must distinguish the stimulus value of `orders`, i.e., `^orders` from its response value, i.e., `orders`, since the rule defines the response value to be something different from the stimulus value in the case where a duplicate order `id` is present.

It is important to note that the result of rule evaluation is oblivious to the order in which rules are declared. We believe this feature makes it much easier to update the functionality of a reactor by changing the rule set without concern for control- or data-dependencies. We see this feature demonstrated in the progression of examples depicted in Fig. 1, where rules can be mixed and matched liberally to yield updated functionality. Thus rules allow orthogonal functional "concerns" to be specified in an aspect-like fashion [10].

Initialization, constants, and reaction failure. Consider the pair of rules `r(x) <- s(x)` and `not r(x) <- s(x)`. These rules are inherently contradictory, since they require that `x` be both present and absent from relation `r`. In such cases, a *conflict* results. Because rules are conditionally evaluated, conflicts cannot in general be detected statically and must be detected during rule evaluation. If such a conflict occurs, the reaction *fails*: the reactor rolls back to its pre-state and no update bundles are dispatched.

Consider the reactor definition `Cell` depicted in Fig. 4. Each instance of a `Cell` is intended to hold exactly one value. Instances of `Cell` contain two relations: a public unary relation `val` containing the publicly-accessible value of the cell, and a private

```
def Cell = {
  public val:(int).
  live:().

  (* initializations *)
  (*1*) live() <- .
  (*2*) val(0) <- not -live().

  (* singleton constraint: if not
     satisfied, reaction fails;
     reactor rolls back *)
  (*3*) not live() <-
          val(x), val(y), x <> y.
}
```

Fig. 4. Cell

nullary (i.e., boolean) relation `live`. Recall that a reactor's relations are initially empty when the reactor is instantiated. Rules (1) and (2) together define an idiom which will allow us to initialize relations to non-empty values. First, consider rule (1). Rule (1) is an *unconditional rule*, and is an instance of the shorthand notation depicted in Fig. 5(b). Rule (1) defines `live` to be a a *constant*, since its response value evaluates to non-empty (i.e., "true") at the end of every reaction. Because of rule (1), `-live` in rule (2) is nonempty only during the first reaction in which the `Cell` is instantiated. Hence `val` will be initialized to 0 only once, in the reaction in which `Cell` is created. Thereafter, `-live` will be non-null, and the initialization will not recur, allowing `val` to be freely updated to arbitrary values.

Finally, consider rule (3) of `Cell`. The three clauses in its body collectively check to see whether `val` contains more than one value, i.e., whether it is a singleton. If not, the rule requires that its goal clause (left-hand side) be satisfied, i.e., that `live` be set to empty (false). However, any such attempt is inconsistent with the assertion in rule (1) that `live` is non-empty (true), hence any attempt to update `Cell` without maintaining the singleton invariant will result in a conflict and reaction failure. We thus see that the reactor model allows "assertions" and "integrity constraints" in the style of databases to be expressed in precisely the same form as rules that express state updates. When some assertion fails, the reaction rolls back. Fig. 5(d) depicts a notational convention that will allow us to use `FAIL` to define rules that represent assertions.

4 Asynchronous Reactor Composition

Up to this point, we have not explained how update bundles are generated, only how reactors react when an update bundle is applied. In this section, we show how updates are generated, and explain how this is intimately connected to asynchronous interaction.

Single-reactor asynchrony. Consider the reactor definition `Fibonacci` in Fig. 6, which computes successive values of a Fibonacci series. The relation `series` contains pairs whose first element is the ordinal position of the sequence value, and whose second element is the corresponding value of the sequence. The value of `series` is initialized using notational conventions (e) and (f) of Fig. 5. To compute the next element of the series, we need to first identify the last two elements of the series computed thus far. Universal quantification is required to determine the maximum element of a series; however, the body of a datalog rule can essentially encode only existential properties. To compute universal properties, we typically require auxiliary relations, thus, e.g., in `Fibonacci`, we use the ephemeral relation `notLargest` to contain all the indices of elements of `series` which are less than the maximum index.

	Notation	Translation	Comments
a	r1(*exp*$_0$) <-...ri(*exp*$_i$)... .	r1(x0) <- ...ri(xi)..., x0 = *exp*$_0$, ..., xi = *exp*$_i$.	Expressions exp$_i$ are instances of non-terminal EXP in Fig. 2; the xi are fresh variables.
b	*head* <- .	*head* <- 0 = 0.	
c	r(x1,...xn) := *body*.	r(x1,...xn) <- *body*. not r(x1',...xn') <- *body*, -r(x1',...xn'), x1' <> x1. ... not r(x1',...xn') <- *body*, -r(x1',...xn'), xn' <> xn.	
d	FAIL <-... .	not live() <-... .	Assumes the following definitions exist: live: (). live() <- not -live().

e	*body*$_0$:{
	head$_1$ <- *body*$_1$.
	...
	head$_n$ <- *body*$_n$.
	}
		head$_1$ <- *body*$_1$, *body*$_0$.
		...
		head$_n$ <- *body*$_n$, *body*$_0$.	

f	INIT	not -live()	Assumes same definitions as (d).
g	*head*$_1$, ··· , *head*$_n$ <- *body*.	*head*$_1$ <- *body*. ... *head*$_n$ <- *body*.	

Fig. 5. Notational conventions

```
def Fibonacci = {
  (* complete series thus far: 1st elt.
     is index, 2nd elt. is value *)
  public read series: (int, int).
  (* must be true for reactor to run *)
  public write run: ().
  (* holds indices in the sequence
     less than the maximum *)
  ephemeral notLargest: (int).

  INIT: {
    series(1,0), series(2,1) <- .
    run() <- .
  }

  (* indices in series less than max *)
  (*1*) notLargest(n) <- series(n, _),
          series(n', _), n' > n.

  (* compute next series value *)
  (*2*) series^(n, x1+x2) <-
          not notLargest(n),
           series(n-1, x1),
          series(n, x2).

  (* halts if "run" set to false *)
  (*3*) FAIL <- not -run(), not ^run().
}
```

Fig. 6. Self-Reacting Fibonacci

Note that the relation in the head of rule (2) computing the next value of the Fibonacci sequence has the form `series^`. A relation name of this form refers to the *future state* of the relation. The future state defines the contents of an update bundle which is processed *after* the current reaction ends, in a subsequent reaction. One can thus think of the future value of a relation as defining an asynchronous update or dispatching a "message". As a result, successive values of the series are separately visible to external observers as they are added to the list. Rule (3) results in failure if both the pre-state and the stimulus values of `run` are false, thus preventing further updates to the series from being generated. Instances of `Fibonacci` can react to two distinct classes of update bundles: "internally" generated update bundles containing only new values of the series, and client-generated update bundles which only affect the value of `run`. A client cannot update `series` since `series` is not public. The `Fibonacci` reactor does not produce update bundles affecting the value of `run`, since it has no rules referring to the future value of `run`.

In general, distinct reactors operate *concurrently* and *independently*. Given this fact, it is possible for an update bundle to be generated by a client attempting to update the value of `run` while a previous reaction by the same instance is in progress. Since reactions take place atomically, we must enqueue pending client updates until the current reaction is complete. To this end, every reactor has an associated *inbox* queue containing a multiset of pending update bundles. When a reaction is complete, the reactor checks for a new update bundle in the queue. If it exists, the reactor dequeues it and uses it to initiate a reaction. If no update bundle is present, the reactor performs no further computation until a new update arrives. We make no assumptions about the order in which inbox items are processed, except that they must be processed fairly. Fig. 3 illustrates this process.

Reactor references and multi-reactor asynchrony. Until now, our examples have only considered a single reactor type. Consider now the definitions for reactors `Sample`, `Sensor`, and `Nonce` depicted in Fig. 7. Reactor types `Sample` and `Sensor`

```
def Sample = {
  (* rSensor: ref. to sensor; assumed
     to be initialized by client *)
  public rSensor: (ref Sensor).
  (* samples collected thus far; nonces
     distinguish sample instances *)
  public log: (ref Nonce, int).
  (* pulse: set to collect sample *)
  public write ephemeral pulse: ().
  (* response: holds sensor response *)
  public write ephemeral response:
    (int).

  (* request sample when pulse set *)
  (*1*) s.req^(self) <-
          pulse(), rSensor(s).

  (* process response:
     add sample to log *)
  (*2*) log(new Nonce, r) <-
          response(r).
}

def Sensor = {
  (* set when sample is to
     be collected; value is ref.
     to sample reactor *)
  public write ephemeral req:
    (ref Sample).
  (* val: current sensor value *)
  public val:(int).

  (* send resp. when client sets req *)
  r.response^(v) <- val(v), req(r).

  (* sensor value is a singleton *)
  FAIL <- val(x), val(y), x <> y.
}

def Nonce = {}
```

Fig. 7. Asynchronous query/response

encode a "classical" asynchronous request/response interaction. To enable two reactor instances to communicate, we use *reactor references* such as those stored in relation `rSensor`. Rule (1) of `Sample` has the effect of dispatching an asynchronous request for the sensor value (maintained a `Sensor` reactor) whenever a client of `Sample` updates `pulse`. The expression `s.req^(self)` in Rule (1) contains an *indirect reference* to relation `rSensor`: after the reactor reference stored in relation `rSensor` is bound to variable `s`, we refer to relation `req` of the sensor instance indirectly using the expression `s.req`. Since we refer to the future value of `s.req`, an asynchronous update bundle is dispatched to the `Sensor` instance. The update bundle contains a self-reference to the requesting `Sample` instance, which is generated by the `self` construct.

A `Sensor` instance responds to a request (in the form of an update to relation `req`) by dispatching the current value of the sensor back to the corresponding `Sample` instance. It does so by by setting the `Sample`'s `response` relation via the reactor reference sent by the requester. The response is asynchronous, since `r.response^` refers to a future value. The requester processes the response from the `Sensor` instance by updating its `log` relation with the value of the response.

There are two ways of introducing references to reactors. The keyword `self` evaluates to a reference to the enclosing reactor. An expression of the form `new` *reactor-type-name* instantiates a new instance of the given reactor type. Instantiation expressions may only appear in the head of a rule. Rule (2) of `Sample` creates instances of the trivial reactor `Nonce`. A reactor reference is globally unique, hence trivial reactors such as `Nonce` can be used as sources of keys for relations. In particular, `Sample` uses instances of `Nonce` to distinguish multiple instances of the same sensor value. While `Nonces` contain no rules, in general, when a reactor is instantiated in a reaction, its rules are evaluated along with the rules of the parent reactor, as we shall see in Section 5.

In order to instantiate and connect `Sample` and `Sensor` instances together, another reactor must contain rules of the form `s.rSensor(new Sensor) <- theSampler(s)` and `theSampler(new Sample) <-` . A request-response cycle between `Sample` and `Sensor` instances requires three distinct reactions: the reaction in which a `Sample` client sets `pulse` (which dispatches the request to the sensor), the reaction in which the sensor responds to the request, and the reaction in which the requester updates the value of `log`.

5 Synchronous Reactor Composition

In the example in Fig. 8, an instance of `MiniBank` receives asynchronous requests to transfer money between accounts. As with the example in Fig. 7, we use references to "plumb" the reactors together. However, unlike the previous example, the remote references in Fig. 8 refer to *response* values of relations, not future values. This means that if a reaction is initiated by an update bundle containing a new `req` tuple at an instance of `MiniBank`, the scope of the reaction will *extrude* to include both of the account reactors (referred to by variables `to` and `from`, respectively). This results in a composite, synchronous, atomic reaction involving *three* reactor instances. Scope extrusion is

```
def Acct = {
  (* account balance *)
  public balance: (int).

  (* balance is a singleton *)
  FAIL <-
    balance(x), balance(y), x <> y.

  (* negative balances not allowed *)
  FAIL <- balance(x), x < 0.
}
```

```
def Minibank = {
  (* transfer request:transfer amount,
     to account, from accout *)
  public write ephemeral transferReq:
    (int, ref Acct, ref Acct).

  to.balance(x+amt) :=
    ^transferReq(amt, to, _).

  from.balance(y-amt) :=
     ^transferReq(amt, _, from).
}
```

Fig. 8. Classic Transaction

an inherently dynamic process, similar to a distributed transaction—see Section 6 for details. `MiniBank` uses the notation of Fig. 5(c) to define "assignments" to singleton relations.

Note that the rules in `Acct` encode constraints on the allowable values of `balance`. In a composite reaction, all of the rules of all of the involved reactors must be satisfiable in order for the reaction to succeed. If any of the rules fails, the composite reaction fails, and all of the reactors revert to their pre-reaction states. A composite reaction is always initiated at a single reactor instance at which some asynchronously-generated update bundle is processed—in the case of the example in Fig. 8, reactor instance `MiniBank`.

The example in Fig. 9 shows how multiple user interface components can be instantiated *dynamically* based on the current contents of an associated database. This mimics the process of building dynamic, data-driven user interface components. The basic idea of `DataDisplay` is that a button and an output field are generated for each item in a database. `ButtonWidget` and `OutputWidget` are reusable user interface components representing the button and output field generated for each item in relation `db`. The buttons thus generated are "active": pushing them causes the associated data to be updated, which in turn results in updates to the UI. For example, rule (8) "wires" together corresponding button and database items such that when a button is pressed, the corresponding data item is decremented. Rule (7) sets the value of the output field to the value of the quantity currently maintained in the database. The rules for newly-created reactors are evaluated as part of the "parent" reactor that created them.

6 Synchronous Reactions: Scope Extrusion and Locking Details

Scope Extrusion. In response to an update bundle, a reactor evaluates all its rules and while doing so it may extrude the scope of the reaction to include other reactors. Extrusion can happen in two ways: First, when a new reactor is instantiated it is included in the scope of the reaction that caused it to be instantiated. Second, when the response state of any relation (local or remote) is written, the reactor that contains that relation and all reactors containing rules reading that relation are included in the scope of the ongoing reaction. We say a relation is written whenever a rule produces a response-state update for that relation regardless if this results in a state change or not (e.g. adding an tuple that already exists constitutes a write). One important exception is the *passive read*. A passive read is a read from the pre-state of a remote relation. It differs from other remote reads in that writes to the remote relation in itself will not cause the reaction to extrude to the reader.

A reaction is complete when all reactors included in the reaction have reached a state that satisfies their rules. If one or more involved reactors cannot reach a state that satisfies their rules, there has been a *conflict* and all involved reactors roll back to their pre-state, i.e. the state they were in before the update bundle occurred or before they were included in the reaction. If different reactors involved in the same composite reaction separately define future values for relations of the same target reactor instance, the updates are combined into a *single* update bundle, which will be dispatched at the end of the composite reaction to the target reactor. In case the reaction rolls back, no update bundles are produced. In this sense, from the point of view of an external observer, a

```
def ButtonWidget = {
  public label: (string).
  public write ephemeral pressed: ().
}

def OutputWidget = {
  public label: (string).
  public val: (string).
}

def DataDisplay = {
  (* database: itemid, quantity *)
  public db: (int, int).

  (* list of button / output widget
     pairs,indexed by itemid *)
  widgets: (int, ref ButtonWidget,
            ref OutputWidget).

  (* labels of widgets are constants *)
  (*1*) o.label("Inventory: ") <-
          widgets(_, _, o).
  (*2*) b.label("Click to decr") <-
          widgets(_, b, _).

  (* projection of relns. on itemids *)
  ephemeral oldDisplayItems: (int).
  ephemeral currDbItems: (int).

  (*3*) oldDisplayItems(i) <-
          -widgets(i, _, _).
  (*4*) currDbtems(i) <- db(i, _).

  (* create new child widgets when new
     item added to db *)
  (*5*) widgets(i, new ButtonWidget,
                new OutputWidget) <-
          db(i, _),
          not oldDisplayItems(i).

  (* delete widgets if corresp. items
     removed from db *)
  (*6*) not widgets(i, _, _) <-
          -widgets(i, _, _),
          not currDbItems(i).

  (* output val set to qty of
     corresp. item *)
  {*7*) o.val(toString(q)) :=
          widgets(i, _, o), db(i, q).

  (* button decrements qty. of
     corresp. item *)
  (*8*) db(i, q-1) :=
          widgets(i, b, _),
          b.pressed().
}
```

Fig. 9. Data-Driven UI

composite reaction has the same atomicity properties as a reaction involving a single reactor. If a reaction updates the future state of (local or remote relations in) reactors $C_1, \ldots, C_n$, it produces n different update bundles—one per target reactor—and each C_i will have a separate and independent reaction to its own update bundle. Even if the scope of C_i's reaction should happen to expand to include some other C_j, C_j's update bundle will not be processed (or even visible) in that reaction.

Locking. Conceptually a rule *reads* all relations that appear in the body as well as any relation that appears in negated form in the head. Thus a rule's need for read access can be determined statically. Whether a rule will *write* the head relation when evaluated can generally only be determined at runtime, because the read has to match a non-empty set of facts that satisfy the body for a write to occur[1]. For this reason we will use the term *read* to refer to the static property, regardless of any optimization that might avoid unnecessary reads. The term *write*, on the other hand, will be strictly reserved for the dynamic property, i.e. a head relation is considered to be written only if the body yields at least one match when evaluated.

When a reaction extends to include several reactors, the composite reaction should remain atomic. The following locking conventions ensure this property. A reactor locks when it agrees to react to an update bundle and remains locked for the duration of the reaction until either a quiescent response state is found and committed or a conflict causes the reactor to roll back to its pre-state. When a reactor is locked, it denies any

[1] When a tuple t is present in a relation r, we say that $r\ t$ is a fact.

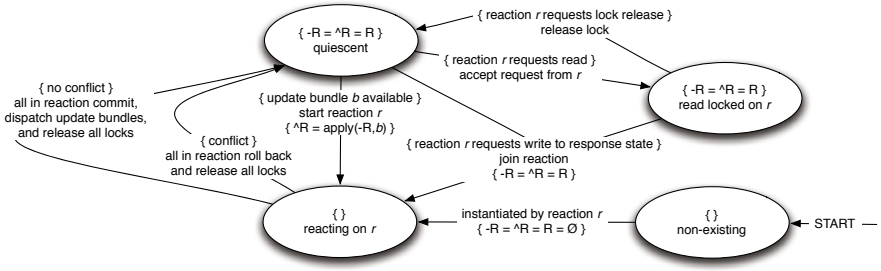

Fig. 10. Reactor state machine. Transitions are written {precondition}activity{postcondition}. States are written {invariant}state.

interaction (read-access, write-access, and beginning to react to other update bundles) with reactors that are not part of the same reaction. Refer to Fig. 10 for an overview.

Intuitively, *reaction* scope extends to include all reactors owning or (non-passively) reading relations being *written*, while *lock* dynamically extends to include all reactors owning relations being *read* (as well as all reactors in the reaction scope). When a remote reactor is read, an exclusive lock on the reactor is acquired and held for the entire duration of the reaction. Should the same remote reactor need to be written later in the same reaction, this defensive locking strategy guarantees that references to the remote reactor's pre-state and response state will in fact refer to two consecutive states of that reactor because it has not been free to serve any other reactions in the meantime. The problem of deadlock naturally arises in this setting; a reactor implementation would require that standard deadlock detection, avoidance, or recovery techniques (e.g., optimistic concurrency control) be used.

7 Advanced Semantic Issues

The rule evaluation model for reactors extends standard datalog with head clause negation (to express deletion) and reactor references. In this section, we review key aspects of standard datalog semantics and define our extensions. We evaluate the rules by forward derivation (also called bottom-up derivation). This strategy applies all rules on the facts to produce all possible consequences—the appropriate approach for our purpose of creating a completely defined new state.

7.1 Head Clause Negation: Semantics Via Translation

We handle negation in head clauses by transforming reactor rules to normal datalog rules. First, we treat each of the four states of a given relation as distinct relations from the point of standard datalog semantics. The goal of reactor rule evaluation is to determine a unique, minimal solution for the response and future values of local and remote relations. Let r_i represent the response or future value of a local relation we wish to compute (we will consider reactor references shortly). Let $\mathtt{r}_1^P$, ... $\mathtt{r}_n^P$ denote the contents of the persistent relations immediately prior to a reaction; if the reactor has just been created the relations are empty by default. Let $\mathtt{\hat{}r}_i^{\Delta^+}$, $\mathtt{\hat{}r}_i^{\Delta^-}$ be the addition

and the deletion sets of the update bundle applied to the reactor. The basic idea for determining the solution to r_i is as follows: (1) introduce a pair of auxiliary relations ($\underline{\mathtt{r}}_i^{\Delta^+}$,$\underline{\mathtt{r}}_i^{\Delta^-}$) which contains the sets of tuples that will be added to and deleted from r_i; (2) eliminate negation in head clauses by transforming the program to a normal datalog program containing references to $\underline{\mathtt{r}}_i^{\Delta^+}$ and $\underline{\mathtt{r}}_i^{\Delta^-}$; (3) evaluate the transformed program using standard datalog semantics; (4) for r_i a response state overwrite $\mathtt{r}_i^P$ to include $\underline{\mathtt{r}}_i^{\Delta^+}$ and exclude $\underline{\mathtt{r}}_i^{\Delta^-}$; for r_i a future state copy ($\underline{\mathtt{r}}_i^{\Delta^+}$,$\underline{\mathtt{r}}_i^{\Delta^-}$) into ($\mathtt{\hat{}r}_i^{\Delta^+}$,$\mathtt{\hat{}r}_i^{\Delta^-}$).

We now describe our program transformation in detail. Given a reactor C with persistent relations r_i and ephemeral relations t_i, let us redefine $\mathtt{\hat{}r}_i^{\Delta^+}$, $\mathtt{\hat{}r}_i^{\Delta^-}$, $\mathtt{\hat{}t}_i^{\Delta^+}$ and $\mathtt{\hat{}t}_i^{\Delta^-}$ to be the addition and the deletion sets of the update bundle applied to reactor C. We can assume that $\mathtt{\hat{}r}_i^{\Delta^+} \cap \mathtt{\hat{}r}_i^{\Delta^-} = \emptyset$ and $\mathtt{\hat{}t}_i^{\Delta^+} \cap \mathtt{\hat{}t}_i^{\Delta^-} = \emptyset$ because this property is checked by the originating reactor before creating new update bundles. Let $\mathtt{-r}_i$ denote the pre-, $\mathtt{\hat{}r}_i$ the stimulus, r_i the response, and $\mathtt{r\hat{}}_i$ the future state of the reaction.

Rewrite rules. Fig. 11 shows how our rewriting technique transforms a program with negation in the head clauses to a program without them. Let us redefine $\underline{\mathtt{r}}_i^{\Delta^+}$, $\underline{\mathtt{r}}_i^{\Delta^-}$, $\underline{\mathtt{t}}_i^{\Delta^+}$ and $\underline{\mathtt{t}}_i^{\Delta^-}$ the sets of additions/deletions to the response state of the persistent and ephemeral relations, correspondingly.

Rewrite:	r_i <- *body*	as: $\underline{\mathtt{r}}_i^{\Delta^+}$ <- *body*	(I)
Rewrite:	not r_i <- *body*	as: $\underline{\mathtt{r}}_i^{\Delta^-}$ <- $\underline{\mathtt{r}}_i$, *body*	(II)
Rewrite:	*head* <- r_i, *body*	as: *head* <- $\underline{\mathtt{r}}_i$, not $\underline{\mathtt{r}}_i^{\Delta^-}$, *body*	(III)
Rewrite:	*head* <- not r_i, *body*	as: *head* <- $\underline{\mathtt{r}}_i^{\Delta^-}$, *body* *head* <- not $\underline{\mathtt{r}}_i$, *body*	(IV)
Rewrite:	$\mathtt{r\hat{}}_i$ <- *body*	as: $\underline{\mathtt{r\hat{}}_i}^{\Delta^+}$ <- *body*	(VII)
Rewrite:	not$\mathtt{r\hat{}}_i$ <- *body*	as: $\underline{\mathtt{r\hat{}}_i}^{\Delta^-}$ <- *body*	(VIII)
Add:	r_i <- $\mathtt{\hat{}r}_i$		(V)
Add:	r_i <- $\underline{\mathtt{r}}_i^{\Delta^+}$		(VI)

Fig. 11. Rewrite rules defining the semantics of reactor rule evaluation in terms of normal datalog

Rewrite rule (I) computes the set of tuples to be added to r_i as the set of tuples that the body clauses resolve to. Rule (II) computes the deletion set very similarly; the only difference is adding a body clause which makes sure that a tuple gets deleted from a relation only if it was already there. The extra clause ensures that this rewriting rule does not introduce domain dependence—see further in this section for details. Rule (VI) adds the new addition sets to the response state as soon as they are computed; this ensures that the most current tuple additions are visible and propagating to the rest of the reactor rules. We would like to similarly account for the deletion sets but to reflect that in the response state we have to express it as negation in the rule head—exactly what the program transformation technique is trying to eliminate. Therefore the deletion sets must be accounted for and propagated via the reactor rules. As a result, rule (III) restricts the matching for the tuples in r_i to the ones that are not in the deletion set; conversely, rule (IV) allows matching on tuples in the deletion set. The rest of the

rules are trivial. For t_i rules (I) to (VI) apply unchanged; rules (VII) and (VIII) do not apply because ephemeral relations do not have a future state.

State updates. At the beginning of a reaction the following assignments take place:

$$\texttt{-r}_i ::= \texttt{r}_i^P, \quad \texttt{\^{}r}_i ::= \texttt{\^{}r}_i^{\Delta^+} \cup \texttt{-r}_i \setminus \texttt{\^{}r}_i^{\Delta^-}, \quad \texttt{\^{}t}_i ::= \texttt{\^{}t}_i^{\Delta^+}$$

where '::=' should be read as relation overwriting. Intuitively the first assignment overwrites the pre-state of the current reaction's persistent relations with the contents of the relations prior to the reaction. The next assignments then apply the corresponding update bundles to the pre-state to obtain the stimulus state. Note that the deletion set of the update bundle $\texttt{\^{}t}_i^{\Delta^-}$ for ephemeral relations has no effect.

All assignments are done outside datalog and have the effect of keeping a snapshot copy of the pre-state and the stimulus state in case the rules need to read them or the reaction rolls back. After the assignments take effect we can apply standard datalog techniques to evaluate the program rules up to a fixpoint. If at any point during the evaluation either $\underline{\texttt{r}}_i^{\Delta^+} \cap \underline{\texttt{r}}_i^{\Delta^-} \neq \emptyset$, $\underline{\texttt{r\^{}}}_i^{\Delta^+} \cap \underline{\texttt{r\^{}}}_i^{\Delta^-} \neq \emptyset$, $\underline{\texttt{t}}_i^{\Delta^+} \cap \underline{\texttt{t}}_i^{\Delta^-} \neq \emptyset$, or $\underline{\texttt{t\^{}}}_i^{\Delta^+} \cap \underline{\texttt{t\^{}}}_i^{\Delta^-} \neq \emptyset$ the evaluation stops and the reaction rolls back. If we reach the fixpoint (without either of the checks failing) we update $\texttt{r}_i^P$ to take into account the deletion sets: $\texttt{r}_i^P ::= r_i \setminus \underline{\texttt{r}}_i^{\Delta^-}$. Before quiescing, the reaction forms the update bundles ($\underline{\texttt{r\^{}}}_i^{\Delta^+}$,$\underline{\texttt{r\^{}}}_i^{\Delta^-}$) and ($\underline{\texttt{t\^{}}}_i^{\Delta^+}$,$\underline{\texttt{t\^{}}}_i^{\Delta^-}$) for other reactors and for itself, if applicable.

7.2 Semantics of Normal Datalog Programs: Stratification and Safety

As a result of applying our rewrite technique, we are left with a normal datalog program containing negation in body clauses only. Since it is possible to use negation to encode logical paradoxes, we wish to apply a syntactic constraint to rules that will ensure that a unique solution to collection of rules exists, without unduly affecting expressiveness. We adopt the *stratification* semantics for normal programs with negation [15]. The main idea behind stratification is to partition the program along negation such that for any relation we fully compute its content before applying the negation operator on it. For example, a program consisting of the rules `q(x) <- p(x,y), not q(y)` and `p(1,2) <-` is not stratified because it contains recursion through negation.

Another desirable property of a datalog program is *domain independence*: a solution should depend only on the known facts and not on the universal set of all facts. We would also like to ensure *finiteness*, or at least *weak finiteness*: that the evaluation of a rule from a finite set of facts yields a finite set of results, but infinite results caused by infinite recursion cannot be ruled out. To ensure domain independence and weak finiteness, we adopt the syntactic *safety* condition of Topor [14], which supports arithmetic expressions and negation. Briefly, a rule is safe if all of its variables are *limited*. A variable is limited if it occurs in a non-negated clause in the body, if is occurs in a negated clause in the body and is not used elsewhere, or if it occurs in an expression where a unique value of the variable can be computed given all the limited variables of the expression. A program is safe if all of its rules are safe.

7.3 Remote Reactor References

In a naive approach, every time a set of synchronously executing reactors C_i tries to extrude its scope to a new reactor C which is executing, C_i (1) waits for C to quiesce and accept the scope extrusion request, then (2) rolls back and restarts execution within the new scope expanded to include C.

A less naive approach will first statically compute the transitive closure of all the synchronously executing reactors based on the type information. The goal is to coordinate the order of rule evaluation with the order of locking—see Section 6. The algorithm will therefore compute the global stratification for the statically computed reactor closure; any time there is a choice of rules to evaluate next, the algorithm will choose the one that is consistent with the global stratification. This will make sure that positive information in relations is fully computed before evaluating its negation.

The program obtained by putting together the rules of all reactors C_k in a reaction will contain remote references to relations in C_k. To make all remote references local we define the following transformation. For every relation `r` in a reactor, the reactor defines a shadow copy `r~` and an implicit rule of the form `r~(self,x) <- r(x).` Every remote reference `c.r(x)` to relation `r` can then be transformed into a local access to `r~(c,x)`. At this point the statically computed set of rules only contains local references and it is treated as a single program to which we can apply the transformation in Fig. 11.

7.4 Reactor Instantiation Details

The semantics of reactor instantiation must be defined with some care to ensure that the number of reactors instantiated in a reaction is unambiguous. Consider a rule whose head has the form $\texttt{r}(x_1, ..., x_i, \texttt{new M}, x_{i+1}, ..., x_n)$, where `M` is a reactor type name and $x_k, k \in [1...n]$ are variables bound in the rule body. Then a tuple $\langle v_1, ..., v_i, \alpha, v_{i+1}, ..., v_n \rangle$ is a satisfying solution for the rule if and only if (1) α is a reference to a reactor of type `M`, (2) α is globally unique, (3) α did not exist prior to the current reaction, (4) $\langle u_1, ..., u_j, \alpha, u_{j+1}, ..., u_m \rangle$ does not satisfy any rule with a head of the form $\texttt{s}(y_1, ..., y_j, \texttt{new M}, y_{j+1}, ..., y_m)$, and (5) $\langle u_1, ..., u_i, \alpha, u_{i+1}, ..., u_n \rangle$ does not satisfy any rule with a head of the form $\texttt{r}(x_1, ..., x_i, \texttt{new M}, x_{i+1}, ..., x_n)$, unless $u_k = v_k$ for all $k \in [1...n]$. The basic idea behind this definition is that in each reaction, for each instance of `new` in a rule head, a new, globally unique reactor reference is generated *for each satisfying combination of values bound to variables in the rule*. The generalization of this semantics to rules containing multiple instances of `new` is straightforward.

Consider the rules `[1] r(new Foo, i) <- t(i)`, `[2] s(y) <- r(y, _)`, and `[3] s(new Foo) <- .` If relation `t` initially contains three tuples, then following the instantiation semantics above, rule `[1]` causes three new reactors to be instantiated. Similarly, rule `[3]` will generate one new reactor, distinct from those generated by rule `[1]`. If relation `s` is initially empty, it will contain four distinct reactor references at the end of the reaction: three from the reactors instantiated by rule `[1]`, and one from the reactor instantiated by rule `[3]`. If we were to duplicate any of rules `[1]`-`[3]`, the result of the reaction would remain the same, since the instantiation behavior of rules is dependent on their semantics, not their syntactic form.

Note that although the instantiation semantics distinguishes between `new` expressions and variables in rule heads, it does not distinguish reference-bound *variables* from other variables. Consider, e.g., the rules `[4] t(new Bar, i) <- r(i)` and `[5] s(new Bar, b) <- t(b, j)`. Rule `[5]` requires that there exist a unique, newly generated reference for each `b` satisfying `t(b, j)`. If `r` initially contains two tuples, then rules `[4]` and `[5]` each instantiate two new reactors.

It will be convenient to assume that reactor references generated by the same reactor are totally ordered, which allows them to be used as a source of ordered—but otherwise uninterpreted—keys. The `Nonce` reactors used in Fig. 7 serve this purpose.

8 Extended Example: Three-Tier Web Application

Fig. 12 depicts an extended example which follows the structure of a conventional three-tier web application for catalog ordering (note that this example makes extensive use of the notational abbreviations of Fig. 5). Unlike the example in Fig. 9, which combined "client" and "server" functionality in a single component, the example in Fig. 12 explicitly models a database (`DB`), web server (`WebServer`), and browser (`Browser`) as distinct components. The browser and web server communicate entirely asynchronously, while the web server and database communicate synchronously (i.e., transactionally). "Page content" in the browser is modeled by the `CompositeWidget` reactor type, which is instantiated with various primitive widget reactors (`OutputWidget` and `ButtonWidget` from Fig. 9 and a new `FormWidget`), depending on the type of page being displayed. Instances of `CompositeWidget` perform local (i.e., "browser-side") computation which performs basic form validation. Such functionality could easily be replaced with more elaborate browser-based widgets, e.g., with AJAX-style asynchronous behavior. One limitation of our current model is that all reactor references must be strongly typed, which makes it difficult to model a web browser reactor that can render arbitrary pages, also represented as reactors. In the future, we will consider more flexible type systems as well as weaker stratification requirements, which would allow a completely generic browser to be defined.

9 Related and Future Work

Related Work. Fundamentally, reactors are "reactive systems" [8], combining and extending features from several, largely unrelated areas of research: synchronous languages, datalog [15], and the actor model [1].

Esterel [2], Lustre [4], Signal [7], and Argos [11] are prominent synchronous languages. In synchronous languages, the term *causality* refers to dependencies, and all have restrictions on cyclic dependencies. Esterel only admits a program if all signals can be inferred to be either present or absent (as opposed to unknown); this is referred to as *constructiveness*. Esterel adopts a strict interleaving semantics, i.e. it assumes that reactions cannot overlap temporally. In Esterel signals are broadcast instantaneously so that all receptors of the signal will see it in the same instant and the signal will only exist in that reaction. The reactor model, on the other hand, supports both synchronous and asynchronous broadcasts (readers can react when a relation is changed) as well as

```
def DB = {
  (* trivial database: single value
     containing item inventory *)
  public inv: (int).
}

def WebServer = {
  (* reference to the database *)
  rDB: (ref DB)
  (* server accepts two request types
     from browsers: session initiation
     and form submission *)
  public ephemeral newSession:
    (ref Browser).
  public ephemeral formSubmit:
    (ref Browser, int).
  (* temp to hold new page *)
  ephemeral newPage:
    (ref CompositeWidget).

  (* generate page on every reaction *)
  newPage(new CompositeWidget) <- .
  (* each has link back to server *)
  c.rServer(self) <- newPage(c).

  (* newSession creates three primitive
     widgets for the new page *)
  newSession(_), newPage(c): {
    c.outWidget(new OutputWidget),
    c.formWidget(new FormWidget),
    c.buttonWidget(new ButtonWidget)
      <- .
    (* init primitive widget data,
       in particular, copy current
       inventory from db *)
    o.label("Available: "),
    o.val(toString(q)) <-
      c.outWidget(o), rDB(d), d.inv(q).
    f.label("Quantity to order: "),
    f.val("1") <- c.formWidget(f).
    b.label("Submit") <-
      c.buttonWidget(b).
  }

  (* formSubmit checks whether
     requested qty. is avail.; returns
     appropriate responses *)
  formSubmit(br, qr), newPage(c): {
    c.outWidget(newOutputWidget) <- .
    ephemeral reqOK() <-
      qr >= q, rDB(d), d.-inv(q).
    o.label("Success!") <-
      reqOK(), c.outWidget(o).
    d.inv(q') := reqOK(), rDB(d),
      d.-inv(q), q' = q - qr.
    o.label("Sorry!") <-
      not reqOK(), c.outWidget(o).
  }

  (* submit new page to browser *)
  br.showPage^(c) <- .
}

def Browser = {
  (* request to display new page *)
  public ephemeral showPage:
    (ref CompositeWidget).
  (* current page visible in browser *)
  thePage: (ref CompositeWidget).

  (* request to show new page updates
     current page; link to browser *)
  thePage(c) := showPage(c).
  c.rBrowser(self) <- showPage(c).
}

def CompositeWidget = {
  (* refs to server and browser *)
  rServer: (ref WebServer).
  rBrowser: (ref Browser).
  (* widgets on page; the form and
     button widgets are empty (not
     used) on the response page *)
  outWidget: (ref OutputWidget).
  formWidget: (ref FormWidget).
  buttonWidget: (ref ButtonWidget).

  (* perform local validation: ensure
     qty. requested less than inv. *)
  ephemeral validateOK: ()
  validateOK() <-
    buttonWidget(b), b.pressed(),
    formWidget(f), f.val(qty),
    outWidget(o), o.val(inv),
    toInt(qty) <= toInt(inv).
  (* validation OK: submit to server *)
  rServer.formSubmit^(br, toInt(qty))
    <- validateOK(), rBrowser(br),
       formWidget(f), f.val(qty).
  (* validation fails: just reset qty.
     to 1; do not submit *)
  f.val("1") <- not validateOK().
}

def FormWidget = {
  public label: (string).
  public val: (string).
}
```

Fig. 12. Mini three-tier web application

synchronous and asynchronous point-to-point communication (by writing directly into a public relation of the receiver).

Lustre and Signal also limit cyclic dependencies, but add *sampling* in the form of the construct `x = Exp when BExp` meaning that `Exp` should be evaluated only

when `BExp` is true. This facility provides a sophisticated way of reading values from preceding reactions other than the immediately previous one. In the reactor model, such predicates can be expressed directly as `x(Exp) <- BExp` where `x` should be a singleton relation. Argos is based on State Charts and hierarchical automata and distinguishes itself from other synchronous languages by being graphical.

Generally speaking, the group of synchronous languages does not allow cycles in the data flow graph – only pre-state to response-state connections are permitted when referring to the same variable. In the reactor model, stratification provides a more refined classification that widely allows recursion while ruling out cases where the fixed point could be ambiguous (of course, programs may still loop infinitely). Reactors provide several features not found in synchronous languages, namely asynchrony, generativity, and distributed transactions. We are not familiar with any other language that combines these features.

Active databases [13] commonly express triggers of the form *Event–Condition–Action* (ECA), where the action is carried out if on receipt of a matching event the condition holds true. This can be expressed as `action <- event, condition` in the reactor model. The reactor model eliminates the distinction between conditions and events, and adds support for distribution, process generation, and synchronous composition.

Transaction Datalog [3] introduces transactions and database updates to datalog. In Transaction Datalog, inserts and deletes are special atoms in rule bodies, and backward derivation rather than forward derivation is used. To achieve concurrency in transactions a concurrent conjunction operator, `|`, is added. In the reactor model, all rules execute concurrently within the same reaction (subject to stratification) by default, and thus *sequentiality*, rather than concurrency, must be programmed explicitly when needed.

Future Work. While this paper has not focused on implementation, there are two broad areas that are amenable to optimization: query incrementalization, and efficient implementation of synchronous composite reactions through low-overhead concurrency control. The former has already been studied in the datalog community (e.g., [5]), and we intend to adapt those results appropriately to our setting. In the case of synchronous reactions, recent results on efficient implementation of software transactions (e.g., [9]) are likely to be relevant.

Other issues we plan to investigate include: (1) contract/interface type systems; (2) various abstraction facilities, such as reactor and rule parametricity and high-order rules, that read, write, and deploy other rules; (3) more sophisticated access control mechanisms; (4) function symbols (functors); (5) reactor garbage collection; (6) a truly distributed implementation; (7) support for long-running (rather than atomic) transactions.

Acknowledgments. The authors gratefully acknowledge the contributions of Rafah Hosn, Bruce Lucas, James Rumbaugh, Mark Wegman, and Charles Wiecha to the development of the ideas embodied in this work.

References

1. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7(1), 1–69 (January 1997)
2. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
3. Bonner, A.J.: Workflow, transactions and datalog. In: PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, pp. 294–305. ACM Press, New York (1999)
4. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, pp. 178–188. ACM Press, New York (1987)
5. Dong, G., Topor, R.W.: Incremental evaluation of datalog queries. In: Hull, R., Biskup, J. (eds.) ICDT 1992. LNCS, vol. 646, pp. 282–296. Springer, Heidelberg (1992)
6. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
7. Gautier, T., Guernic, P.L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Proceedings of the Functional Programming Languages and Computer Architecture, London, UK, pp. 257–277. Springer, Heidelberg (1987)
8. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and models of concurrent systems. NATO ASI Series F: Computer And Systems Sciences, vol. 13, pp. 477–498. Springer, Heidelberg (1989)
9. Harris, T., Marlow, S., Jones, S.P., Herlihy, M.: Composable memory transactions. In: ACM Conf. on Principles and Practice of Parallel Programming, Chicago, pp. 48–60 (June 2005)
10. Kiczales, G.: Aspect-oriented programming. ACM Computing Surveys 28, 4es, 154 (1996)
11. Maraninchi, F., Rémond, Y.: Argos: an automaton-based synchronous language. Computer Languages 27, 61–92 (2001)
12. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I-II. Information and Computation 100(1), 1–77 (1992)
13. Paton, N.W., Díaz, O.: Active database systems. ACM Comput. Surv. 31(1), 63–103 (1999)
14. Topor, R.: Safe database queries with arithmetic relations. In: Proceedings of the 14th Australian Computer Science Conference (1991)
15. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. 1. Computer Science Press, ch. 3 (1988)
16. Wang, X.: Negation in Logic and Deductive Databases. PhD thesis, University of Leeds (1999)