Theoretical Computer Science 410 (2009) 168–201

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications

John Field [a,*], Maria-Cristina Marinescu [a], Christian Stefansen [b]

[a] IBM Research, T.J.Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532, United States

[b] DIKU, University of Copenhagen, Universitetsparken 1, København Ø, Denmark

ARTICLE INFO

Keywords:
Reactors
Datalog
Actors
Synchronous programming
Asynchronous programming
Synchronization
Distributed programming

ABSTRACT

Our aim is to define the kernel of a simple and uniform programming model – the *reactor* model – which can serve as a foundation for building and evolving internet-scale programs. Such programs are characterized by collections of loosely-coupled distributed components that are assembled on the fly to produce a composite application. 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.

© 2008 Elsevier B.V. All rights reserved.

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 and user interaction 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.

The reactor model is a synthesis and extension of key ideas from three linguistic foundations: synchronous languages [1–3], Datalog [4], and the actor model [5]. From Datalog, we get an expressive, declarative, and readily composable language for

* Corresponding author. Tel.: +1 914 784 6650.
E-mail address: jfield@watson.ibm.com (J. Field).

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.09.052

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

Fig. 1. Reactor syntax.

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 [4]. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's pre-reaction state (*pre-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 [6], 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 [7]), 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. We believe that Internet components can be developed more productively and composed more readily when these attributes are provided in a single programming model.

This paper is an updated and extended version of [8]. This version fills in many semantic details absent from the earlier version, corrects errors, and adds additional examples.

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. 1.

Consider the declaration for `OrderEntryA` in Fig. 2. `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 in Section 3.2.

The state of a reactor is embodied in a fixed collection of *persistent relations*. Relations are sets of $(\tau_1, \ldots, \tau_n)$ tuples, where each τ_i is one of the types `int`, `string`, or `ref` *reactor-type-name*. The primitive types have the usual meanings. *Reactor references*, of the form `ref` *reactor-type-name*, are described in Section 3.2. 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 are written and read in the same manner as persistent relations, but they are re-initialized as empty with every reaction.

```
def OrderEntryA = {
  // (id, itemid, qty)
  public orders: (int, int, int).
  // (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 the
  // same as a previously logged id,
  // but which is not a duplicate
  not orders(id, itemid, qty) <-
    ^orders(id, itemid, qty),
    not -log(id, itemid, qty),
    -log(id, _, _).
}
```

Fig. 2. Order entry: Variations.

The state of `OrderEntryA` 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`.

A reaction begins when a reactor receives an *update bundle* from an external source. An update bundle is a partial 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. For any update bundle (Δ^+, Δ^-), we require that $\Delta^+ \cap \Delta^- = \emptyset$ and $\Delta^+ \cup \Delta^- \neq \emptyset$. 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 an 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 the contents of 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 reactors will have one or more rules which compute a response state distinct from the reactor's stimulus state (Section 2.1).

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.

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^`).

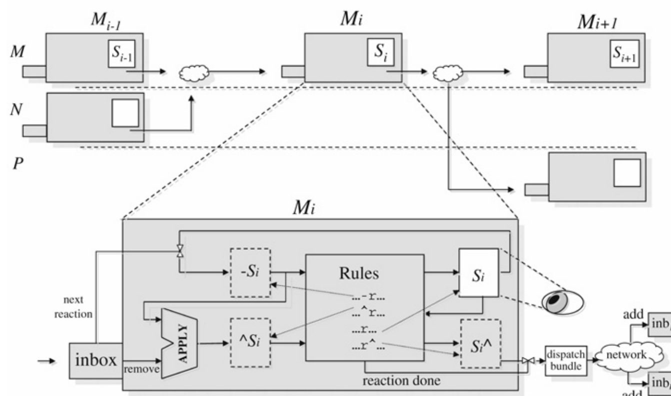

Fig. 3. Reaction schematic.

2.1. Rule valuation

Reactor rules are written in the style of Datalog [4,9]; their syntax is given in Fig. 1. 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`. Each clause in a rule may be regarded as a predicate in the logical sense, hence a logical reading of `OrderEntryA`'s rule would be "for all `id`, `itemid`, `qty`, if `orders(id, itemid, qty)` then `log(id, itemid, qty)`."

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. 2. 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., [10]).

Reactor definitions `OrderEntryB` and `OrderEntryC` of Fig. 2 add additional rules that 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.

```
def OrderEntryD = {
  // (itemid, qty)
  public ephemeral write req: (int, int).
  // (key, itemid, qty)
  public log: (ref Nonce, int, int)
  // pending orders keyed to log entries
  public pending: (ref Nonce).
  // new key for this reaction
  ephemeral newKey: (ref Nonce).

  newKey(new) <- .              // 1
  log(k, item, qty) <-             // 2
      req(itemid, qty), newKey(k).
  pending(k) <- newKey(k).         // 3
}

def Nonce = {}
```

Fig. 4. Service-style order entry.

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

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

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

Fig. 5. Cell.

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 easier to update the functionality of a reactor in an "aspect-like" fashion [11] 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. 2, where rules can be mixed and matched liberally to yield updated functionality.

Consider now the very different formulation of an order entry reactor depicted in Fig. 4, `OrderEntryD`. In reactor type `OrderEntryA`, the set of all orders active in the system is publicly readable and writable by external clients. In the alternative `OrderEntryD` formulation above, the ephemeral relation `req` functions as a request "channel" or "port": incoming collections of order entries in an update bundle are processed and cleared (deleted) immediately after the reaction, since `req` is ephemeral. Relation `req` has a `public write` access annotation, hence it is not externally readable.

Proponents of the "representational state transfer" ("REST") style of component interaction [7] embodied by `OrderEntry` argue that it makes evolution of web applications easier by exposing state directly, rather than encapsulating it through access channels/ports/methods in a "service-oriented" style [12]. A notable feature of the reactor model is that both styles are easily accommodated.

The rules of `OrderEntryD` are straightforward, except for rule (1), which creates a new key. The expression `new` generates a new and unique instance of a *nonce*, a trivial reactor whose only function is to serve as a generator of globally-unique values. It is convenient to use such values as keys. While `Nonce`s 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 3.3. Rule (1) is an *unconditional rule*, and is an instance of the shorthand notation depicted in Fig. 6(b). It is important to note that rule (1) only generates one instance of a `Nonce` per reaction; a rule head clause containing `new` is satisfied once unique values have been generated for each instance of `new` per vector of rule variables that are instantiated by the rule.

2.2. Initialization, constants, and reaction failure

Consider the following rules:

```
r(x) <- s(x).
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. 5. 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 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) defines `live` to be 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 head clause (left-hand side) be satisfied, i.e., that

	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 nonterminal EXP in Fig. 1; 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() <- .

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. 6. Notational conventions.

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. 6(d) depicts a notational convention that will allow us to use live to define rules that represent assertions.

Clients of instances of reactor type Cell are required to ensure that they maintain its singleton constraint, e.g., by deleting the current value of the cell before adding a different value. However, if we wished, we could augment the declaration of Cell to make it easier for clients that wish to update its value to avoid having to delete the previous value by adding the following rule:

```
not val(x) <- -val(x), ^val(x'), x <> x'.
```

This rule is interesting because it refers to all three reactor states we have discussed thus far: pre-state, stimulus state, and response state. The body of the rule checks to see whether the stimulus value of val contains an item different from the pre-state. If so, the offending pre-state item is deleted from the response value of val. Note, however, that it is still possible for the singleton constraint to fail if a client attempts to insert more than one distinct value in a single update bundle. Fig. 6(c) depicts a notational convention we will use in the sequel whenever we wish to "assign" values to singleton relations such as val. The example in Fig. 5 illustrates how the declarative nature of reactor rules makes it straightforward to "progressively refine" existing functionality by adding new features in a non-intrusive way.

3. Reactor composition

In the following sections we explain how reactors interact asynchronously and synchronously; we refer to this as *reactor composition*.

3.1. 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.

```
def Fibonacci = {
  // complete series thus far: each
  // elt. has form (index, 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
  notLargest(n) <- series(n, _),       // 1
    series(n', _), n' > n.

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

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

Fig. 7. Self-reacting Fibonacci.

Consider the reactor definition `Fibonacci` in Fig. 7, 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. 6. 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 only existential properties directly. To query universal properties, we typically require auxiliary relations. In `Fibonacci`, we use the ephemeral relation `notLargest` to contain all the indices of elements of `series` which are less than the maximum index. The body clause `not notLargest(n)` in rule (2) has the effect of binding `n` to the largest ordinal index currently contained in the series.

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 effectively defines the contents of an update bundle which is processed in a subsequent reaction, *after* the current reaction ends. Hence, the head of rule (2) defines an update bundle containing a single pair encoding the next value of the series; this pair will be inserted into `series` at the beginning of the next 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), which uses the notation defined in Fig. 6(d), causes the reaction to fail and roll back if both the pre-state and the stimulus values of `run` are false. By using this encoding, `run` serves as a sort of switch which can be used to turn the self-reaction process on and off. 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* containing a multiset of pending update bundles. When a reaction is complete, the reactor checks for a new update bundle in the inbox. If it exists, the reactor removes 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. Fig. 3 illustrates this process. We make no assumptions about the order in which inbox items are processed, except that they must be processed fairly.

While `Fibonacci` is designed such that update bundles can only update one relation at a time, update bundles can in general contain updates to more than one relation. Consider, e.g., a client that wishes to update ordered trees (e.g., XML trees) maintained on a server. Ordered trees can be maintained using two relations on nodes: a parent–child relation, and a next sibling relation. In this case, it is natural for an update to affect both relations.

3.2. *Reactor references and multi-reactor asynchrony*

Up to this point, our examples have only considered a single reactor type. Consider now the definitions for reactors `Sample`, `Sensor`, and `Nonce` depicted in Fig. 8. Reactor types `Sample` and `Sensor` 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 by a `Sensor` reactor) whenever a client of `Sample` updates `pulse`. The expression `s.req^(self)` in Rule (1) contains an *indirect reference* to (the future value of) 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.

```
def Sample = {
  // 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: ().
  // holds sensor response
  public write ephemeral response: (int).

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

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

```
def Sensor = {
  // set when sample is to be collected;
  // value is ref.to sample reactor
  public write ephemeral req: (ref Sample).
  // 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. 8. Asynchronous query/response.

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 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` in a head clause creates a new instance of the appropriate reactor type. The appropriate reactor type is inferred; e.g., if a relation a is declared as `a: (int, ref Acct, ref Acct)` then new instances created by the head clause `a(5, new, fromAcct)` will have the type `Acct`. Instantiation expressions may only appear in non-negated head clauses. Rule (2) of `Sample` creates instances of the trivial reactor `Nonce`. `Sample` uses nonces to distinguish multiple instances of the same sensor value.

In order to instantiate and connect `Sample` and `Sensor` instances together, another reactor must contain rules of the form

```
theSampler(new Sample) <- .
s.rSensor(new Sensor) <- theSampler(s).
```

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`.

3.3. *Synchronous reactor composition*

While the asynchronous form of process composition depicted in Fig. 8 is similar to that used in many process calculi and message-based distributed programming models, Fig. 9 depicts an example of *synchronous* reactor composition that is somewhat more unusual. In this example, a `MiniBank` reactor receives requests to transfer money between two `Acct` reactors. Such requests are encoded by updates to the ephemeral `transferReq` relation. As with the example in Fig. 8, we use references to wire the reactors together. However, unlike the previous example, the remote references in Fig. 9 refer to *response* values of relations, not future values. This means that a reaction initiated at a `MiniBank` reactor will *extrude* to include both of the `Acct` reactors (bound to variables `to` and `from`, respectively). This results in a composite, synchronous, atomic reaction involving *three* reactor instances. Scope extrusion is an inherently dynamic process, similar to a distributed transaction—see Section 9 for details. `MiniBank` uses the notation of Fig. 6(c) to define "assignments" to the singleton `balance` relations.

Note that the rules in `Acct` encode constraints on the allowable values of `balance`. In a composite reaction, a conflict (which is here manifested by a constraint failure) in any of the involved reactors causes the composite reaction to fail. When a composite reaction fails, 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. 9, reactor instance `MiniBank`.

Reactors involved in a composite reaction may separately define future values for relations of the same reactor instance. In such cases, a *single* update bundle, combining the composite future value updates for all of the involved reactors, is dispatched at the end of the composite reaction to each target reactor. In this sense, from the point of view of an external observer, a composite reaction has the same atomicity properties as a reaction involving a single reactor.

The example in Fig. 10 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 user interface components representing the button and output field generated

```
def Acct = {
  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 amount, to account,
  //  from account)
  public write ephemeral transferReq:
    (int, ref Acct, ref Acct).

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

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

Fig. 9. Classic transaction.

for each item in relation db. ButtonWidget's buttonListener relation contains a reference (in general, there could be more than one) to the "parent" DataDisplay reactor; rule (1) has the effect of notifying the parent whenever the button is pushed.

Rule (9) synchronizes the value of the output field to the value of the quantity currently maintained in the database. Rule (10) "wires" together corresponding button and database items such that when a button is pressed, the corresponding data item is decremented.

Ephemeral relations currDbItems and oldDbItems along with rules (4) and (5) compute the set of itemids present before and after the reaction. Using this information, rules (6) and (7) together create new widgets whenever a new item is added to db, while rule (8) deletes a widget whose corresponding item has been deleted from the database.

Note that the auxiliary ephemeral relations currDbItems and oldDbItems are not just included for clarity, they are necessary to correctly compute the set of old and new items. Consider, for example, a rule of the form

```
...<- db(i, _, _), not -db(i, _, _)
```

One might initially expect i to be bound only to newly-added items in the database. However, i will actually be bound to *every* itemid in the database, since for any itemid i in the database, there always exists *some* widgets b and o such that $\langle i, b, o\rangle$ is not in the pre-state of db. In other words, as written, the second body clause will always succeed, regardless of the value of i. This phenomenon can occur whenever unbound variables (such as the wildcards used above) are used in a negative body clause.

The rules which wire the database, the output field, and the button together constitute a dependency loop which would be tricky to manage if written in a traditional language. However, the normal Datalog evaluation process evaluates such recursive loops without difficulty.

4. Reactor semantics overview

In the next several sections, we formalize the operational semantics of the reactor model. The first two sections contain preliminaries: in Section 5, we review standard concepts from the semantics of Datalog with negation. In Section 6, we define basic notation for reactor concepts we will use in the rest of the paper.

The reactor model takes the point of view that each reactor instance maintains a set of relations which the instance "owns and manages" autonomously. However, as we have observed, a reactor can synchronously read or write the state of a different, *remote* reactor via reactor references. To facilitate formalizing the semantics of synchronous reactor interaction, Section 7 contains a sequence of source transformations which will serve to define a single "virtual address space" of relations. The transformed relations have the property that they may in principle contain tuples from *different* reactor instances of the same type. However, as we will see in Section 9, our operational semantics properly models the fact that each reactor independently manages only the persistent state defined by its own relation instances. To model synchronous multi-reactor reactions, the operational semantics temporarily *copies* the state of remote reactors into the state of the reactor instance which initiated the reaction. The *augmentation* transformation defined in Section 7.3 allows the initiating reactor to manage both its own state and temporarily-copied state of remote reactors in a uniform way.

Section 8 defines additional source transformations on the program resulting from the transformation of Section 7. These additional transformations eliminate head negation by defining auxiliary relations which encode addition and deletion information separately.

The transformations of Sections 7 and 8 rewrite a reactor program $\mathcal{T}$ consisting of multiple reactor definitions into a single "normal" (i.e., with body negation) Datalog program that captures the synchronous execution semantics of all of the reactor definitions in $\mathcal{T}$. This approach might seem to imply that every reactor in a distributed system needs to contain the code of all other reactors in the system. In practice, however, a reactor need only be aware of the rules of reactors which

```
def ButtonWidget = {
  // button label
  public label: (string).
  // set to true when button is pressed
  public write ephemeral pressed: ().
  // reference to DataDisplay, which will
  // be notified when button is pressed
  public write buttonListener:
    (ref DataDisplay).

  // notifies DataDisplay when button
  // is pressed, passing a self-reference
  // to indicate which button was pressed
  rD.buttonPressed(self) <-
    buttonListener(rD), pressed().      //
1
}

def OutputWidget = {
  // label (constant)
  public label: (string).
  // current value to be displayed
  public val: (string).
}

def DataDisplay = {
  // (itemid, quantity)
  public db: (int, int).
  // set when button(s) pressed
  public ephemeral write buttonPressed:
    (ref ButtonWidget).
  // (itemid, button widget)
  bWidgets: (int, ref ButtonWidget).
  // (itemid, output widget)
  oWidgets: (int, ref OutputWidget).

  // initialize widget labels
  rO.label("Inventory: ") <-
    oWidgets(_, rO).                  // 2
  rB.label("Click to decr") <-
    bWidgets(_, rB).                  // 3

  // itemids of db entries in prestate
  ephemeral oldDbItems: (int)
  oldDbItems(i) <- -db(i, _).         // 4

  // itemids of db entries in curr. state
  ephemeral currDbItems: (int).
  currDbItems(i) <- db(i, _).         // 5

  // create widgets when new item
  // added to db
  currDbItems(i), not oldDbItems(i): {
    bWidgets(i, new) <- .             // 6
    oWidgets(i, new) <- .             // 7

    // add self to notification list
    // for button widget
    rB.buttonListener(self) <-
      bWidgets(i, rB).                // 8
  }

  // delete widgets when corresp. item
  // deleted from db
  not currDbItems(i), oldDbItems(i): {
    not oWidgets(i, _) <- .           // 9
    not bWidgets(i, _) <- .           // 10
  }

  // output widget values track qty.
  // of corresponding db item
  rO.val(toString(q)) :=
    oWidgets(i, rO), db(i, q).        // 11

  // button decrements qty. of
  // corresponding db item
  db(i, q-1), not db(i, q) <-
    -db(i, q), bWidgets(i, rB),
    buttonPressed(rB).                // 12
}
```

Fig. 10. Data-driven UI.

are accessible via reference types that are elements of relations declared in T, i.e., peer reactors with which it has already agreed to interact.[1]

In Section 9, we complete the definition of the semantic model for reactors using a labeled transition system. This transition system carries out the following functions:

- It orchestrates the creation and processing of *update bundles*, messages that take the form of state updates. Once paired with its recipient reactor, an update bundle can initiate a reaction; dually, reactors may generate update bundles (by defining *future values* of remote relations); the transition semantics takes care of delivering these bundles to their destination.
- It models multiple concurrent, autonomous reactions and asynchronous interactions among such reactions.
- It orchestrates copying of relation values from remote reactors to the initiating reactor. The initiating reactor evaluates all the local and remote rules needed to define the state resulting from the synchronous reaction, and the state transition system ensures that the results of the synchronous reaction are copied back to remote reactors as necessary.
- It defines an *optimistic locking protocol* which ensures that the results of a synchronous composite reaction are computed atomically from the point of view of any external observer, and which allows multiple concurrent reactions to access a common reactor instance, provided that the results of the reactions do not conflict.
- It handles state *rollback* in the event that a reaction yields a *conflict*: an inconsistent state in a predicate is asserted to be both true and false (alternatively, in which a tuple is both inserted and deleted into the same relation).

In Section 10, we show that the notions of stratification and safety defined for normal Datalog programs extend in a straightforward way to reactor programs.

[1] It is possible to further weaken this "mutual knowledge" requirement, see Section 13.

5. Datalog background

In this section, we review basic concepts from logic programming and Datalog [13]. We will occasionally deviate from standard definitions when appropriate for our context.

5.1. Basic definitions

A *term* is either a constant, a variable, or a built-in function applied to constants or variables. An *atom* consists of an n-ary *predicate symbol* and a list of arguments such that each argument is a term. We will consider both *user-defined* predicates (which we will also refer to as *relations*) and *built-in* predicates, such as equality, inequality, and arithmetic comparison. We allow function symbols only in arguments to built-in predicates. A *literal* is an atom or a negated atom. A *clause* is a finite list of literals. A *ground* clause is a clause which contains neither variables nor function symbols. A *unit* clause is a clause containing only one literal. A *fact* is a positive ground unit clause.

A *Herbrand interpretation* (or just "interpretation") is a set of facts. A *Herbrand model* is a Herbrand interpretation that satisfies every formula belonging to a given set of closed formulas (a given set of facts and rules). A *positive* Datalog program contains rules with only positive atoms. For such programs, there exists a least Herbrand model such that any other Herbrand model is a superset of this model. A *normal* Datalog program is a Datalog program in which negation may appear in body clauses, but not in the head. For normal programs, the existence of a least Herbrand model is no longer guaranteed.

5.2. Stratification semantics for normal programs

To ensure that a least Herbrand model exists without unduly affecting expressiveness, we will restrict the set of normal Datalog programs we consider to those that are *stratified* [14–16]. The main idea behind stratification is to partition the program such that for any relation, we fully compute its contents before applying the negation operator. For example, a program consisting of the following rules is not stratified because it contains recursion through negation:

```
q(x) <- p(x,y), not q(y).
p(1,2) <- .
```

Given a Datalog program P, its *dependency graph* D is a directed graph $\langle N, A\rangle$ with N the set of all user-defined predicate symbols in P and $a \in A$ an edge from p to q if p and q are user-defined predicate symbols in the body and head clauses of a rule r, respectively. An arc between user-defined predicate symbols p and q is marked if the body clause that has p as predicate symbol is negative. P is stratified if there exists no cycle in D containing a marked arc.

The *stratum* of a node in the dependency graph is the maximum number of negative arcs on a path leading to that node. Intuitively, the stratum of a computed relation r is the maximum number of negations that can be applied in evaluating r.

When a Datalog program is stratified, we can designate a single Herbrand model as its semantics by evaluating all the rules of a stratum in the minimal model of the preceding stratum to obtain another minimal model. This unique minimal model is called the *perfect* model.

5.3. Computing a solution

Given a set of rules R, the *immediate consequence operator* Γ_R for normal Datalog programs is a mapping on sets of ground atoms and is defined for a Herbrand interpretation I as follows: $\Gamma_R(I) = \{A|A \in I$ or there exists a rule $A \leftarrow L_1 \wedge \cdots \wedge L_n \in [\![R]\!]$ such that $L_i \in I$ for all positive literals L_i and $L_j \notin I$ for all negative literals $L_j \equiv \neg L\}$, where $[\![R]\!]$ denotes the set of all ground instances of a set of rules R. $\Gamma_R^*(I)$ is the limit of the sequence of sets $J_0, J_1, \ldots$ such that $J_0 = I$ and $J_k = \Gamma_R(J_{k-1})$ for $k > 0$. For the purpose of this paper we will apply Γ_R on one rule at a time.

We can partition a stratifiable program P into a collection of sets of rules $\mathcal{P} = P_o, \ldots, P_n$ such that each P_j consists of the rules of P whose head relation is in stratum j. Let I_o be an initial Herbrand interpretation. For $0 < j \leq n$, the sequence of instances I_j is defined as follows:

$$I_j = I_{j-1} \cup \Gamma^*_{P_{j-1}}(I_{j-1}).$$

The final instance I_n provides the semantics of P under a stratification $P_o, \ldots, P_n$.

The *standard rule evaluation strategy* for a stratified program P

(1) computes a stratification of P
(2) partitions P into $P_0, \ldots, P_n$ such that each P_j consists of the rules of P whose head belongs to stratum j
(3) given an initial instance I_0, computes the instances I_j for $0 < j \leq n$ as above, yielding solution I_n.

Since we require that the full state of a reactor be computed during each reaction, we compute the interpretation of its Datalog program *exhaustively* (bottom-up), in contrast to some evaluation strategies that compute queries *on-demand* (top-down) as required by a particular query.

$$\frac{P_i \in \mathcal{P} \quad R \in P_i \quad I' = \Gamma_R(I) \quad I' \neq I}{\mathcal{P} \vdash \langle I, i \rangle \hookrightarrow \langle I', i \rangle}$$

$$\frac{i < numstrata(\mathcal{P}) \quad \forall R,\ (R \in P_i \text{ s.t. } P_i \in \mathcal{P}) \ \Rightarrow \ I = \Gamma_R(I)}{\mathcal{P} \vdash \langle I, i \rangle \hookrightarrow \langle I, i+1 \rangle}$$

Fig. 11. Small-step operational semantics for *eval*.

5.4. Additional semantic restrictions

A Datalog program is *domain independent* if the solution depends only on the initial set of facts and not on the universal set of all facts. A program is *weakly finite* [9] if applying the immediate consequence operator a finite number of times on a finite set of facts yields a finite set of results; i.e., infinite results can only be obtained via infinite recursion. Both domain independence and weak finiteness are desirable properties for Datalog programs intended to represent computations. Since these properties are in general undecidable, we will adopt a conservative syntactic characterization called *safety* [9] which guarantees domain independence and weak finiteness.

A Datalog rule is safe if all of its variables are *limited*. A variable is limited if:

(1) it occurs as an argument to a non-negated user-defined predicate in the body
(2) it occurs as one of the arguments to the built-in equality predicate and all of the other variables that occur in the same clause are limited.

A program is safe if all of its rules are safe. Consider a few examples:

```
answer(x) <- mynumber(x), not zero(x).  // 1
P(x) <- Q(y), R(z), x = y * z.          // 2
P(x) <- Q(y), R(z), y = x * z.          // 3
```

Rule (1) is safe, but removing `mynumber(x)` renders it unsafe because `x` then is not limited. Rule (2) is safe because `x` is the only non-limited argument to the equality predicate, hence its value is uniquely determined by `y` and `z`. By contrast, rule (3) is not safe, since both arguments contain non-limited variables and the value of `x` is not uniquely determined by `y` and `z`.

5.5. Datalog evaluation

The evaluator is defined as a relation $\hookrightarrow$ on pairs of *evaluation states*. An evaluation state is a pair $\langle I, i \rangle$ where

- I is a set of literals defining an *interpretation*.
- i is the index of the current stratum under evaluation.

Fig. 11 shows two small-step semantic rules which capture the behavior of the evaluator. The first rule specifies a single step in which the immediate consequence operator Γ_R is applied to the current Herbrand interpretation I to obtain the next interpretation I' for a given rule R. R is a non-deterministically chosen rule from the stratum under current evaluation $P_i \in \mathcal{P}$. If applying Γ_R modifies the state by inferring new facts, then the $\hookrightarrow$ relation transitions to the next state corresponding to I'. If, on the other hand, there exists no rule that can be evaluated which modifies the current state, then the evaluation of the current stratum ends and is marked appropriately by incrementing the stratum index. The evaluation stops when the last stratum has been fully evaluated.

6. Basic reactor formalities

In this section, we will define some basic reactor notation that will be used in the rest of the paper.

A reactor *program* $\mathcal{T}$ is a collection $T_1, T_2, \ldots, T_n$ of reactor *type definitions*, each defined using the REACTOR grammar of Fig. 1, with the proviso that a RULE-DECL may contain at most one instance of the `new` keyword in its HEAD-CLAUSE.

Each type definition T has the form

`def` *reactor-type-name* = $\cdots$

hence, we will frequently use *reactor-type-name* to denote the corresponding type definition.

Each reactor type T defines a collection of *ephemeral* relations *ephemeral*(T), declared using the syntactic form RELATION-DECL of Fig. 1 and the `ephemeral` keyword; similarly, it defines a collection of *persistent* relations *persistent*(T), declared using the same syntactic form without the `ephemeral` keyword. The *access attributes* `public`, `public read`, and `public write` may be used in the declaration to allow (possibly restricted) access to a relation by other reactors; otherwise, the relation is *private*, and not accessible to remote reactors. To simplify exposition, we will assume in the sequel that all

reactor names for a program $\mathcal{T}$ are globally unique. The set of persistent relations declared in $\mathcal{T}$ are given by *persistent*(T); the set of ephemeral relations declared in $\mathcal{T}$ are given by *ephemeral*(T).

A *local* literal is a literal of the form

[not] *ref*(...)

where *ref* has the form `t`, `-t`, `^t`, or `t^`, for some relation t.

A *remote* r-literal is a literal of the form

[not] r.*ref*(...)

where *ref* is as above and r is a variable bound to a reactor reference. If r is a reference to a reactor of type S, then the relation referred to in *ref* (say p) must be a public relation, i.e., declared using the `public` access attribute in the definition of S. Furthermore, if the remote r-reference occurs in a HEAD-CLAUSE, then p must be declared using either the `public` or `public write` attributes; if the reference occurs in a BODY-CLAUSE, then p must be declared using either the `public` or `public read` attributes.

A reactor reference of the form `r` denotes the *response value* of relation `r`; `-r` denotes the *prestate value* of `r`; `r^` denotes the *future value* of `r`, and `^r` denotes the *stimulus* value of `r`.

7. Semantics of reactor references

In this section, we show how we can "compile away" reactor constructs which manipulate reactor references by translating such constructs into ordinary Datalog. This translation process serves to define the semantics of reference-manipulating constructs.

The first phase of the reference translation process (Sections 7.1 and 7.2) formalizes when remote reactors are *read* and *written* by another reactor via a reactor reference. The formalization works by introducing two auxiliary relations for every reactor of type T: *active_T* and *touched_T*, and transforming the rules of the original program to use these new relations appropriately. Intuitively, *touched_T* will become true for reactor instance t of type T whenever a remote reactor accesses relations of t. Similarly, *active_T* will become true for reactor t whenever a remote reactor writes a relation of t. The relations *touched_T* and *active_T* will be examined by the concurrent transition semantics of Section 9 to determine when and how to access the state of remote reactors. Importantly, the relation *active_T* is also used to ensure that only the rules of reactors which have been *written to* are evaluated in the current reaction.

The second phase of the reference translation process (Section 7.3) encodes the semantics of synchronous reactions, in which multiple reactor instances which refer to one another via references are evaluated atomically. This *augmentation* translation adds an extra column to every relation r of reactor type T. This column contains reactor references (of type T) which encode the reactor instance which "owns" the tuples of r.

The final phase of the reference translation process (Section 7.4) formalizes the semantics of reactor instantiation and initialization, i.e., the `new` construct.

7.1. Formalizing remote reactor writes and reaction scope: The active transformation

The first step of the reference translation process makes reaction scope explicit by adding to all reactor definitions a relation called *active_T*, where T is the reactor's type. The *active_T* relation defines the semantics of "scope extrusion", the process by which multiple reactors react together synchronously. Only those reactor instances for which *active_T* is true are included in the scope of the current reaction. The basic idea is to set *active_T* to true for reactor instance t whenever a relation of t is written, and to guard all rules with a test of *active_T* in such a way as to ensure that a rule is only executed for reactor instances that are part of the current reaction scope.

In every reactor in the program the transformation adds the declaration

`public ephemeral` *active_T* : ().

We assume, without loss of generality, that there is no pre-existing relation called *active_T*.

Next, we non-recursively transform each rule such that

[not] $r.a(\mathbf{e})$ [^] <-*body*.	becomes	[not] $r.a(\mathbf{e})$ [^]	<-	*body*, *active_T* ().
		r.*active_U* () [^]	<-	*body*, *active_T* ().
[not] $a(\mathbf{e})$ [^] <-*body*.	becomes	[not] $a(\mathbf{e})$ [^]	<-	*body*, *active_T* ().

where T is the type name of the reactor containing the rules that are being transformed, and U is the type name of the reactor r.

Since all rules are now guarded by an *active_T* relation, the *active_T* relation of the reactor instance that initiated the reaction (i.e. received an update bundle) must be set to true. We therefore require that an update bundle write the *active_T* relation of the receiving reactor, but conveniently the transformation already handles this.

Example 1. Let us consider how an example program is transformed. The following example models a simple publisher/subscriber mechanism, but for simplicity the mechanism handling subscription and unsubscription have been left out. Here is what the program looks like before (left) and after (right) the transformation:

```
def Pub = {
  public data : (int).
  subscriber  : (ref Sub).

  s.copy(x)   <- data(x), subscriber(s).
}

def Sub = {
  copy        : (int).
  intStore    : (int).

  intStore(x) <- copy(x).
}
```

```
def Pub = {
  public data    : (int).
  subscriber     : (ref Sub).
  public ephemeral active_Pub : ().
  s.copy(x)      <- data(x), subscriber(s),
                    active_Pub().
  s.active_Sub() <- data(x), subscriber(s),
                    active_Pub().
}

def Sub = {
  public copy    : (int)
  intStore       : (int)
  public ephemeral active_Sub : ().
  intStore(x)    <- copy(x), active_Sub().
}
```

Example 2. Intuitively, the semantics of the *active_T* relations is intended to expose the control flow implicit in the scope extrusion process via an explicit data dependency. The example in the left column at first glance seems benevolent enough. Intuitively, however, we should not accept it, because `q` is not fully computed before it is tested for non-membership, but this test is necessary to extrude the scope to the reactor of type `R2` whose rule adds tuples to `q`. The right column shows the transformed example after adding the two *active_T* relations. The example is now non-stratifiable due to the relation `active_R2` for the reactor of type `R2`. This relation effectively acts as a "summary node"—a relation which guards all rules and is written when the scope extrudes. A negative cycle is created between `q`, `s`, and `active_R2`. This negative cycle makes concrete our intuition that the example in the left column should not be an acceptable program.

```
// The rules are stratifiable, but the
// program should intuitively be rejected

def R1 = {
  r         : (int, int, ref R2).
  public q  : (int).
  s         : (int).


  s(x)      <- not q(x).
  z.t(y, x) <- s(x), r(x, y, z).
}

def R2 = {
  p         : (int, int, ref R1).
  public t  : (int, int).


  y.q(x)    <- p(z, x, y).
}
```

```
// After introducing the active relation
// the hidden negative cycle is exposed

def R1 = {
  r              : (int, int, ref R2).
  public q       : (int).
  s              : (int).
  public active_R1 : ().

  s(x)           <- not q(x), active_R1().
  z.t(y, x)      <- s(x), r(x, y, z),
                    active_R1().
  z.active_R2()  <- s(x), r(x, y, z),
                    active_R1().
}

def R2 = {
  p              : (int, int, ref R1).
  public t       : (int, int).
  public active_R2 : ().

  y.q(x)         <- p(z, x, y), active_R2().
  y.active_R1()  <- p(z, x, y), active_R2().
}
```

7.2. Formalizing remote reactor reads: The touched *transformation*

In this section, we define the *touched_T* relation, which records those remote reactor instances of type T which could be accessed in the current reaction (this set can be an overapproximation of the set of reactor instances which are going to be read or written in the current reaction). For each reactor of type T, we add a declaration of the form

```
public ephemeral touched_T : ().
```

Let rule R be a rule of the form:

$$head \texttt{ <- } clause_1, \ldots, clause_n\,.$$

containing one or more remote literals r.

For each $clause_i$ which binds r we add the following additional rules to T's declaration:

$$r\,.\,touched_T_j() \quad \texttt{<-} \quad clause_i\,.$$

where T_j is the reactor type name of r respectively.

This construction ensures that the $touched_T_j$ relation of reactor instance r becomes true whenever a relation of r is read or written by another reactor.

7.3. *Flattening inter-reactor references: The* augmentation *transformation*

The *augmentation* transformation in this section readies reactors for synchronous reaction by allowing them to be treated together as single "virtual" reactor. The augmentation transformation is applied *after* the *active_T*, *touched_T*, and instantiation (new) transformations, which are each applied independently to source rules.

The program is given as a set of reactor declarations, named $R_1, \ldots, R_k$. We assume, without loss of generality, that relation names are globally unique, i.e. any relation name is declared at most once.

In each reactor declaration R rewrite each rule

$$\begin{array}{ll}[\texttt{not}]\ r.h(\mathbf{e})\,[\hat{}\,] \texttt{ <- } & r_1.\,[\hat{}|\texttt{-}]\,b_1(\mathbf{x_1}), \ldots, r_n.\,[\hat{}|\texttt{-}]\,b_n(\mathbf{x_n}),\\ & [\hat{}|\texttt{-}]\,c_1(\mathbf{y_1}), \ldots, [\hat{}|\texttt{-}]\,c_m(\mathbf{y_m}),\\ & predicates.\end{array}$$

as

$$\begin{array}{ll}[\texttt{not}]\ h(r, \mathbf{e})\left[\hat{}\,\right] \texttt{ <- } & [\hat{}|\texttt{-}]\,b_1(r_1, \mathbf{x_1}), \ldots, [\hat{}|\texttt{-}]\,b_n(r_n, \mathbf{x_n}),\\ & [\hat{}|\texttt{-}]\,c_1(s, \mathbf{y_1}), \ldots, [\hat{}|\texttt{-}]\,c_m(s, \mathbf{y_m}),\\ & predicates[s\backslash\texttt{self}].\end{array}$$

and rewrite

$$\begin{array}{ll}[\texttt{not}]\ h(\mathbf{e})\,[\hat{}\,] \texttt{ <- } & r_1.\,[\hat{}|\texttt{-}]\,b_1(\mathbf{x_1}), \ldots, r_n.\,[\hat{}|\texttt{-}]\,b_n(\mathbf{x_n}),\\ & [\hat{}|\texttt{-}]\,c_1(\mathbf{y_1}), \ldots, [\hat{}|\texttt{-}]\,c_m(\mathbf{y_m}),\\ & predicates.\end{array}$$

as

$$\begin{array}{ll}[\texttt{not}]\ h(s, \mathbf{e})\,[\hat{}\,] \texttt{ <- } & [\hat{}|\texttt{-}]\,b_1(r_1, \mathbf{x_1}), \ldots, [\hat{}|\texttt{-}]\,b_n(r_n, \mathbf{x_n}),\\ & [\hat{}|\texttt{-}]\,c_1(s, \mathbf{y_1}), \ldots, [\hat{}|\texttt{-}]\,c_m(s, \mathbf{y_m}),\\ & predicates[s\backslash\texttt{self}].\end{array}$$

where s is a fresh identifier, and $[x\backslash y]$ means "put x instead of y in the preceding expression".

The transformation does two things: it replaces all references to `self` with a fresh identifier, s, and it augments the relations by adding reactor references as the first column of every relation. Note that the transformation also alters the *active_T* relations added by the previous transformation. The occurrence of `active_T()` in the body of each rule gets replaced by `active_T(`s`)`; and r`.active_T()` in the head gets replaced by `active_T(`r`)`.

Example 3. Consider the example from last section. We now take the program with the *active* relation already added, and pass it through the augmentation transformation. Here is the program before (left) and after (right):

```
def Pub = {
  public data    : (int).
  subscriber     : (ref Sub).
  public ephemeral active_Pub : ().
  s.copy(x)      <- data(x), subscriber(s),
                    active_Pub().
  s.active_Sub() <- data(x), subscriber(s),
                    active_Pub().
}

def Sub = {
  public copy    : (int)
  intStore       : (int)
  public ephemeral active_Sub : ().
  intStore(x)    <- copy(x), active_Sub().
}
```

```
// Rules from Publisher



copy(s,x) <- data(id1,x),subscriber(id1,s),
             active_Pub(id1).
active_Sub(s) <-
             data(id2,x),subscriber(id2,s),
             active_Sub(id2).

// Rules from Subscriber




intStore(id3,x) <-
             copy(id3,x),active_Sub(id3).
```

7.4. Formalizing reactor creation: The instantiation transformation

The `new` construct is used to instantiate new reactors. To model this behavior, we must generate fresh reactor references appropriately for each rule containing `new` (recall that `new` can occur at most once in any rule head). But what behavior is "appropriate"?

Consider the following examples. We would like the rule

```
r(new) <- .
```

to produce exactly one new reactor per reaction. The rule

```
r(x, new) <- t(x)
```

should generate a distinct reactor for every value of `x`, in every reaction. A program containing the rules

```
s(x, new) <- t(x). // 1
s(x, new) <- t(x). // 2
```

should produce the same result as a program containing a single instance of the rule; moreover, replacing rule (2) above with the rule `s(x, new) <- q(x)` should also yield the same result when the contents of `t` and `q` are identical.

In general, we want a rule of the form

$r(\mathbf{x}_{1,i},\ \texttt{new},\ \mathbf{x}_{i+1,n})$ `<-`*body*.

to generate a fresh reference value for every tuple of values $\langle \mathbf{x}_{1,i}, \mathbf{x}_{i+1,n} \rangle$ which satisfy the rule, in every reaction.

We make the intuition above precise by transforming rules containing `new` to standard Datalog, augmented with functors (term constructors).

In every reactor of type T in the program, the transformation adds the declaration

`public ephemeral` *reactions_T* `: (int).`

The idea is that *reactions_T* relations maintain a count of the number of reactions that the containing reactor has undergone since being instantiated. This count will be used to ensure that distinct reference values are appropriately generated for each reaction.

After applying the augmentation transformation of Section 7.3, each rule in a reactor of type T containing an instance of `new` will have the form

$r(s,\ \mathbf{x}_{1,i},\ \texttt{new},\ \mathbf{x}_{i+1,n})$ `<-` *body*, *active_T*(s).

Let us assume that relation `r` has a declaration of the form

`...` r: $(\mathbf{T}_{1,i},\ \texttt{ref}\ S,\ \mathbf{T}_{i+1,n})$.

where $\mathbf{T}_{1,i}$ and $\mathbf{T}_{i+1,n}$ are vectors of types.

We transform the rule above into the following three rules:

$r(s,\ \mathbf{x}_{1,i},\ \texttt{s'},\ \mathbf{x}_{i+1,n})$ `<-` *body*, *reactions_T*(s, `j`), *active_T*(s)
  `s' =` *new_r_i*`<`s, `j`, $\mathbf{x}_{1,i}$, $\mathbf{x}_{i+1,n}$`>.` // 1

active_S(`s'`) `<-` *body*, *reactions_T*(s, `j`), *active_T*(s)
  `s' =` *new_r_i*`<`s, `j`, $\mathbf{x}_{1,i}$, $\mathbf{x}_{i+1,n}$`>.` // 2

reactions_S(`s', k`) `<-` *body*, *reactions_T*(s, `j`), *active_T*(s)
  `s' =` *new_r_i*`<`s, `j`, $\mathbf{x}_{1,i}$, $\mathbf{x}_{i+1,n}$`>, k = 0.` // 3

Here, *new_r_i* $<\cdots>$ is a functor (data constructor) defined for each relation `r` in which `new` appears in position i in the head of some rule. The first argument (s) of the functor encodes a reference to the reactor's "parent" (i.e., the reactor generating the new reference); the second argument (`j`) is the current reaction count, and the remaining arguments contain the values to which the other variables in the original rule are bound. Embedding the parent reference in the encoding of a newly-generated (child) reference value ensures that the each reference value is globally unique. In addition to "installing" the newly-generated reference in its containing relation (rule 1 above), we must also "activate" the reactor rules for the type corresponding to the reference (rule 2), and initialize the reaction count for the newly-generated reactor to 0 (rule 3)

Finally, for every reactor of type T, we add the following two rules

reactions_T(s, `j'`) `<-` `-`*reactions_T*(s, `j`), *active_T*(s), `j' = j+1.` // 4
reactions_T(s, `j`) `<-` `-`*reactions_T*(s, `j`), *active_T*(s). // 5

Rules 4 and 5 together increment the reaction count for a reactor of type T on every reaction.

It is straightforward to define a total order on generated reactor references (e.g., based on a total ordering for relation names). This property will be used in the operational semantics defined in Section 9.

Rewrite:	$\mathtt{r}_i(\mathbf{x})$ <- *body*.	as:	$\underline{\mathtt{r}}_i^{\Delta+}(\mathbf{x})$ <- *body*.	(I)
Rewrite:	not $\mathtt{r}_i(\mathbf{x})$ <- *body*.	as:	$\underline{\mathtt{r}}_i^{\Delta-}(\mathbf{x})$ <- ^$\mathtt{r}_i(\mathbf{x})$, *body*.	(II)
Rewrite:	*head* <- $\mathtt{r}_i(\mathbf{x})$, *body*.	as:	*head* <- ^$\mathtt{r}_i(\mathbf{x})$, not $\underline{\mathtt{r}}_i^{\Delta-}(\mathbf{x})$, *body*. *head* <- $\underline{\mathtt{r}}_i^{\Delta+}(\mathbf{x})$, *body*.	(III)
Rewrite:	*head* <- not $\mathtt{r}_i(\mathbf{x})$, *body*.	as:	*head* <- not ^$\mathtt{r}_i(\mathbf{x})$, not $\underline{\mathtt{r}}_i^{\Delta+}(\mathbf{x})$, *body*. *head* <- $\underline{\mathtt{r}}_i^{\Delta-}(\mathbf{x})$, *body*.	(IV)
Rewrite:	r^$_i(\mathbf{x})$ <- *body*.	as:	$\underline{\mathtt{r}}$^$_i^{\Delta+}(\mathbf{x})$ <- *body*.	(V)
Rewrite:	not r^$_i(\mathbf{x})$ <- *body*.	as:	$\underline{\mathtt{r}}$^$_i^{\Delta-}(\mathbf{x})$ <- *body*.	(VI)

Fig. 12. Rewrite rules defining the semantics of reactor rule evaluation in terms of normal datalog.

8. Eliminating head negation

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. If the reactor has just been created all the relations are empty by default; ephemeral relations are always initially empty. Let ^$\mathtt{r}_i^{\Delta+}$ and ^$\mathtt{r}_i^{\Delta-}$ be the addition and the deletion sets of the update bundle applied to the reactor. We will use r $(\mathbf{x})$ to denote the relation r applied to a tuple of arguments $(\mathbf{x})$. The basic idea for determining the solution to r_i is to:

(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 $\underline{\mathtt{r}}_i^{\Delta-}$.

The resulting program is the input to the evaluator presented in Section 5.5. Fig. 12 shows how our rewriting technique transforms a program with negation in the head clauses to a program without them. The rules apply recursively. Let us define $\underline{\mathtt{r}}_i^{\Delta+}$, $\underline{\mathtt{r}}_i^{\Delta-}$, $\underline{\mathtt{r}}$^$_i^{\Delta+}$ and $\underline{\mathtt{r}}$^$_i^{\Delta-}$ the sets of additions and deletions to the response and future states of the persistent and ephemeral relations.

Rewrite rule (I) computes the set of tuples to be added to $\mathtt{r}_i$ 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 in the stimulus state. The extra clause ensures that this rewriting rule does not introduce domain dependence—see Section 10.1 for details. Our rewriting approach does not update relations in place during a reaction; accounting for the addition and the deletion sets in the rules must be therefore done explicitly by modifying the rules. As a result, rewrite rule (III) restricts the matching for the tuples in r_i to the ones that are not in the deletion set; it also adds another rule which matches tuples in the addition set. Conversely, rule (IV) allows matching on tuples in the deletion set, as well as restricting matching to tuples not in the addition set. Rules (V) and (VI) do not apply for ephemeral relations because they do not have a future state.

Let us consider the order entry example in Fig. 2. After applying the head negation transformation the rule in the reactor of type OrderEntryA' will result in the following set of rules:

```
log^Δ+(id, itemid, qty)  <-  orders(id, itemid, qty).
log^Δ-(id, itemid, qty)  <-  ^log(id, itemid, qty),
                             not ^orders(id, itemid, qty),
                             not orders^Δ+(id, itemid, qty).
log^Δ-(id, itemid, qty)  <-  ^log(id, itemid, qty),
                             orders^Δ-(id, itemid, qty).
```

Note that the tuple (id, itemid, qty) will be deleted from log if it is in the prestate and either is in the deletion set of orders or it is neither in the prestate, nor in the addition set of orders.

9. Semantics of reactor interaction

In this section, we define the operational semantics of distributed reactor interaction, using the rules depicted in Fig. 13. These rules effectively define a "scheduler" that initiates concurrent reactions and orchestrates the interchange of data

$$\text{START}\ \frac{b = (\beta, I^{\Delta}) \quad i = 0 \quad I' = \text{init}(I_0, I^{\Delta})}{\mathcal{P}, N \uplus \{b\} \vdash (\beta, I_0) \xrightarrow{start} \mathcal{P}, N \vdash \langle b, I', i \rangle}$$

$$\text{EVAL}\ \frac{\mathcal{P} \vdash \langle I, i \rangle \hookrightarrow \langle I', i' \rangle \quad \underline{\texttt{touched_}T}^{\Delta^+}(\alpha) \in I' \implies \underline{\texttt{touched_}T}^{\Delta^+}(\alpha) \in I}{\mathcal{P}, N \vdash \langle b, I, i \rangle, \xrightarrow{eval} \mathcal{P}, N \vdash \langle b, I', i' \rangle}$$

$$\text{READ}\ \frac{\mathcal{P} \vdash \langle I, i \rangle \hookrightarrow \langle I', i' \rangle \quad \underline{\texttt{touched_}T}^{\Delta^+}(\alpha'') \in I' \setminus I \quad I''' = I' \cup \text{init}(I''_0, \emptyset)}{\mathcal{P}, N \vdash \langle b, I, i \rangle, (\alpha'', I''_0) \xrightarrow{read} \mathcal{P}, N \vdash \langle b, I''', i' \rangle, (\alpha'', I'')}$$

$$\text{CONFLICT}\ \frac{\text{conflict}(I) \quad \{(\alpha'', I'')\} = \text{revert}(\{\alpha''\}, I') \quad b = (\alpha, I^{\Delta})}{\mathcal{P}, N \vdash \langle b, I, i \rangle \xrightarrow{conflict} \mathcal{P}, N \vdash (\alpha'', I'')}$$

$$\text{PREPARE}\ \frac{i = numstrata(\mathcal{P}) \qquad \neg\text{conflict}(I) \quad L = \{\alpha_i \mid \underline{\texttt{active_}T}^{\Delta^+}(\alpha_i) \in I\} \setminus \beta \quad b = (\beta, I^{\Delta})}{\mathcal{P}, N \vdash \langle b, I, i \rangle \xrightarrow{prepare\ to\ lock} \mathcal{P}, N \vdash \langle b, I, i \rangle_{\{\},L}}$$

$$\text{ACQUIRE}\ \frac{\text{uptodate}((\alpha_i, I_i), I) \quad \forall \alpha_j,\ i < j \le n \implies \alpha_i < \alpha_j}{\mathcal{P}, N \vdash \langle b, I, i \rangle_{\{\alpha_1, \ldots, \alpha_{i-1}\}, \{\alpha_i, \alpha_{i+1}, \ldots, \alpha_n\}}, (\alpha_i, I_i) \xrightarrow{acquire\ lock} \mathcal{P}, N \vdash \langle b, I, i \rangle_{\{\alpha_1, \ldots, \alpha_i\}, \{\alpha_{i+1}, \ldots, \alpha_n\}}}$$

$$\text{RETRY}\ \frac{\neg\text{uptodate}((\alpha_i, I_i), I) \qquad \{(\alpha', I'), (\alpha_1, I_1), \ldots, (\alpha_{i-1}, I_{i-1})\} = \text{revert}(\{\alpha', \alpha_1, \ldots, \alpha_{i-1}\}, I) \quad b = (\alpha', I^{\Delta})}{\mathcal{P}, N \vdash \langle b, I, i \rangle_{\{\alpha_1, \ldots, \alpha_{i-1}\}, \{\alpha_i, \alpha_{i+1}, \ldots, \alpha_n\}}, (\alpha_i, I_i) \xrightarrow{snapshot\ stale/retry} \mathcal{P}, N \uplus \{b\} \vdash (\alpha', I'), (\alpha_1, I_1), \ldots, (\alpha_{i-1}, I_{i-1}), (\alpha_i, I_i)}$$

$$\text{COMMIT}\ \frac{\{(\alpha_1, I_1) \ldots, (\alpha_n, I_n)\} = \text{persist}(I) \quad \{b_1, \ldots, b_m\} = \text{bundles}(I)}{\mathcal{P}, N \vdash \langle b, I, i \rangle_{\{\alpha_1, \ldots, \alpha_n\}, \{\}} \xrightarrow{commit} \mathcal{P}, N \uplus \{b_1, \ldots, b_m\} \vdash (\alpha_1, I_1), \ldots, (\alpha_n, I_n)}$$

$$\text{CONTEXT}\ \frac{\mathcal{P}, N \vdash q_{l+1}, \ldots, q_k, \langle\cdot\rangle_{m+1}, \ldots, \langle\cdot\rangle_n \xrightarrow{<any>} N' \vdash q'_{l+1}, \ldots, q'_{k'}, \langle\cdot\rangle'_{m+1}, \ldots, \langle\cdot\rangle'_{n'}}{\mathcal{P}, N \vdash q_1, \ldots, q_l, q_{l+1}, \ldots, q_k, \langle\cdot\rangle_1, \ldots, \langle\cdot\rangle_m, \langle\cdot\rangle_{m+1}, \ldots, \langle\cdot\rangle_n \xrightarrow{<any>} N' \vdash q_1, \ldots q_l, q'_{l+1}, \ldots, q'_{k'}, \langle\cdot\rangle_1, \ldots, \langle\cdot\rangle_m, \langle\cdot\rangle'_{m+1}, \ldots, \langle\cdot\rangle'_{n'}}$$

Fig. 13. High-level operational semantics.

among reactor instances. The transition relation of the operational semantics defines a computation on *worlds*. A world has the form

$$\mathcal{P}, N \vdash q_1, \ldots, q_k, s_{k+1}, \ldots, s_n.$$

The components of a world are defined as follows:

- The *program* $\mathcal{P}$ is an ordered collection of subprograms $P_1, \ldots, P_m$ derived from a given reactor program $\mathcal{T}$ by first transforming $\mathcal{T}$ using the transformations described in Sections 7 and 8, then stratifying the resulting collection of Datalog rules into subprograms $P_1, \ldots, P_m$, as described in Section 5.2.
- The *network* N is a multiset of *update bundles*, each of which has the form (β_i, I_{β_i}), where β is a reactor reference, and I is a Herbrand interpretation containing only facts of the form $\underline{\mathtt{r}}\hat{}^{\Delta^+}(\beta, \mathbf{x})$ or $\underline{\mathtt{r}}\hat{}^{\Delta^-}(\beta, \mathbf{x})$, representing future additions or deletions, respectively, to reactor β
- A set of *initiating reactors* $s_{k+1}, \ldots, s_n$. An initiating reactor is a reactor at which an uncompleted reaction has been initiated. Each s_j takes the form $\langle b_{\alpha_j}, I_{\alpha_j}, i_j \rangle$, where b_{α_j} is an update bundle, I_{α_j} is an interpretation (over all of the relations defined by the transformation of $\mathcal{T}$), and i_j is an integer defining the current stratum of $\mathcal{P}$ being executed by the reactor.
- A set of *quiescent reactors* $q_1, \ldots, q_k$. A quiescent reactor is a reactor that is not an initiation site for an active reaction. Each q_i takes the form (α_i, I_{α_i}), where α is a reactor reference and I_{α_i} is an interpretation containing only facts of the form $\mathtt{-r}(\alpha, \mathbf{x})$. We thus see the *persistent state* of a reactor is encoded by terms over pre-state values of relations.

Each reactor in a world occurs exactly once in the world state, either as a quiescent reactor or as an initiating reactor.

In the world representation described above, we view a relation as a set of *facts*, i.e., a ground Herbrand interpretation, rather than as a named collection of tuples. While the two views of state are semantically equivalent, the "soup of atoms" Herbrand representation accommodates state manipulation involving multiple relations more readily than a state representation using sets of maps from names to tuples.

The rules of Fig. 13 require a number of auxiliary functions, primarily to manage operations on the Herbrand interpretations representing reactor states. The formal definitions of these functions are given in Fig. 14, here we provide more intuitive definitions.

$\uplus$: Multiset union.

$\hookrightarrow$: The Datalog evaluation relation as defined in Section 5.5. Note that the translations of Sections 7 and 8 have the property that when a remote reactor is *written*, the reactor's reference is added to a relation of the form $\underline{\texttt{active_}T}^{\Delta^+}$; when it is *read*, the reference is added to a relation of the form $\underline{\texttt{touched_}T}^{\Delta^+}$.

$\text{init}(I_0, I^\Delta)$: Given an an interpretation I_0 representing the persistent state of a quiescent relation and an interpretation I^Δ representing the contents of an update bundle, yields an initial interpretation I' suitable for evaluation. I' contains I_0, as well as stimulus values for relations generated by applying I^Δ to I_0.

$\text{conflict}(I)$: Given an interpretation I, determines whether I contains a *conflict*, i.e., whether the encoding of response or future state in I contains both a fact and its negation.

$\text{revert}(S, I)$: Given a set of reactor references S and an interpretation I representing the current state of the reaction, yields a collection of quiescent reactors of the form (α, I_α) such that $\alpha \in S$, and I_α contains only the prestate values of α's relations. In other words, this operation "rolls back" the state of the reactors in S to their pre-reaction values.

$\text{persist}(I)$: Given an interpretation I representing the current state of the reaction, yields a collection of quiescent reactors of the form (α_i, I_{α_i}), whose prestate values are computed from response values of the corresponding relations in I. The response values are in turn computed from relations of the form $\underline{\texttt{r}}^{\Delta^+}$ and $\underline{\texttt{r}}^{\Delta^-}$ and the stimulus values of the corresponding relations. The references α_i correspond to the active (written) reactors in the current reaction. This operation has the effect of updating the persistent state of reactors involved in the reaction with the new state resulting from the reaction.

$\text{bundles}(I)$: Given an interpretation I representing the current state of the reaction, yields a collection of update bundles corresponding to the future state values of relations computed in I.

$\text{uptodate}(q, I)$: Given the contents of a quiescent reactor q and an interpretation I representing the current state of the reaction, determines whether the response values of relations of q in I are identical to the prestate values of the corresponding relations in q, i.e., that the state "snapshot" created when q's state was incorporated into the current reaction has not been invalidated by updates to q made by other reactions.

Let us now look at the intuition behind each of the rules in the semantics. It may be useful to read the following descriptions in parallel with the inference rules given in Fig. 13.

START A reactor takes an update bundle addressed to it off the network and applies the update bundle to obtain the stimulus state used for the reaction. It also sets the index of the current stratum that is to be evaluated to zero, the bottom stratum.

EVAL This corresponds to a single step of the evaluator, where no further intervention by the scheduler is required.

READ If, on inspection of the evaluator state, the scheduler observes that at least one new reactor reference was added to a relation of the form $\underline{\texttt{touched_}T}^{\Delta^+}$, then for each new reactor reference in $\underline{\texttt{touched_}T}^{\Delta^+}$, the scheduler creates a "snapshot" of its state and copies it into the state of the reaction.

CONFLICT If the reaction state contains a conflict, the reaction rolls back. The update bundle that initiated the reaction is not re-queued on the network.

PREPARE If all strata have been fully evaluated (the evaluation has reached a fixpoint) and no conflict was detected during evaluation, then the scheduler prepares to lock all reactor instances that are active in the current reaction (except for the initiating reactor, which is effectively already locked). The set of locks to be acquired is represented by the set L.

ACQUIRE If the state of a reactor instance that is to be locked has not changed since its snapshot was taken, we lock the reactor instance by removing it from the externally-visible world (its state is preserved in the reaction state I). Since reactor references are totally ordered (see Section 7.4), we acquire locks in order of reference value to avoid deadlocks.

RETRY Conversely, if the state of a reactor instance that is to be locked has changed since its snapshot was taken, then the reaction rolls back and the update bundle that initiated the reaction is re-queued on the network; the reaction can thus be re-initiated in the future. This rule effectively encodes a form of optimistic transaction.

COMMIT If all reactor instances involved in a reaction were successfully locked, we generate the persistent values of the written relations, release them for future reactions, and add update bundles to the network. This rule also has the effect of "materializing" newly-instantiated reactors created during the reaction.

ANY This rule states that any combination of reactors that can react in isolation can react in any context containing them.

$$\text{active}(I) \triangleq \{s \mid \underline{\texttt{active_}T}^{\Delta^+}(\alpha) \in I \text{ for some type } T\}$$

$$I \downarrow \alpha \triangleq \{r(\alpha, \mathbf{x}) \mid r(\alpha, \mathbf{x}) \in I, \text{ where } r \text{ is an arbitrary relation name}\}$$

$$\text{init}(I_0, I^{\Delta}) \triangleq I_0 \quad \cup \quad \{ \ \hat{}\mathtt{r}(\mathbf{x}) \mid (\text{-}\mathtt{r}(\mathbf{x}) \in I_0 \text{ and } \underline{\mathtt{r}}\hat{}^{\Delta^-}(\mathbf{x}) \notin I^{\Delta}) \text{ or } (\underline{\mathtt{r}}\hat{}^{\Delta^+}(\mathbf{x}) \in I^{\Delta}) \ \}$$

$$\text{conflict}(I) \triangleq \text{there exists } \mathbf{x} \text{ such that } (\underline{\mathtt{r}}^{\Delta^+}(\mathbf{x}),\ \underline{\mathtt{r}}^{\Delta^-}(\mathbf{x}) \in I) \text{ or } (\underline{\mathtt{r}}\hat{}^{\Delta^+}(\mathbf{x}),\ \underline{\mathtt{r}}\hat{}^{\Delta^-}(\mathbf{x}) \in I)$$

$$\text{revert}(S, I) \triangleq \{ \ (\alpha, I_\alpha) \mid \alpha \in S,\ I' = I \downarrow \alpha,\ I_\alpha = \{\text{-}\mathtt{r}(\mathbf{x}) \mid \text{-}\mathtt{r}(\mathbf{x}) \in I'\} \ \}$$

$$\text{persist}(I) \triangleq \{ \ (\alpha, I'') \mid \alpha \in \text{active}(I),\ I' = I \downarrow \alpha,\ I'' = \{ \ \text{-}\mathtt{r}(\mathbf{x}) \mid \mathtt{r} \in \mathit{persistent}(\mathcal{T}),\ (\ (\hat{}\mathtt{r}(\mathbf{x}) \in I' \text{ and } \underline{\mathtt{r}}^{\Delta^-}(\mathbf{x}) \notin I') \text{ or } (\underline{\mathtt{r}}^{\Delta^+}(\mathbf{x}) \in I') \ ) \ \} \ \}$$

$$\text{bundles}(I) \triangleq \{ \ (\alpha, I^{\Delta}) \mid I' = I \downarrow \alpha,\ I^{\Delta} = \{\underline{\mathtt{r}}\hat{}^{\Delta^+}(\mathbf{x}) \mid \underline{\mathtt{r}}\hat{}^{\Delta^+}(\mathbf{x}) \in I'\} \cup \{\underline{\mathtt{r}}\hat{}^{\Delta^-}(\mathbf{x}) \mid \underline{\mathtt{r}}\hat{}^{\Delta^-}(\mathbf{x}) \in I'\} \ \}$$

$$\text{uptodate}((\alpha, I_\alpha), I) \triangleq (\text{-}\mathtt{r}(\mathbf{x}) \in I_\alpha) \iff (\text{-}\mathtt{r}(\mathbf{x}) \in (I \downarrow \alpha))$$

Fig. 14. Auxiliary functions used in the operational semantics.

10. Translation properties

This section presents results which show that the program transformation which eliminates negation in head clauses preserves two important properties we care about: safety and stratification. Recall that the rewriting transformation assumes no remote references to other reactor instances are present. We then define what it means for a program before applying the augmentation transformation to be *D-stratified*. Lastly we informally show that the augmentation transformation yields stratifiable augmented programs.

10.1. Removing head negation

Given a Datalog program P with possible negative head clauses, the *rule/relation dependency graph* G is a directed graph $\langle N, R, A\rangle$ with N the set of all predicate symbols in P, R the set of all rules in P, and $a \in A$ either one of the following:

- An edge from $n \in N$ to $r \in R$ if n is a predicate symbol in the body of rule r. An edge between $n \in N$ and $r \in R$ is marked if the body clause that has n as predicate symbol is negative.
- An edge from $r \in R$ to $n \in N$ if n is a predicate symbol in the head of rule r. An edge between $r \in R$ and $n \in N$ is marked if the head clause that has n as predicate symbol is negative.

Definition 4. A program is *head-negation-stratified* if there exists no cycle in the corresponding rule/relation dependency graph G containing a marked edge. □

Theorem 5. *If a program with negation in head clauses is head-negation-stratified then the program obtained via the transformation in Fig. 12 is stratified.*

Proof. Starting from a dependency graph with no negative cycles, the only way to get a negative cycle is by adding edges—from either extra body clauses or new rules altogether. Adding a negatively marked edge between two nodes already reachable from each other can create a negative loop, but adding a positive edge can also have the same effect if the existing path contains a negatively marked edge. Given that the original dependency graph has no negative cycles, a negative cycle can only be created through the newly created relations, $\underline{\mathtt{r}}_i^{\Delta^+}$ and $\underline{\mathtt{r}}_i^{\Delta^-}$. For a cycle to be created it is necessary that it exist a rule that reads, and another one that writes, a newly created relation.

- Rewrite rules (V) and (VI) write the future states $\underline{\mathtt{r}}\hat{\ }_i^{\Delta^+}$ and $\underline{\mathtt{r}}\hat{\ }_i^{\Delta^-}$, but the future state cannot be read, and therefore they cannot create a loop.
- Rewrite rule (I) writes $\underline{\mathtt{r}}_i^{\Delta^+}$ and rewrite rules (III) and (IV) read it.
  Rules (I) and (IV) can create a negative loop only if there also exists another rule *body* $\leftarrow$ *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^+}$, *body*, and *head*. But in this case, the original program would not be head-negation-stratifiable as it would have a negative loop between *head*, *body*, and $\mathtt{r}_i$. Therefore rules (I) and (IV) cannot create a negative loop.
  Rules (I) and (III) can create a negative loop only if there also exists a rule of the form `not` $\mathtt{r}_i \leftarrow$ *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^-}$ and *head*. But in this case, the original program would not be head-negation-stratifiable as it would have a negative loop between $\mathtt{r}_i$ and *head*. Therefore rules (I) and (III) cannot create a negative loop.
- Rewrite rule (II) writes $\underline{\mathtt{r}}_i^{\Delta^-}$, and rewrite rules (III) and (IV) read it.
  Rules (II) and (IV) can create a negative loop only if there also exists another rule $\mathtt{r}_i \leftarrow$ *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^+}$ and *head*. But in this case, the original program would not be head-negation-stratifiable as it would have a negative loop between $\mathtt{r}_i$ and *head*. Therefore rules (II) and (IV) cannot create a negative loop.
  Rules (II) and (III) can create a negative loop only if there also exists a rule of the form `not` $\mathtt{r}_i \leftarrow$ *head*; the loop would then contain $\underline{\mathtt{r}}_i^{\Delta^-}$ and *head*. But in this case, the original program would not be head-negation-stratifiable as it would have a negative loop between $\mathtt{r}_i$ and *head*. Therefore rules (II) and (III) cannot create a negative loop. □

Definition 6. A rule is *head-negation-safe* if all of its variables are *head-negation-limited*. A variable is head-negation-limited if it occurs as:

- an argument to a non-negated user-defined predicate in the body
- an argument to a negated user-defined predicate in the body and it is used only in negated head user-defined predicates involving response states
- one of the arguments to the built-in equality predicate and all of the other variables that occur in the same clause are limited.

A program is *head-negation-safe* if all of its rules are head-negation-safe. □

Intuitively, the reason for allowing variables occurring in some negated body clause *B* to appear in the negated head clause *H* is that the rule has to remove every value occurring in *H* which is not in *B*. It does not need to compute the set of all values not occurring in *B*—which may not be finite.

Theorem 7. *If a program with negation in head clauses is head-negation-safe then the program obtained via the transformation in Fig. 12 is safe.*

Proof. We will show that this property holds for each program rule `r` that is rewritten.
Rewrite rule (I):

- If `r`'s *body* has no negative clause then the transformation trivially preserves safety.
- Otherwise the rule does not apply.

Rewrite rule (II):

- If `r`'s *body* has no negative clause: the transformation trivially preserves safety.
- If `r`'s *body* contains at least one negated clause and the variables occurring in this clause are used only in the negated head clause: the rewrite rule transforms the negated head clause into a positive head clause which can use variables occurring only in the original negated body clause. Nevertheless, the rewrite rule also adds a corresponding positive body clause which now bounds the variables used by the head clause. This renders the transformed `r` rule safe.

Rewrite rule (III):

- If `r`'s *body* has no negative clause, the concern is that the variable occurring in the newly added negative clause may not be bounded in a positive body clause but used in the head clause; this is obviously false by construction.
- If `r`'s *body* contains at least one negated clause and the variables occurring in this clause are used only in the negated head clause, the rewrite rule leaves *body* and *head* unmodified and therefore cannot affect the result of the transformation.

Rewrite rule (IV):

- If `r`'s *body* has no negative clause, the concern is that *head* is using variables occurring solely in the negated body clause. This property would then hold for the transformed `r`, which thus would not be safe. Rewrite rule (II) applies now, and will result in a safe rule.
- If `r`'s *body* contains at least one negated clause and the variables occurring in this clause are used only in the negated head clause, the rewrite rule preserves safety up to rule (II).

Rewrite rule (V): The argument is the same as for rewrite rule (I).
Rewrite rule (VI):

- If `r`'s *body* has no negative clause: the transformation trivially preserves safety.
- Otherwise the rule does not apply. □

10.2. Augmentation and stratification

Given an initial reactor type C, and given the types of all reactors, we define the *extended rule/relation stratification graph* G on the program before augmentation as a directed graph $\langle N, R, A\rangle$ obtained by applying the following steps repeatedly until the reaction scope has been fully extruded—i.e. no new reactor types can possibly be added to the reaction scope. Let $S = \{C\}$.

- Let C_{crt} iterate over the set of newly added reactor types in S.
- For every rule r in C_{crt} which writes a remote relation p in C_i of a type that already exists in S, treat p as a local relation.
- For every rule in C_{crt} which reads and writes only local relations, build the rule/relation graph as explained in Section 10.1.
- For every rule r in C_{crt} which writes a remote relation p in C_i of a type that does not already exist in S:
  - . Add a node to N for each predicate symbol in C_i, and a node to R for each rule in C_i.
  - . For each rule r' in the current extended stratification graph which read relations in C_i, update A to contain an edge from $n \in N$ corresponding to the relation being read to $r' \in R$ if n is a predicate symbol in the body of rule r'. The edge between $n \in N$ and $r' \in R$ is marked if the body clause that has n as predicate symbol is negative.
  - . For rule r build the rule/relation graph as explained in Section 10.1.
  - . $S = S \cup C_i$.
- For every rule r in C_{crt} which instantiates a new reactor of type C_j that does not already exist in S, $S = S \cup C_j$.

Definition 8. A set of reactor types is *D-stratified* if there exists no cycle in the corresponding extended stratification graph G containing a marked edge. □

Note that we are interested in a program to be stratified after the *active_T* relation has been added, for reasons explained in Section 7.1.

Theorem 9. *If a set of reactor types is D-stratified then the program obtained via augmentation is head-negation-stratified.*

The augmentation transformation creates a global relation for all corresponding relations p in reactors of the same type. But, before augmentation all relations p in reactors of identical type that the rules may refer to were represented by a single node via the graph construction above. Therefore, no edges are either created, nor deleted, by the transformation, and the marks on the edges remain unchanged. Practically, the augmentation transformation creates a monolithic dependence graph which is isomorphic to the extended stratification graph constructed above. It follows that stratification of the graph is unaffected by the augmentation transformation.

11. Extended examples

In this section, we complete the exposition of the reactor model with a collection of extended examples. These examples show how a number of common distributed programming patterns can be concisely encoded in the reactor model. All of the examples in this section make extensive use of the notational abbreviations defined in Fig. 6.

Example 10 (*AJAX-Style Web Form*). Figs. 15–17 depict code for a simple AJAX (asynchronous Javascript and XML) web application, due to Bercik [17]. In this style of application, individual elements on a web page can be updated via asynchronous server requests. In contrast to traditional "page at a time" web applications, this approach minimizes the amount of data that needs to be exchanged between browser and server, which can in turn yield more responsive user interfaces and a reduced load on the server.

Figs. 15–17 represent browser code, server code, and database code, respectively, for a web form containing fields in which a user can enter a US zip (postal) code, a city name, and a US state name. When the user enters a zip code, a background server request (`XMLHttpRequest`) is dispatched, which ultimately has the effect of looking up the zip code in a database and returning the city and state to which the zip code corresponds back to the browser. The returned values are then used to complete the city and state fields on the web form (the user can overwrite the server-generated field values if desired).

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
<title>ZIP Code to City and State using XmlHttpRequest</title>
<script language="javascript"  type="text/javascript">
var url = "getCityState.php?param="; // The server-side script
function handleHttpResponse() {
  if (http.readyState == 4) {
    if (http.responseText.indexOf('invalid') == -1) {
      // Use the XML DOM to unpack the city and state data
      var xmlDocument = http.responseXML;
      var city = xmlDocument.getElementsByTagName('city').item(0).firstChild.data;
      var state = xmlDocument.getElementsByTagName('state').item(0).firstChild.data;
      document.getElementById('city').value = city;
      document.getElementById('state').value = state;
      isWorking = false;
    }
  }
}
var isWorking = false;
function updateCityState() {
  if (!isWorking && http) {
    var zipValue = document.getElementById("zip").value;
    http.open("GET", url + escape(zipValue), true);
    http.onreadystatechange = handleHttpResponse;
    isWorking = true;
    http.send(null);
  }
}
function getHTTPObject() {
  var xmlhttp;

  if (!xmlhttp && typeof XMLHttpRequest != 'undefined') {
    try {
      xmlhttp = new XMLHttpRequest();
          xmlhttp.overrideMimeType("text/xml");
    } catch (e) {
      xmlhttp = false;
    }
  }
  return xmlhttp;
}
var http = getHTTPObject();
</script>
</head>
<body>
<form action="post">
  <p>
  ZIP code:
  <input type="text" size="5" name="zip" id="zip" onblur="updateCityState();" />
  </p>
  City:
  <input type="text" name="city" id="city" />
  State:
  <input type="text" size="2" name="state" id="state" />
</form>
</body>
</html>
```

Fig. 15. AJAX address lookup form: HTML and JavaScript client code.

Despite the conceptual simplicity of the web form example, a large amount of code in multiple programming languages is required to build a robust AJAX application. In this case, the browser code (Fig. 15 is written in a combination of HTML and Javascript; the server code (Fig. 16) is written in PHP; SQL queries and data definitions are used to access the database (Figs. 16 and 17); and XML is used to transmit data from the server back to the browser (Figs. 16 and 15). Specialized Javascript libraries are also needed to dispatch the server request and process the XML response. The result is code that is complex and fragile, even for a very simple application.

By contrast, consider the reactor definitions in Figs. 18 and 19. While the reactor model is a foundational model, not a full-blown programming language, it is expressive enough to accurately model all of the essential operations present in the example. Here, reactors are used to model "widgets" representing single form fields (`InputWidgets`), the contents of of the page containing the main web form (`AjaxPage`), server side code (`Server`) and the zip code database (`ZipDB`). In

```
<?php
/**
 * Connects to the database.
 */
function db_connect() {
  $database_name = 'mysql';
  $database_username = 'root';
  $database_password = '';
  $result = mysql_pconnect('localhost',$database_username, $database_password);
  if (!$result) return false;
  if (!mysql_select_db($database_name)) return false;
  return $result;
}
$conn = db_connect(); // Connect to database
if ($conn) {
  $zipcode = $_GET['param']; // The parameter passed to us
  $query = "select * from zipcodes where zipcode = '$zipcode'";
  $result = mysql_query($query,$conn);
  $count = mysql_num_rows($result);
  if ($count > 0) {
    $city = mysql_result($result,0,'city');
        $state = mysql_result($result,0,'state');
  }
}
if (isset($city) && isset($state)) {
  // $return_value = $city . "," . $state;
  $return_value =
    '<?xml version="1.0" standalone="yes"?><zip><city>'.
      $city.'</city><state>'.$state.'</state></zip>';
}
else {
  $return_value = "invalid".",".$_GET['param']; // Include Zip for debugging purposes
}
header('Content-Type: text/xml');
echo $return_value; // This will become the response value for the XMLHttpRequest object
?>
```

Fig. 16. AJAX address lookup form: PHP server code.

```
CREATE TABLE `zipcodes` (
  `zipcode` mediumint(9) NOT NULL default '0',
  `city` tinytext NOT NULL,
  `state` char(2) NOT NULL default '',
  `areacode` smallint(6) NOT NULL default '0',
  PRIMARY KEY  (`zipcode`),
  UNIQUE KEY `zipcode_2` (`zipcode`),
  KEY `zipcode` (`zipcode`)
) TYPE=MyISAM;
```

Fig. 17. AJAX address loookup form: Database code.

addition, we also model the behavior of a rudimentary web browser (`Browser`). From this example, we see that reactors can compactly and uniformly model the synchronous interaction between the web page code and the browser, the asynchronous interaction between the browser and the server, the synchronous interaction between the server and the database, and the requisite logic used to connect these components together. Contrast the "three-tier" style of this example with the example in Fig. 10, which encapsulates its "business logic" in a single reactor.

Example 11 (*Three-Tier Web Application*). Fig. 20 depicts another web application which mimics the structure of a conventional three-tier application for catalog ordering. In a manner similar to the previous example, page content in browser is modeled by the `OrderPage` reactor type, which is instantiated with various primitive widget reactors—`OutputWidget` and `ButtonWidget` from Fig. 10 (in the case of `ButtonWidget`, the type of its `buttonListener` relation must be changed from `(ref DataDisplay)` to `(ref OrderPage)` to account for its new "parent" type) and a new `FormWidget`—depending on the type of page being displayed. Instances of `OrderPage` perform local (i.e., "browser-side") computation which performs basic form validation.

One limitation of our current model is that all reactor references must be strongly typed, hence the need to update the declaration of `ButtonWidget` to refer to a different "parent" reactor type, which makes it difficult to model a browser reactor that can render arbitrary pages represented as reactors. In the future, we will consider more flexible type system which would allow a completely generic browser, oblivious to the type of the page, to be defined.

```
def InputWidget = { // Reactor representing single input field in web form

  public size: (int).             // fixed field width, if specified
  public label: (string).         // field label
  public val: (string).           // value entered in field
  public blurListener:
    (ref Client).                 // listener(s) for blur event (change of focus)
}

def AjaxPage = { // Reactor representing composite web page for address lookup

  public rServer: (ref Server).   // ref. to server; initialized when page is created
  public rBrowser: (ref Browser). // ref. to browser; initialized when page
                                  //   received by browser

  title: (string).                // page title
  rZip: (ref InputWidget).        // widgets for zip, city, state
  rCity: (ref InputWidget).
  rState: (ref InputWidget).

  public ephemeral write          // positive response from server to lookup request
    zipFound: (string, string).
  public ephemeral write          // negative response from server to lookup request
    zipNotFound: (string).
  waiting: ().                    // flag set when zip lookup request in progress

  onBlur(_), not -waiting(): {    // when notified of blur event for zip widget and not
    s.getZip^(self,zs) <-         //   awaiting a prev. response, request city/state
      z.val(zs), rZip(z),         //   corresponding to zip
      rServer(s).
    waiting() <- .                // set flag while awaiting response
  }

  zipFound(cs,ss): {              // success case: copy city/state values returned
    c.val(cs) <- rCity(c).        //   to corresponding widgets and reset waiting
    s.val(ss) <- rState(s).       //   flag
    not waitingForZip() <- .
  }

  not waitingForZip() <-          // failure case (no city/state pair found): just
    zipNotFound(_).               //   reset waiting flag

  INIT: {                         // initialize new child widgets when parent page
    rZip(new) <- .                //   is created
    rZip(z) : {
      z.label("ZIP Code:") <- .
      z.size(5) <- .
      z.blurListener(self) <- .  // this reactor listens for blur events on zip widget
    }
    rCity(new) <- .
    c.label("City:") <- rCity(c).
    rState(new) <- .
    rState(s): { s.label("State:") <- . s.size(2) <- . }
    title("ZIP Code to City and State using the Reactor Model")
  }
}

def Browser = { // Reactor representing web browser

  currPage: (ref AjaxPage).       // current page in browser
  public ephemeral write          // response from getPage request
    showPage: (ref AjaxPage).

  currPage(p) := showPage(p).     // response from server to page request updates
  p.rBrowser(self) <-             //   current browser page and sets browser ref.
    showPage(p).                  //   in page to the browser
}
```

Fig. 18. Reactor-based address lookup form: "client"-related definitions.

Example 12 (*Small Workflow System*). Consider a workflow system (shown in Fig. 21) where employees handle incoming cases (say, problem reports) and attempt to resolve them. Regardless if a case is resolved or not, it is eventually archived to signify that no more work should be carried out on that case. Of course, it can be important to know the entire history of unresolved cases whether they are archived or not. To this end, the system outlined in Fig. 21 declares the relations

```
def Server = { // Reactor representing web server

  public rDB: (ref ZipDB).         // ref. to zip database (assumed initialized elsewhere)
  public ephemeral write           // corresponds to HTTP GET of main page
    getPage: (ref Browser).
  public ephemeral write           // correponds to HTTP GET for zip lookup
    getZip: (ref Browser).

  ephemeral newPage:               // temporary to hold newly-created main page
    (ref AjaxPage).
  getPage(b): {                    // respond to getPage request
    newPage(new) <- .              // instantiate new page
    newPage(p): {                  // initialize server ref. in new page; return
      p.rServer(self) <- .         //   page to browser
      b.showPage^(p) <- .
    }
  }

  ephemeral lookupRes:             // temporary to hold result of zip lookup in DB
    (string, string).
  getZip(b,zs): {                  // respond to getZip request
    lookupRes(cs,ss) <- d.addr(zs,cs,ss), rDB(d).
    b.zipFound^(cs,ss) <- lookupRes(cs,ss).
    b.zipNotFound^(zs) <- not lookupRes(_,_).
  }
 }

def ZipDB = { // Reactor representing zip code / address database

  public addr: (string, string, string, string).     // zip, city, state, area code

  FAIL <- addr(z,c1,_,_), addr(z,c2,_,_), c1 <> c2.  // rules which together assert that
  FAIL <- addr(z,_,s1,_), addr(z,_,s2,_), s1 <> s2.  //   zip is primary key
  FAIL <- addr(z,_,_,a1), addr(z,_,_,a2), a1 <> a2.
}
```

Fig. 19. Reactor-based address lookup form: ''server''-related definitions.

`inbox`, `unresolved`, and `archive`. Cases received from the outside world through the `inbox` are initially duplicated to `unresolved` and eventually put in `archive` although they may continue to exist in `unresolved`.

Incoming cases are allocated to employees who each has a work queue. Employees interact with the system through a GUI where they can mark cases resolved, archived or deallocated. These GUI events are handled through the public relation `GUIevent`.

In the normal workflow employees can navigate freely between the various states, as long as two rules are satisfied: (1) every case that has not been archived must be allocated, and (2) any resolved item should be archived (although it will stay in the work queue for post-review until the employee explicitly deallocates it). These two rules are encoded by the lines marked `(1)` and `(2)` in the example. The two rules demonstrate a desirable coding style, where the constraints are separate from the GUI handling and any other rules that may modify the relations; this improves compositionality with future rules.

Example 13 (*Aspect-like Modeling of Logging and Access Control*). The reactor code in Figs. 22 and 23 is based on an example due to Hankin et al. [18]. Fig. 22 depicts a simple application in which a `RequestMgr` reactor responds to requests by telephone company customers for billing information (via the ephemeral relation `getBill`). The billing information is derived by combining information in two databases: `WP`, a database of ''white pages'' listings, and `BillDB`, a billing database.

In Fig. 23, we augment the definition of `RequestMgr` in Fig. 22 with additional rules which add logging and access control functionality. The last two rules of the example encode assertions representing alternative access control policies. These rules have the effect of rolling back reactions which violate the policies. Hankin et al. [18] implement the basic billing functionality of Fig. 22 in the KLAIM process calculus [19], then add access control facilities using aspect-like [11] extensions to the core calculus. The aspect extensions can modify the behavior of the underlying application by inserting or removing expressions of the original program (in this case, by inserting code around database accesses which validates access control policies).

As with aspect-like approaches, the reactor model allows many ''additive'' behaviors, such as the new functionality of Fig. 23, to be incorporated into an existing program as additional rules, without requiring that existing code be modified. On the other hand, aspect languages typically provide meta-constructs to define ''pointcuts'' at which new behavior is introduced via syntactic pattern-matching; the reactor model provides no analogous functionality. Existing functionality in a reactor program can generally be altered or removed only by editing rules in place. By contrast, [18] allow ''advice'' to be defined which has the effect of conditionally eliminating existing behaviors. Nevertheless, we believe that the rule-based style of the reactor model usually allows conceptually independent behaviors to be specified as distinct collections of rules, which in turn means that the scope of required edits is minimized when application requirements change.

```
def DB = {  // Reactor representing a trivial database (list of items in inventory)

  public inv: (int).
}

def Server = { // Reactor representing web server

  rDB                   : (ref DB).                    // reference to the database
  public ephemeral write getPage : (ref Browser).      // new page request
  public ephemeral write postForm : (ref Browser, int). // form submission request

  ephemeral newPage     : (ref OrderPage).             // temp to hold new page
  newPage(new)          <- .                           // generate pg. on every reaction;
  c.rServer(self)       <- newPage(p).                 //   each has link to server

  getPage(br), newPage(p): {                           // getPage creates three
    p.outWidget(new),                                  //   primitive widgets for the
    p.formWidget(new),                                 //   page
    p.buttonWidget(new) <- .

    o.label("Available: "), o.val(toString(q))         // init widget data; in particular
                <- c.outWidget(o), rDB(d), d.inv(q). //   copy current inv. from db
    f.label("Quantity to order: "), f.val("1")  <-  p.formWidget(f).
    b.label("Submit"), b.buttonListener(p)  <-  p.buttonWidget(b).
    br.showPage^(p)     <- .                           // submit page to browser
  }

  ephemeral reqOK : ().
  postForm(br, qr), newPage(p): {                      // postForm checks whether
    p.outWidget(new)    <- .                           //   requested qty. is avail.;
    reqOK()             <- qr >= q, rDB(d), d.-inv(q). //   returns appropriate responses
    o.label("Success!") <- reqOK(), p.outWidget(o).
    d.inv(q-qr)         := reqOK(), rDB(d), d.-inv(q).
    o.label("Sorry!")   <- not reqOK(), p.outWidget(o).
    br.showPage^(p)     <- .                           // submit page to browser
  }
}

def Browser = { // Reactor representing web browser

  currPage: (ref OrderPage).                           // current page in browser
  public ephemeral write                               // response from getPage request
    showPage: (ref OrderPage).

  currPage(p) := showPage(p).                          // response from server to page
  p.rBrowser(self) <-                                  //   request updates curr. browser
    showPage(p).                                       //   page and sets browser ref.
                                                       //   in page
}

def OrderPage = { // Reactor representing composite page for order submission
  public rServer        : (ref Server).                // ref to server
  public rBrowser       : (ref Browser).               // ref to browser
  outWidget             : (ref OutputWidget).          // widgets on page; the form and
  formWidget            : (ref InputWidget).           //   button widgets are empty (not
  buttonWidget          : (ref ButtonWidget).          //   used) on a response page
  public ephemeral write onButton
                        : (ref ButtonWidget).          // set when button pressed

  ephemeral validateOK  : ()                           // perform local validation
  validateOK()          <- buttonWidget(b), onButton(b), formWidget(f), f.val(qty),
                           outWidget(o), o.val(inv), toInt(qty) <= toInt(inv).
  rServer.postForm^(br, toInt(qty))                    // validation OK: submit to server
                        <- validateOK(), rBrowser(br), formWidget(f), f.val(qty).
  f.val("1")            <- not validateOK().           // validation fails: just reset
                                                       //   qty. to 1; do not submit
}
```

Fig. 20. Mini three-tier web application.

Example 14 (*Refactoring from Synchronous to Asynchronous Interaction*). The examples in Figs. 24 and 25 both implement the functionality of the example in Fig. 8, which we considered earlier. Reactor `SyncSample` below is an entirely synchronous variant of the sensor sampling reactor `Sample` of Fig. 8. Reactor `AsyncSample` of Fig. 25 is rather more interesting. Like `Sample`, it is asynchronous. However, `AsyncSample` does not require that either the sensor or the sampling

```
def WorkflowSystem = {
// Workflow system for case handling

public ephemeral inbox    : (string)           // (case)
public ephemeral GUIevent : (string, string,   // (case, employee,
                             string)           //  {"Resolve","Archive","Dealloc"})

employee                  : (string)           // (employee)
workQueue                 : (string, string)   // (case, employee)
archive                   : (string)           // (case)
unresolved                : (string)           // (case)


// Normal workflow

unresolved^(case)   <-  inbox (case)                              // Put in unresolved
workQueue^ (case,p) <-  not archive (case), unresolved (case),    // (1) Allocate
                        employee (p)
archive^ (case)     <-  workQueue (case,p), not unresolved (case) // (2) Archive resolved


// GUI events

not unresolved (case)  <-  GUIevent (case, _, "Resolve")          // Case resolved
archive (case)         <-  GUIevent (case, _, "Archive")          // Archive case
not workQueue (case,p) <-  GUIevent (case, p, "Dealloc")          // Remove from queue
}
```

Fig. 21. Example from a workflow system for case handling.

```
def Cust = { // Reactor representing a customer

  ...                              // customer name, address, etc.
  public ephemeral write           // response to request for billling info.
    bill: (int).
}

def WP = { // Reactor representing "white pages" database

  public dir: (ref User, string).  // customer, phone number
}

def BillDB = { // Reactor representing billing information

  public billing:                  // phone number, cost of call, customer
    (string, int, ref User).
}

def RequestMgr = { // Reactor managing requests to phone-related databases

  rWP: (ref WP).                   // refs. to white pages and billing
  rDB: (ref BillDB).               //   databases; assumed initialized elsewhere

  auth: (ref Cust, ref BillDB).    // authorization matrix: auth(c, d) means
                                   //   that customer c is authorized to
                                   //   access database d

  public ephemeral write getBill:  // getBill(r, c): requester r requests
    (ref Cust, ref Cust).          //   billing data for customer c
  r.bill^(a) <-
    getBill(r, c), y.dir(c, n), d.billing(n, a, c), rWP(y), rDB(d).
}
```

Fig. 22. Telephone records management system.

reactor handle asynchronous requests and responses. Instead, we use an auxiliary "agent" reactor, `SampleAgent`, which first reacts synchronously with `SyncSensor`, then (in a separate reaction) reacts synchronously with `AsyncSample`. Using such auxiliary agents, we can refactor synchronous reactor interaction to asynchronous interaction with minimal change to the original reactor code.

Example 15 (*Business Process for Order Fulfillment*). The example in Fig. 26 depicts a simple business process example modeling order fulfillment. In this example, a `Gateway` reactor processes requests to fulfill a certain quantity of items (in

```
def RequestMgr = { // Reactor managing requests to phone-related databases

  rWP: (ref WP).
  rDB: (ref BillDB).
  auth: (ref Cust, ref BillDB).

  public ephemeral write getBill: (ref Cust, ref Cust).
  r.bill^(a) <-
    getBill(r, c), y.dir(c, n), d.billing(n, a, c), rWP(y), rDB(d).

  log: (ref Cust, ref Cust)        // log accesses
  log(r, c) <- getBill(r, c).

  FAIL <- getBill(r, c),           // ...and/or disallow unauthorized access
    not auth(r, d), rDB(d).

  FAIL <- getBill(r, c), r <> c,   // ...or use a more sophisticated policy: requester
    not auth(r, d), rDB(d).        //   is allowed to request information about
                                   //   self; otherwise, must be explicitly  authorized
}
```

Fig. 23. Telephone records management system augmented with logging/authorization functionality.

```
def SyncSensor = {
  public val: (int)
}

def SampleData = {
  // data collected to date; nonce used to
  // distinguish multiple measurements of same
value
  public log: (ref Nonce, int).
}
```

```
def SyncSample = {
  public rSensor: (ref SyncSensor).
  public rSampleData: (ref SampleData).
  public write ephemeral pulse: ().

  // add current value of sensor to log
  c.log(new, v) <-
    pulse(), rSensor(s),
    rSampleData(c), s.val(v).
}
```

Fig. 24. Fully synchronous sensor sampling.

```
def AsyncSample = {
  public rSensor: (ref SyncSensor).
  public rSampleData: (ref SampleData).
  public write ephemeral pulse: ().
  ephemeral newAgt:(ref SampleAgent).

  // delegates functionality to reactor
  // of type SampleAgent
  pulse(): {
    newAgt(new) <- .
    a.rSensor(s), a.rSampleData(c) <-
      rSensor(s), rSampleData(c).
  }
}
```

```
def SampleAgent = {
  // implements state machine that first
  // gets sample, then writes to log
  public rSensor: (ref SyncSensor).
  public rSampleData: (ref SampleData).
  ephemeral readSample: ().
  ephemeral writeLog: (int).

  // sends an async update to itself
  // when created
  getSample^() <- not -live().

  // reads sensor, sends async update
  // to self with the value read
  writeLog^(v) <-
    getSample(), rSensor(s), s.val(v).

  // finally, writes the sensor value to log
  c.log(new, v) <-
    writeLog(v), rSampleData(c).
}
```

Fig. 25. Fully asynchronous sensor sampling.

this example, only one type of item is considered, for simplicity). The `Gateway` maintains an ordered list of references to `Warehouse` reactors which are capable of fulfilling the order. When a request is received, a new `Processor` reactor is spawned specifically to fulfill the (single) request. The `Processor` reactor asynchronously queries each warehouse on the list to determine if it can fulfill the order itself; if not, the remainder of the order is passed on to the next warehouse on the list for fulfillment, and so on. If all of the requested items can be supplied, an asynchronous response is sent to the requester, indicating success. Otherwise, a failure response, containing the number of items unfulfilled, is dispatched. The `FIRST` and `EMPTY` expressions are syntactic for rules and auxiliary relations which compute the first element of a list, and test a list for emptiness, respectively.

```
def Gateway = {
  // requests are pairs of the form
  // (requester, quantity requested)
  public ephemeral request:
    (ref Requester, int).

  // ordered list of warehouse refs which may
  // be used to fulfill requests;
  // lists are sets of tuples of the form
  // { <l1, l2>,<l2, l3>,...<ln, ln> }
  public whs:
    (ref Warehouse, ref Warehouse).

  // auxiliary reactor spawned to
  // process each request
  ephemeral processor: (ref Processor).

  // spawn reactor for request
  processor(new) <- request(_,_).

  // initialize persistent relations
  // of request processor
  p.whs(w1, w2),
  p.requester(r),
  p.remaining(q) <-
    whs(w1, w2), request(r, q).
}
```

```
def Processor = {
  // requester
  public requester: (ref Requester).
  // quantity of items left to fulfill
  public remaining: (int).
  // warehouses ramining to be queried
  public whs: (ref Warehouse, ref Warehouse).
  // quantity of items last warehouse
  // queried is willing to supply
  public ephemeral write willSupply: (int).

  // the following rules apply both when the
  // processor is initialized and after a
  // warehouse query returns
  remaining(q): {
    // quantity left to fulfill is nonzero:
    // query first warehouse on list
    w.request^(q) <-
      q > 0, FIRST[whs(_,_)](w).
    // qty. left to fullfill zero:
    // respond positively
    r.complete^() <- q = 0.
    // no warehouses left, qty. remaining
    // nonzero, respond with unfullfilled amt.
    r.incomplete^(q) <-
      q > 0, EMPTY[whs(_,_)].
  }

  willSupply(qw): {
    // decrement quantity supplied from
    // amount remaining to fulfill
    // (qty. to be supplied is assumed less
    // than or equal to qty. requested)
    remaining(q - qw) := -remaining(q).
    // remove first warehouse from list
    // when response received
    not whs(w1, w2) <- -whs(w1, w2),
      FIRST[-whs(_,_)](w1).
  }
}
```

Fig. 26. Order fulfillment workflow.

This example in Fig. 26 illustrates how auxiliary reactors can be instantiated that carry out independent "task threads", in a manner similar to process replication in standard process calculi. The example also illustrates how the data manipulation components of the language (here, e.g., the machinery used to manage lists of warehouses) interact naturally with process creation and inter-process communication.

Example 16 (*Views Defined by Rules: Shopping Cart Management*). The example in Fig. 27 models shopping cart management for a catalog order application. Reactor type `CartManager` is intended to manage a shopping cart for a single user interaction ("session"). Each `CartManager` instance contains a "public" view of private cart data maintained on behalf of *all* users by reactor `DB`. Relation `currCart` contains the public contents of a single user cart, and `allCarts` maintains the private contents of all shopping carts currently managed in the system, along with auxiliary information about users and shipping information for each cart. The client interacts with the public cart (`currCart`) by reading and writing its contents directly. The rules of `CartManager` synchronize the public contents of `currCart` with private internal data; hence `currCart` functions as a *view*, in the database sense, of the private `DB` data. Note that the shipping information is maintained by `DB`, rather than `CartManager` instances. The `CartManager` rules are concerned only with synchronizing the public and private cart data. As before, `EMPTY` is syntactic sugar for auxiliary rules/relations which test a relation for emptiness. `SUM` is sugar for rules/relations which compute the sum of the quantity values for each item in a cart.

The cart synchronization rules must account for various edge conditions: e.g., when there is a pre-existing private cart that can be used to populate the public cart, and when there is no pre-existing cart at all. This is quite straightforward to do by composing rules that separately handle each edge case, but would likely require much more thought and care if written in a non-declarative style. Similarly, the rule-based computation allows the distinct "concerns" of cart synchronization and shipping logic to be specified and managed independently, in an aspect-like manner.

12. Related work

Fundamentally, reactors are "reactive systems" [20], combining and extending features from several, largely unrelated areas of research: synchronous languages, Datalog [4], and the actor model [5].

```
def CartManager = {
  // singleton relation containing current
userid
  public userid: (string).
  // public "view" of cart for clients
  public currCart: (int, int)
  // reference to persistent data
  rDB: (ref DB).

  userid(u), rDB(db), db.userCarts(u, n): {
    // when cart manager first created,
    // initialize currCart with allCarts
    // entry for user, if any exists
    currCart(i, q) <-
      INIT, -db.allCarts(n, i, q).

    // if no cart exists, create one
    ephemeral preExistingCart: (string).
    preExistingCart(u) <- -db.userCarts(u, _).
    db.userCarts(u, new) <-
      EMPTY[preExistingCart(u)].

    // all currCart entries must
    // also be in allCarts
    db.allCarts(n, i, q) <-
      currCart(i, q), db.allCarts
    // if qty. changed, remove old qty.
    not db.allCarts(n, i, q) <-
      currCart(i, q'),
      -db.allCarts(n, i, q), q <> q'.
    // item not in currCart:
    // delete from allCarts
    not db.allCarts(n, i, _) <-
      not currCart(i, _).
  }
}
```

```
def DB = {
  // maps userids to cartids
  userCarts: (string, ref Nonce).
  // maps cartids to items, qty
  allCarts: (ref Nonce, int, int).
  // carts with standard shipping
  standardShipping: (ref Nonce).
  // carts with free shipping
  freeShipping: (ref Nonce).

  // std. shipping if qty. < 10, free o.w.
  standardShipping(n) <-
    SUM[n, q, allCarts(n, _, q)](n,s),
    s <= 10.
  freeShipping(n) <-
    SUM[n, q, allCarts(n, _, q)](n,s),
    s > 10.
}
```

Fig. 27. Shopping cart management.

Esterel [1], Lustre [2], Signal [3], and Argos [21] 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 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 [22] 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 [23] 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. We feel this increases the opportunity for aggressive parallelization. Another

piece of work on Datalog with updates is DatalogU [24], which follow a similar approach to Transaction Datalog, but without transactions. DatalogU proposes a semantics in which deletions are performed before insertions so that newly inserted facts cannot be deleted immediately. The reactor model augments the features found in these Datalog variants with a framework for distribution, communication, and generativity (the ability to create new reactors), and allows deletion to happen concurrently with the rest of the computation.

In [25] an approach to deletion is presented which focuses only on modifying the extensional database. The semantics of rules with updates in the head is given by re-writing them into equivalent update-free rules that are XY-stratified—a special subcase of locally stratified programs. These programs can express all programs under inflationary fixpoint semantics. Deletions (and additions) in this approach occur as additional commands, separate from the program, which request the removal (or addition) of extensional facts. In this context, a deletion request is handled by simply not copying a fact to the database.

U-Datalog [26] has an non-immediate update semantics, in which updates are collected and applied in parallel to the database when the query evaluation is completed. This approach lacks what we think of a nice property that when a reaction quiesces all the rules have been re-satisfied.

In [27], rules which update the database by retracting tuples are regarded as integrity constraints, and programs with rules containing retractions are translated into normal programs. Consistency is restored by making one of the positive literals in the rule body false. This can be updates to the database, database initial facts, or the facts inferred from the rules. The semantic is nondeterministic.

In [28] procedural and declarative update languages are proposed which all have *saturation* semantics—a form of inflationary semantics. This means that facts can be added to the database, but never deleted. [29] defines Datalog-A, an extension of Datalog with updates; this approach promotes explicitly procedural constructs and uses a Dynamic Logic to specify their semantics.

The constructive semantics of LDL [30] for Datalog programs with updates is top-down. The declarative semantics of LDL programs imposes an *update dependency* restriction on programs. If p is the head predicate symbol of a rule s.t. the definition of p contains an update predicate (addition or deletion) p has the *update dependency property*. A set of rules is legal if every head predicate symbol that has the update dependency property is defined by a single rule. In the case in which the order of applying the updates is relevant, the program is rejected or the user can specify procedurally the order of execution.

KLAIM [31] is a process calculus type of programming language based on Linda. It has asynchronous communication based on creating tuples which will be received by any number of processes. There is a shared data space, though multiple located tuple spaces can exist at a time. The language is procedural in the sense that it expresses implementation as opposed to specifications, and operations are ordered.

Orc [32] is an orchestration language for distributed computation. The composition operators for services are parallel composition, sequencing, and selective pruning. The model does not have any computational power; it relies on services to provide this capability. The basic semantics is asynchronous, but it also provides a particular flavor of synchronous semantics which ensures that all internal events are processed as soon as possible, before any external response can be processed.

TCC [33] presents a declarative approach that starts from an asynchronous computational model (CCP) and ensures a timed computational model in which computation is deterministic and time is bounded. The language supports hierarchical and modular construction of applications via parallel composition and pre-emption. To express time-out TCC introduces the notion of negative information. The absence of some positive information is interpreted as negative information solely at a point where the computation has quiesced, and can only trigger more activity in the next computation interval.

Aspect-oriented programming [11] addresses the separation and composition of orthogonal, but interacting, concerns of a program. Reactors do not have the features usually identified with aspects (point-cuts, advice, etc.), but instead provide one general mechanism for composition of rules in a non-intrusive and well-defined manner.

Other approaches to Internet application programming include Flapjax [34] (an event-stream and message library built on top of JavaScript), and Links [35] (a code generation approach, where the code for several tiers is generated from one source).

What differentiates us from the existing approaches beyond the technical details is a uniform programming model which synthesizes concepts from several domains without the usual complexity associated with this process. The semantic and syntactic machinery for capturing all these features in a single programming model mostly consists in minimal extensions to the standard Datalog semantics.

13. 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., [36]), 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., [37]) 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.

Our semantic account of the reactor model has treated a collection of reactor types as defining an augmented program, whose evaluation is orchestrated by the rules defined in Section 9. However, it is neither necessary nor desirable in a practical system to require that the set of all reactor types be translated into a single program. It it only necessary that each reactor be aware of the other reactors that it interacts with within a given reaction. A truly distributed implementation will build the stratification graph and evaluate the rules incrementally. The following issues would need to be addressed in a practical distributed implementation of the reactor model. Given an initiating reactor which selects an update bundle off the network, we would like it to be the case that:

- The initiating reactor is responsible for defining the scope of the reaction by including all, and only, the reactor types reachable from it via writing a remote relation, or instantiating a new reactor type. The augmentation transformation only applies to the set of reactor types in the reaction scope. Likewise, the transformation which removes head-clause negation applies to the set of augmented reactor types in the reaction scope instead of the augmented program reflecting the set of all reactor types. We must extend the definition of head-negation-stratified and stratified programs to account for sets of reactor types rather than just augmented programs.
- The initiating reactor is responsible for the evaluation of all, and only, the rules in the reaction scope. This distributed evaluator evaluates rules in the same order as the evaluator presented in Section 5.5. Also, no rule in the distributed evaluator is executed until all instances of tuples for the rule's relation instances have been ''loaded'' into the evaluator's relations.

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] G. Berry, G. Gonthier, The ESTEREL synchronous programming language: Design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87–152.
[2] P. Caspi, D. Pilaud, N. Halbwachs, J.A. Plaice, LUSTRE: A declarative language for real-time programming, in: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL'87, ACM Press, New York, NY, USA, 1987, pp. 178–188.
[3] T. Gautier, P.L. Guernic, SIGNAL: A declarative language for synchronous programming of real-time systems, in: Proceedings of the Functional Programming Languages and Computer Architecture, Springer-Verlag, London, UK, 1987, pp. 257–277.
[4] J.D. Ullman, Principles of Database and Knowledge-Base Systems, vol. 1, Computer Science Press, 1988 (Chapter 3).
[5] G.A. Agha, I.A. Mason, S.F. Smith, C.L. Talcott, A foundation for actor computation, Journal of Functional Programming 7 (1) (1997) 1–69.
[6] R. Milner, J. Parrow, D. Walker, A calculus of mobile processes, parts I–II, Information and Computation 100 (1) (1992) 1–77.
[7] R.T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. Thesis, University of California, Irvine, 2000.
[8] J. Field, M.-C.V. Marinescu, C. Stefansen, Reactors: A data-oriented synchronous/asynchronous programming model for distributed applications, in: Ninth Intl. Conf. on Coordination Models and Languages, in: Lecture Notes in Computer Science, vol. 4467, Springer-Verlag, Paphos, Cyprus, 2007, pp. 76–95.
[9] R. Topor, Safe database queries with arithmetic relations, in: Proceedings of the 14th Australian Computer Science Conference, 1991.
[10] X. Wang, Negation in logic and deductive databases, Ph.D. Thesis, University of Leeds, 1999.
[11] G. Kiczales, Aspect-oriented programming, ACM Computing Surveys 28 (4es) (1996) 154.
[12] T. Erl, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, 2005.
[13] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison Wesley, 1995.
[14] A.V. Gelder, Negation as failure using tight derivations for general logic programs, in: Proc. Symp. on Logic Programming, 1986, pp. 127–139.
[15] A.W.K.R. Apt, H.A. Blair, Towards a theory of declarative knowledge, in: Foundations of Dedective Databases and Logic Programming, Morgan Kaufmann, 1988, pp. 89–148.
[16] S. Naqvi, Negation as failure for first-order queries, in: Proc. Fifth ACM Symposium on Principles of Database Systems, 1986, pp. 114–122.
[17] B. Bercik, Guide to using ajax and xmlhttprequest, 2005. URL: http://www.webpasties.com/xmlHttpRequest/index.html.
[18] C. Hankin, F. Nielson, H.R. Nielson, F. Yang, Advice for coordination, in: Proc. Intl. Conf. on Coordination Models and Languages, in: LNCS, vol. 5052, Springer-Verlag, 2008, pp. 153–168.
[19] R. de Nicola, G.L. Ferrari, R. Pugliese, Klaim: A kernel language for agents interaction and mobility, IEEE Transactions on Software Engineering 24 (5) (1998) 315–330.
[20] D. Harel, A. Pnueli, On the development of reactive systems, in: K.R. Apt (Ed.), Logics and Models of Concurrent Systems, in: NATO ASI Series F: Computer and Systems Sciences, vol. 13, Springer-Verlag New York, Inc., New York, NY, USA, 1989, pp. 477–498.
[21] F. Maraninchi, Y. Rémond, Argos: An automaton-based synchronous language, Computer Languages 27 (2001) 61–92.
[22] N.W. Paton, O. Díaz, Active database systems, ACM Computing Surveys 31 (1) (1999) 63–103.
[23] A.J. Bonner, Workflow, transactions and datalog, in: Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'99, ACM Press, New York, NY, USA, 1999, pp. 294–305.
[24] M. Liu, Extending datalog with declarative updates, Journal of Intelligent Information Systems 19 (2002) 1–23. URL: citeseer.ist.psu.edu/liu02extending.html.
[25] C. Zaniolo, On the unification of active databases and deductive databases, in: British National Conference on Databases, 1993.
[26] M.M.D. Montesi, E. Bertino, Transactions and updates in deductive databases, in: Knowledge and Data Engineering, 1995.
[27] L. Raschid, J. Lobo, Semantics for update rule programs and implementation in a relational database management system, in: ACM Transactions on Database Systems, vol. 22, 1996, pp. 526–571.
[28] S. Abiteboul, V. Vianu, Procedural and declarative database update languages, in: Proceedings of the Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database System, PODS'88, 1988.
[29] S. Naqvi, R. Krishnamurthy, Database updates in logic programming, in: Proceedings of the Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'88, ACM Press, New York, NY, USA, 1988, pp. 251–262.
[30] S. Naqvi, S. Tsur, A Logical Language for Data and Knowledge Bases, Computer Science Press, 1989.

[31] R.D. Nicola, M. Loreti, A modal logic for klaim, in: Algebraic Methodology and Software Technology, 1998.
[32] J. Misra, W. Cook, Computation orchestration, in: Software and Systems Modeling, 2006.
[33] V. Saraswat, R. Jagadeesan, V. Gupta, Foundations of timed concurrent constraint programming, in: 9th Annual IEEE Symposium on Logic in Computer Science, 1994.
[34] S. Krishnamurthi, Flapjax, November 2006. URL: http://www.flapjax-lang.org/.
[35] E. Cooper, S. Lindley, P. Wadler, J. Yallop, Links: Web programming without tiers, in: 5th International Symposium on Formal Methods for Components and Objects (FMCO), Springer-Verlag, 2006, pages 10.
[36] G. Dong, R.W. Topor, Incremental evaluation of datalog queries, in: Database Theory — ICDT'92, 4th International Conference, Germany, 1992, Proceedings, in: LNCS, vol. 646, Springer, 1992, pp. 282–296.
[37] T. Harris, S. Marlow, S.P. Jones, M. Herlihy, Composable memory transactions, in: ACM Conf. on Principles and Practice of Parallel Programming, Chicago, 2005, pp. 48–60.