Open Constraint Programming

Joxan Jaffar, Limsoon Wong and Roland H.C. Yap

Department of Computer Science
School of Computing
National University of Singapore
Republic of Singapore 119260
`Contact: joxan@comp.nus.edu.sg`

23:59:59 Friday 12 Nov 1999 PST

A new model for high-level concurrent programming is presented. Designed for implementing programs which manipulate and react to shared data which is highly structured, its main elements are:

- a shared constraint store, defined as a theory-update system
- a notion of *reactor*, which is a program fragment written in a generic reactive programming language;
- a general use of agents, defined as arbitrary (possibly parallel) programs, which contain embedded reactors;
- a *trigger model*, representing an abstract architecture addressing the key implementation challenges.

This paper presents a "wrapper" framework as the outermost framework toward the definition and implementation of constraint stores and a sufficiently rich set of agents. A constraint store is formalized as in traditional constraint programming as a theory with an intended model. These theories are updatable, and the update functions are available externally via reactive programs. These programs, called reactors, are used to update and synchronize with the store, and are embedded within user programs. In summary, the entire framework provides high-level reasoning on the domain of discourse via its constraint system, provides expressive synchronization and update operations via updates to the constraints, and finally, it provides flexibility via the embedding of reactors in arbitrary agent programs, ie: it is an *open* system.

1 Introduction

A new model, Open Constraint Programming (OCP), for high-level concurrent programming with a shared store is presented. The model is a general one which is designed for processes/agents which need to react and coordinate using highly structured shared data. The main elements are:

- a shared constraint store, defined as an updatable theory;
- a notion of *reactor*, a program fragment written in a generic reactive programming language, and serves to update and synchronize with the store;
- a general use of agents, defined as arbitrary (possibly parallel) programs, which contain embedded reactors and these may come and go at anytime;
- an abstract system architecture to guide implementation, consisting of a constraint solver over the theory, a *trigger model*, facilitating a conservative estimate of the set of reactors (and hence agents) affected by changes to the store, and a reactor registry which coordinates and schedules the execution of reactors.

The programming paradigm addressed here is one of shared computation where many agents need to share data in a timely fashion with synchronization where appropriate. Such a paradigm is used in parallel programming, distributed programming, client-server programming, network programming, etc. While communication based approaches is one way of achieving such programs, in this paper we concentrate on approaches to shared computation using a common (constraint) store with particular emphasis on reactive programming and open systems.

Communication using a shared store is not new. At one end of the spectrum, there are primitive unstructured mechanisms such as shared memory which provides for a common shared array of words, this is exemplified by systems such as Threadmarks [3] which add distributed memory and synchronization constructs. In the middle ground, with semi-structured stores, there are the blackboard architectures well exemplified by coordination languages such as Linda [5]. The other end with highly structured stores is best exemplified by concurrent constraint programming (CCP) [15]. The OCP model is meant to be a common basis for reactive programming across many instances from this spectrum though its strengths are best realized by more structured stores.

We will focus on the CCP model as the closest comparison. CCP provides a concurrent programming language in a logic programming style where reduction of rules are interpreted as processes in the context of a global constraint store. The two basic constraint operations are *ask* and *tell* of constraints. A tell operation monotonically augments the store with a new constraint. An ask achieves synchronization by waiting until a constraint to be entailed by the store. The strengths of constraint programming (CP) such as in CCP (and also CLP [8]) is that the constraint store is not an explicit representation and doesn't mere store data, but rather computes a "closure" of the information so far [13]. Thus CP exploits the fact that the constraint store implicitly represents more information than is put in. In addition, the ask in CCP provides a form of declarative synchronization. The Oz [17] language is another concurrent CP language in a functional style with ask and tell as in CCP and constraints over trees and finite domains. A notable feature of Oz is that it provides for sub-constraint stores (computation spaces) as first class objects and this serves as the basis of encapsulated search.

Traditional constraint programming (CP) [13] need not be based on a specific constraint programming language. The ILOG solver constraint library [14] exemplifies CP in conventional programming languages by providing a C++ library which is a constraint solver for finite domain constraints together with a software architecture for writing search in C++.

We propose that high-level programming of reactive behavior, such as that in CCP, where concurrent agents can share knowledge and synchronize using information in the store is a highly useful programming paradigm given networks and distributed computational resources. OCP addresses some limitations of CCP:

- Entailment is defined on a monotonic store, and often the underlying store represents some logical state. A direct representation of state and a notion of persistence is therefore problematic.
- The underlying programming language, guarded rules equipped with a committed-choice operation, which is used to define the operational behavior of the agents, deals both with application specific part of the agent as well as the *reactive* part. For example, the fine granularity of ensuing processes is often inadvertently obtained. In fact, it is often desirable to have no tie-in to a particular and non-standard agent programming language, and instead have embeddable features for working concurrently with the store.
- CCP defines a closed system where agents are created by other running agents. The environment of OCP is aimed at an open system where the creation, introduction and termination of agents are external to the store but these still interact with the store.
- CCP only provides for atomic *ask* and *tell* operations. OCP incorporates more general notions of atomicity and transactions including those from databases.

A pragmatic approach to implementing a common store is to have remote procedure calls in agents which are executed by the store, and these directly provide both update and synchronization facilities. (This is analagous, in the sequential setting, to using ILOG solver for CP.) The problems with this approach are: (a) it has to provide an expressive enough set of RPC's, and (b) deal with concurrency control and the related problems involved in actually getting the calls executed. In general, a constraint store is a large, established and fixed system. Adapting one for use across a diverse set of agents for different special purposes is impractical.

This paper presents a "wrapper" framework as the outermost framework toward the definition and implementation of constraint stores and a sufficiently rich set of agents. The constraint stores are formalized as in traditional constraint programming as a theory with an intended model. The theories are updatable, and the update functions are available externally via reactive programs. These programs, called reactors, are used to update and synchronize with the store, and are embedded within user programs. In summary, the entire framework provides high-level reasoning on the domain of discourse via its constraint system, provides expressive synchronization and update operations via updates to the constraints, and finally, it provides flexibility via the embedding of reactors in arbitrary agent programs, ie: it is an *open* system.

2 The Model

In what follows, we describe the two main components that comprise an OCP system: the constraint store, and reactors.

2.1 Constraint Store

The starting point for an OCP system is the notion of a constraint store, representing shared memory for agents. Intuitively, a constraint store is defined wrt to a fixed first order language and base theory $\mathcal{T}_0$, and an intended interpretation of this theory. The store itself is simply an open formula $\mathcal{C}$ in the language considered, and its logical meaning is simply that given by the intended interpretation of the closed formula $\mathcal{T}_0 \wedge \tilde{\exists}\mathcal{C}$, where $\tilde{\exists}\mathcal{C}$ denotes the existential closure of $\mathcal{C}$. Operations on a constraint store are defined simply as transitions, and should be thought of as theory updates.

For example, using an intended interpretation of linear integer arithmetic on an appropriate base theory of arithmetic, consider the store

$$
\begin{array}{l}
1 \le X_1 \le 4 \wedge 1 \le X_2 \le 4 \wedge 1 \le X_3 \le 4 \wedge 1 \le X_4 \le 4 \wedge \\
X_1 \ne X_2 \wedge X_1 \ne X_3 \wedge X_1 \ne X_4 \wedge X_2 \ne X_3 \wedge X_2 \ne X_4 \wedge X_3 \ne X_4 \wedge \\
X_1 - X_2 \ne 1 \wedge X_2 - X_1 \ne 1 \wedge \\
X_1 - X_3 \ne 1 \wedge X_3 - X_1 \ne 2 \wedge \ \cdots
\end{array}
$$

which can be used to represent constraints on a four-queen problem over the variables $X_1, \ldots, X_4$. An example store update is the conjoining of this formula with $X_1 = 1$, which intuitively represents placing the first queen on the first row (or column). A subsequent update conjoining $X_2 = 1$ would lead to an inconsistent store, unlike conjoining $X_2 = 4$. More formally:

Definition A *constraint system* Δ is defined over a first-order language $\mathcal{L}$, and a (finite or infinite) set of *decision variables* $\tilde{X}$.

- an intended interpretation $\mathcal{M}$ for $\mathcal{L}$;
- a universe $\{\Delta_1, \Delta_2, \ldots\} \cup \{\perp\}$ of *constraint stores* where each Δ_i is an open formula over $\mathcal{L}$ and $\tilde{X}$ which is *satisfiable* in $\mathcal{M}$, ie. $\mathcal{M} \models \tilde{\exists}\Delta$. $\perp$ is a special formula representing inconsistent stores.
- a set of *updates* $\{\delta_1, \delta_2, \ldots\}$.
  Each update δ_i is such that $\delta_i(\Delta) = (\Delta', \mathcal{C})$
  where Δ and Δ' are constraint stores, and $\mathcal{C}$ an $\mathcal{L}$-formula. Intuitively, δ_i maps Δ into the new store Δ', and $\mathcal{C}$ is the *result* of the update. The result $\mathcal{C}$ is defined via a predefined function $f_{\mathcal{C}}$ on stores. Thus $f_{\mathcal{C}}$ is a description of some specific property of Δ, and often this is a boolean property in which case $\mathcal{C}$ is either *true* or *false*. Further,
  (a) if $\Delta' \ne \perp$, $\tilde{\forall}\tilde{X}.\Delta \longrightarrow f_{\mathcal{C}}(\Delta)$ is true in $\mathcal{M}$;
  (b) if Δ is $\perp$, then $\Delta' = \perp$. $\Box$

We henceforth abuse the notation Δ to denote name of a constraint system, as well as one of its states. This is primarily because we often refer to a store only by referring to its current state. We write $\Delta \xrightarrow{\delta} \Delta'$ to denote an update via δ; we may also write $\mathcal{C} = \Delta \xrightarrow{\delta} \Delta'$ to indicate the return $\mathcal{C}$ of the update. We say that this update is *passive* if $\Delta' = \Delta$. We say that it is *successful* if $\Delta' \ne \perp$; alternatively, δ is *enabled* for Δ. otherwise it is *failed* or not enabled.

Example (arithmetic) The constraints of interest are conjunctions of linear real arithmetic atomic formulas $\mathcal{C}$ open on one decision variable X. An example formula is $2 \le X \wedge X \le 7$. The meaning of a store Δ is characterized by the polytope corresponding to the solution space of the conjunction in Δ. Some example updates:

- tell($\mathcal{C}$)
  Eg. tell($X = 45$): this transition fails if $\Delta \wedge X = 45$ is inconsistent, or else Δ' represents the store $\Delta \wedge X = 45$. The return value in the latter case is *true*.
- softell($\mathcal{C}$)
  Eg. softell($X = 45$): this transition is passive with return value *false* if $\Delta \wedge X = 45$ is inconsistent; otherwise, Δ' is the store $\Delta \wedge X = 45$ and the return value is *true*.
- ask($\mathcal{C}$)
  Eg. ask($X \le 3$): this update is passive; return value is *true* if $\Delta \models X \le 3$, and *false* otherwise. Note that there is no notion of delaying here.
- is_ground(X):
  This update is passive; it returns $X = n$ if $\Delta \models X = n$ for some number n; it returns *false* otherwise.

Example (Constraint Database) We use a PROLOG program P with ground facts [12] here, though any CLP program, treated as a constraint database [10], could be used by a straightforward extension of the following notions. There are variables of the form X_p which represents a sequence of rules defining the relation p. There are also variables of the form X_G representing allowed *goals* G to P. The meaning of a store is characterized by the *least model* of P. For example, given the program P:

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(a).
```

the decision variables of interest include X_p, X_q and X_r. A subtle point here is that the variables in the program are constants in the constraint store, formally speaking. The meaning of this store (and remember that the program itself is the store), assigns a subset of the Herbrand universe for each of X_p, X_q and X_r, in particular, $X_p = \{p(a)\}$, $X_q = \{q(a), q(b)\}$, and $X_r = \{r(a)\}$.

Where $\mathcal{R}_p$ denotes a rule for the relation p, and $\mathcal{G}$ denotes the body of a rule, some possible updates are: $\texttt{append(}X_p\texttt{, }\mathcal{R}_p\texttt{)}$, $\texttt{insert(}X_p\texttt{, }\mathcal{R}_p\texttt{)}$, $\texttt{delete(}X_p\texttt{, }\mathcal{R}_p\texttt{)}$, and $\texttt{goal(}\mathcal{G}\texttt{)}$. Using the above example program and applying the update $\texttt{append(}X_r\texttt{, r(b))}$, we obtain the new program

```
p(X) :- q(X), r(X).
q(a).
q(b).
r(a).
r(b).
```

whose meaning is now updated to $X_p = \{p(a), p(b)\}$, $X_q = \{q(a), q(b)\}$, and $X_r = \{r(a), r(b)\}$. Similarly for the other updates `insert` and `delete`.

Finally, consider an update $\texttt{goal(}\mathcal{G}\texttt{)}$. This update is passive and returns a nonempty list of (ground) substitutions, and fails if there is no such list. For the example program above, `goal(p(X))` returns a sequence of two substitutions $\theta_1 = \{X \mapsto a\}$ and $\theta_2 = \{X \mapsto b\}$. This syntax is to be regarded as shorthand for the formula $\mathcal{C} = \texttt{p(X)}\theta_1 \wedge \texttt{p(X)}\theta_2$, and note that under our interpretation of logic programs, $\forall X.P \models \mathcal{C}$ as required.

Example (one bit) This constraint store represents just one bit, addressed by one global variable X. There are thus only two meaningful stores (excluding the void store $\perp$). Four possible updates, whose meanings are self-explanatory, are:

$\texttt{is\_zero(}X\texttt{)}$, $\texttt{is\_one(}X\texttt{)}$, $\texttt{make\_zero(}X\texttt{)}$, $\texttt{make\_one(}X\texttt{)}$

Clearly more sophisticated data structures such as sets, arrays, trees, etc, can be modelled in this manner.

2.2 Reactor

A *reactor* is a program, invoked by an agent, for the sole purpose of interacting with some constraint store, *at the proper time*. The model is that the agent program submits the reactor to the store and waits for it to terminate (much like a remote procedure call). Multiple agents can submit reactors at any time to interact with the store.

Intuitively, a reactor is a program written in the OCP reactive programming language, whose operations are defined upon the existence of conditions that are realized elsewhere, and whose eventuality and timeliness cannot be reasoned about by the agent. Other examples of reactive programming languages are CCP, Esterel [4] and StateCharts [6]. In this section, we mainly discuss the intuitive use of reactors via examples, using informal syntax enclosed in $\textsc{react}\{\ \cdots\ \}$, and we use informal semantics.

In what follows, the notation Δ denotes a constraint system when spoken about generically, and denotes a particular constraint store otherwise.

The simplest reactor is of the form

$\textsc{react}\{\ \delta(\Delta)\ \}$

which, when executed within an agent, causes the update δ upon the store Δ to be executed (eventually). The reactor terminates upon termination of the update (which if successful, $\delta(\Delta)$ returns a value which can be used in the agent), and control reverts back to the agent.

A simple and useful reactor is

$\textsc{react}\{\ \mathrm{wait}(\delta(\Delta))\ \}$

which suspends until such time the update $\delta(\Delta)$ is enabled, and then it executes. This reactor terminates when δ does. Thus it simple blocking mechanism which can be used to synchronize with other agents (or itself, in case it is parallel or threaded). We could define that such a reactor return as a value that which is returned by the update δ.

In general, a reactor specifies not just when and what happens when certain updates can succeed, but also sequences of updates that are *atomic*. For example, we could define that

$\textsc{react}\{\ \delta(\Delta) \longrightarrow \delta'(\Delta)\ \}$

executes only when $\delta(\Delta)$ is enabled, and then executes both $\delta(\Delta)$ and $\delta'(\Delta)$ (regardless of whether the latter is successful). It could be further specified that these two updates occur atomically in the sense that no intervening update may occur between the two.

Now consider the store in the one-bit example; the two reactors

$\textsc{react}\{\ \mathrm{is\_one}(X) \longrightarrow \mathrm{make\_zero}(X)\ \}$
$\textsc{react}\{\ \mathrm{make\_one}(X)\ \}$

serve as the familiar *wait* and *signal* operations on a binary semaphore, here represented by the variable X in some constraint store. Note that the first reactor above is to be executed atomically.

In the next example, called the Dijkstra Guard, all updates δ_i apply to a given system Δ, for notational brevity:

$$\textsc{react}\left\{\begin{array}{ll} & \delta_1 \longrightarrow \delta_1' \\ [] & \delta_2 \longrightarrow \delta_2' \\ [] & \cdots \\ [] & \delta_n \longrightarrow \delta_n' \end{array}\right\}$$

The idea here is that this reactor awaits for one of the updates δ_i to be enabled by Δ, choosing nondeterministically if there is more than once choice, and then performs the updates δ_i and δ_i'. We could further specify that these two updates are performed *atomically*. In fact, we could specify that only *some* of the δ_j, δ_j' updates are performed atomically (with some new syntax, say by using both $\longrightarrow$ and $\Longrightarrow$). Thus this reactor acts like a primitive step in CCP programs, except for atomicity behavior, where it is indeed quite different.

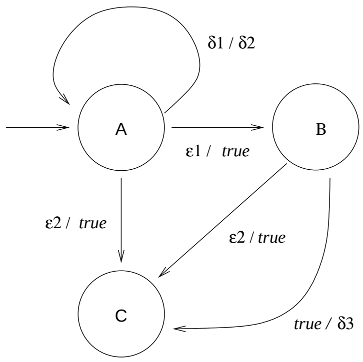
Consider next that the constraint store Δ is a PROLOG program P, as in the example above. The following reactor

$\textsc{react}\{\ \mathrm{goal}(A_1, A_2, \cdots, A_n) \longrightarrow \delta\ \}$

where the A_i are allowed atomic goals, serves to obtain a nonempty and finite sequence of substitutions $\theta_1, \theta_2, \cdots, \theta_k$ such that θ_i is an answer subtitution to P and the goal $?-A_1, A_2, ..., A_n$. If and when such a list is obtained, then the updates given by $\delta\theta_1, \delta\theta_2, \cdots, \delta\theta_k$ are executed. Using the first example PROLOG program above,

```
main ⟵ p.
p ⟵ δ1 | δ2, p.
p ⟵ ϵ1 | q.
p ⟵ ϵ2 | true.
q ⟵ ϵ2 | true.
q ⟵ true | δ3.
```

```
do
    do
        loop
            await δ1 ;
            δ2 ;
        end
    watching ϵ1 ;
    δ3 ;
watching ϵ2 ;
```



(a) (b) (c)

Figure 1: Example Reactive Behavior

REACT{ p(X) ⟶ delete(q(X)) }

results in the store[1] :

```
p(X) :- q(X), r(X).
r(a).
r(b).
```

We now present three examples of what is intuitively the same[2] reactor, written in the style of three familiar reactive programming frameworks: CCP, Esterel and StateMate. Informally, the behavior we'd like to capture is: *as long as δ_1 is true, repeatedly execute δ_2, stopping when ϵ_1 or ϵ_2 is true. Then execute δ_3 unless ϵ_2 is true.* We exemplify later the corresponding OCP reactor. Figure 1(a), (b) and (c) show example programs for this via programs in CCP, Esterel and Statecharts respectively[3]. Note that in the latter, there are 3 states: A (start), B, C (halt). The notation ϵ/δ denotes that the update is enabled upon $\epsilon(\Delta)$ and performs action δ.

So far we have only presented OCP reactors which terminate in one step. In general, a reactor is written in a Turing-complete language,[4] and these may not terminate. Meanwhile, we now consider a formalization of reactors. The syntax for an OCP reactors r is defined below. The notation c stands for a passive update, and δ any update.

$$
\begin{array}{rcll}
r & ::= & c \Rrightarrow_n \delta & \text{trigger immediately, } n \text{ answers} \\
 & | & c \Rightarrow_n r & \text{trigger eventually, } n \text{ answers} \\
 & | & r \,\&\, r & \text{interleave} \\
 & | & r \mid r & \text{early commit} \\
 & | & r : r & \text{late commit} \\
 & | & r \,;\, r & \text{sequencing} \\
 & | & [r] & \text{“atomic” transaction} \\
 & | & \texttt{repeat } r \texttt{ until } c & \text{uninterruptable loop} \\
 & | & \texttt{repeat } r \texttt{ watching } c & \text{interruptable loop}
\end{array}
$$

The formal semantics treatment of reactors is beyond the scope of this paper. We however present in

[1]There are interesting considerations as to whether the sequence of updates should apply to the orginal store, or to the current store as it changes with ongoing updates.

[2]Not necessarily precisely the same because of detailed considerations such as synchronous, asynchronous execution and the timing of operations.

[3]The CCP program is somewhat more liberal in that it does not force the immediate termination of δ_2 upon ϵ_1 being true; similarly for δ_3 and ϵ_2.

[4]Given an appropriate constraint store.

the Appendix a concise formal operational semantics of reactors. Here we rely on informal semantics and examples.

The basic reactor primitives are the two "triggers" immediately ($\Rrightarrow_n$) and eventually ($\Rightarrow_n$). The immediate trigger, $c \Rrightarrow_n \delta$, waits for there to be n distinct instantiations[5] θ_1, ...,θ_n which enable $c\theta_1$, ..., $c\theta_n$ in the current store Δ, and executes the update $\delta\theta_1$, ..., $\delta\theta_n$. The eventually trigger, $c \Rightarrow_n r$, allows the update to be a general reactor and relaxes the immediate update requirement to be any time. Thus it differs from immediately in that c may no longer be enabled when r executes. We disallow a general reactor to be the action of an immediate trigger because of the difficulty of maintaining the enabling condition c given a complex right hand side.

The interleaving construct, r_1 & r_2, corresponds to concurrent execution of r_1 and r_2. Sequencing with r_1 ; r_2 requires that reactor r_1 finish before r_2 starts. Early commit, $r_1 \mid r_2$, is a form of eager disjunction, the first reactor which can execute an update aborts the other. Late commit, $r_1 : r_2$, allows both reactors r_1 and r_2 to run and the reactor which terminates will abort the other. An atomic transaction, $[r]$, executes reactor r with its own private store until it terminates, the results from that private computation have to be merged back into its parent store.[6] Finally, iteration is achieved by the two loop constructs. The uninterruptable loop, `repeat` r `until` c, executes r until c becomes enabled. The interruptable loop is similar but allows r to be aborted when c is enabled.

For the programs in figure 1 above, we now can write (by dropping the now unnecessary REACT notation) a reactor as follows.

$$(\texttt{repeat}\ \delta_1 \Rightarrow \delta_2\ \texttt{watching}\ \epsilon_1)\ ;\ (\epsilon 2 \Rightarrow \texttt{true} \mid \delta_3)) \mid \epsilon_2$$

We conclude this section with yet a few more example reactors.

Assume that there is a clock and that the predicate $\mathcal{T}time(t)$ that holds if and only if t is the current time according to this clock. We also assume arithmetic on time and availability of functions to extract year, month, day, hour, minute, second, etc. from time. Assume that our reactor system is fair in choosing reactors for execution. We can express several interesting time-related features.

Example (Time) The alarm clock is an action r that should be executed between certain hours x and y of the day, provided some condition c is or becomes true. Let n be the number of instantiations on the free variables of c that we want to capture. Then the alarm clock can be expressed as

$$\mathcal{T}time(t) \wedge x \le \mathcal{T}hour(t) \le y \wedge c \Rightarrow_n r$$

Alternatively, if r is some simple update δ, the alarm clock can be as

$$\mathcal{T}time(t) \wedge x \le \mathcal{T}hour(t) \le y \wedge c \Rrightarrow_n \delta$$

Example (Timeout) The "timeout" is some action r that must be executed and completed within x time units. This basic notion of timeout can be implemented in a reactor system as

$$\mathcal{T}time(t) \Rightarrow_1 (r : (\mathcal{T}time(t') \wedge t' - t > x \Rrightarrow_1 !))$$

This implementation of timeout interrupts r if it does not finish within x time units. A more common situation is where r has a guard and so long as this guard is satisfied within x time units, r should be allowed to continue until termination even if it takes more than x time units. This alternate notion of timeout can be implemented as

$$\mathcal{T}time(t) \Rightarrow_1 (r \mid (\mathcal{T}time(t') \wedge t' - t > x \Rightarrow !))$$

A more sophisticated version where some alternate action r' is carried out on a timeout can be implemented as

$$\mathcal{T}time(t) \Rightarrow_1 (r \mid (\mathcal{T}time(t') \wedge t' - t > x \Rightarrow r'))$$

[5]Constraints rather than instantiations are more appropriate, but their treatment is more complicated, and hence omitted for this paper.

[6]Transactions can be nested.

A reactor system can perform recursive queries such as transitive closure of unordered graphs.

Example (Recursion) Let $P(x,y)$ means there is an edge from x to y. We want to create a relation $A(x,y)$ that holds if and only if there is a path from x to y. Here is a reactor system for creating A from P:

$$\begin{array}{l} P(x,y) \;\Rrightarrow\; +A(x,y)\ ; \\ \texttt{repeat} \\ \quad A(x,z) \;\wedge\; A(z,y) \;\wedge\; \neg A(x,y) \;\Rrightarrow\; +A(x,y) \\ \texttt{until}\ \forall x,z,y.\ A(x,z) \;\wedge\; A(z,y) \;\rightarrow\; A(x,y) \end{array}$$

Note that the passive update in the `until` condition assumes the use of quantifiers, and also note that the symbol $\rightarrow$ there denotes logical implication and not a reactor connective.

Finally, consider an active relational database system which allows certain trigger rules in the E-C-A form [18]: ON *event* IF *condition* THEN *action*. There are some connections between a reactor system and an active relational database system.

Example (Active Database) Let c be a Boolean combination and δ be a simple deletion or insertion on a table. Then a reactor such as $c \;\Rightarrow\; \delta$ can be implemented in an active relational database system as a series of trigger rules like this: For each table mention in c and for each possible kind of update event e on this table, add the rule ON e IF c THEN δ to the active relational database system. Hence a simple reactor can explode into a large number of trigger rules. Note that if the reactor was $c \;\Rightarrow_n\; \delta$, it would not be so easy to implement using E-C-A rules, as these rules cannot capture the notion of choosing just n instantiations.

In commercial relational databases, some restrictions are imposed on E-C-A trigger rules. The action in an E-C-A trigger rule is a combination of simple deletions from and insertions into relational tables; the condition in an E-C-A trigger rule is a Boolean combination; and recursive firing of E-C-A trigger rules is disallowed. Under such a setting, we claim that a reactor system is strictly more expressive than typical E-C-A trigger rules.

3 An Abstract Implementation Model

A central issue behind an implementation of an OCP system is clearly the management of efficient wakeups of blocked reactors. In analogy with active databases, there is a dependency on some primitive triggering mechanism on the part of the constraint system. However, such a mechanism works by considering only the internal state of the system, and not at all on the *environment of reactors* currently blocked by the constraint system. This gap motivates this section, which is an informal discussion on where the major implementation considerations lie.

The underlying philoshpy throughout this section is that there are in general a large number of reactors, and each of the frequent updates of the constraint store is of no consequence to most of the reactors.

3.1 Trigger Table

In what follows, we informally say that a reactor r is *blocked at* Δ if no progress is possible for r using the constraint system Δ. (This can be formalized using the transition rules in the appendix.)

Definition A *trigger table* $\mathcal{T}$ is associated with a constraint store Δ. It is characterized by (a) the initial state Δ_{init} of Δ, and (b) the sequence of updates $\delta_1, \delta_2, \cdots, \delta_n$ that gave rise to the current state Δ. That is,

$$\Delta = \delta_n(\delta_{n-1}(\cdots(\delta_1(\Delta_{init}))\cdots)).$$

Thus the trigger table is an *abstraction* of $\delta_1, \delta_2, \cdots, \delta_n$ and Δ. $\Box$

In principle the trigger table can subsume Δ in the sense that it contains all information about Δ, and additionally, its complete history. A naive implementation of an OCP system would not require a trigger table at all. Whenever a update on Δ occurs, every blocked reactor can be re-evaluated for progress. This approach is clearly impractical in general.

The intended use of a trigger table requires that only a *partial* representation of the history of updates. More specifically, the purpose of the trigger table is to efficiently facilitate a mapping which indicates, as a result of recent updates to Δ, *just* those blocked reactors which may now have become unblocked. Most importantly, the maintenance of the trigger table needs to be computed *incrementally*.

The rest of this section assumes, for simplicity, that the OCP system is *reactor eager* in the following sense: let $\Delta \xrightarrow{\delta} \Delta'$ denote an update from the store Δ to Δ', and R denote the set of blocked reactors associated with Δ. Then the only reactors in R that are unblocked on Δ' are those that are determined by the trigger table of Δ'. Intuitively, this means that the trigger table need only keep information about the changes resulting from the *one* transition δ to Δ to Δ' for the purpose of determining which reactors in R may now become unblocked. In a practical setting, it would be necessary to consider not one but a number of transitions.

3.2 The Registry

What remains now is to describe how reactors actually communicate with the constraint server. We'll call this generic interface a "registry" because it registers a reactor as "live" upon execution by an agent, it processes the execution of the reactor, and finally, it deregisters the reactor so that the agent can continue executing. In addition to this bookeeping, the registry

- interprets updates to the store, by inspecting the associated trigger table, in order to identify those reactors which need to be scheduled for execution;
- actually schedules the execution.

One could have that the trigger table itself perform the first function. It would store the set of all currently blocked reactors, and upon an update of Δ, this set is modified by removing those reactors which have become unblocked. (The set is augmented when new reactors arrive.) One disadvantage of this direct method (as opposed to a specialized registry mechanism) is that the full range of all possible blocked transactions needs to be understood by the algorithms of the constraint store. In practice, only certain properties of the store are practically saveable for use in the trigger table. As an analogy, triggers in databases store information about base relations, eg. that one of them has just been deleted, inserted or modified (eg. in the Oracle database). A reactor could await something from a view involving a complicated combination of the base relations. The direct implementation of the triggers would then require that updates to the view must themselves be immediately noticed and appear in the table. This adds considerable complication to the implementation of a general-purpose database system (and this is in fact forbidden in the Oracle database).

Another example is given by the use of a finite-domain solver [7] as the constraint store, with entailment of arithmetic constraints representing the blocked reactors. Reasoning in the store proceeds by algorithms which reason about the current domains of variables, and these are narrowed by propagation methods. A reasonable trigger table would indicate those variables whose domain has just changed by the last update; a separate procedure (the proposed registry mechanism) would then be required to use this information to determine which blocked reactor may now proceed (eg, consider only those reactors involving a variable that appears in the trigger table).

Recall the philosophy that the trigger table stores just certain and *predefined* kinds of information about recent updates (such as updates to base relations in databases). Such information, determined by the nature of the constraint store operations and internal structures, and not by the particular

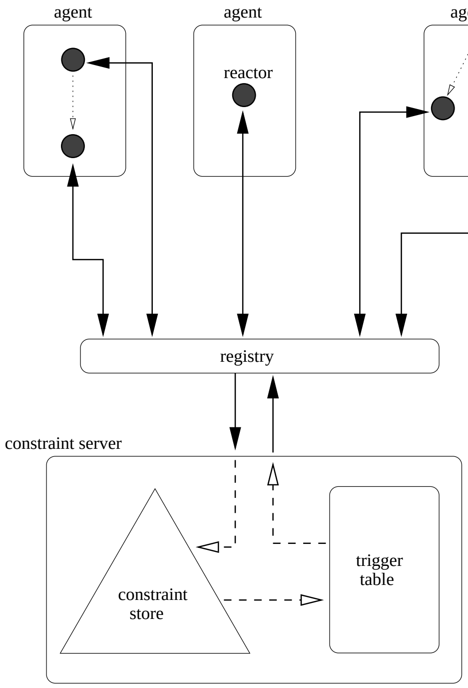


Figure 2: OCP Architecture

application at hand, must be obtained efficiently. This is because the costs of updates to the trigger table are somewhat fixed, and are borne regardless of the context of which reactors are currently live and waiting on the store. It is not feasible to develop a store which can update its trigger table selectively. In other words, a trigger table contains all the basic information about store updates that can be used to accurately determine if *any* reactor may now proceed, regardless of whether this reactor actually exists.

We now exemplify a possible efficient registry mechanism. Suppose reactors were awaiting arithmetic conditions of the form $n \leq X \leq m$ where n and m are numbers (eg. a stock trading agent awating certain stocks to be priced in a certain range). The value of X is updated frequently (as are stock prices), but the number of reactors interested in a particular update to X (in a particular stock price change) is relatively small. The registry could employ a range-indexing algorithm [2] in order to identify the appropriate subset of reactors.

In summary, by adopting the implementation philosophy that there are many registered reactors, and any one store update is unrelated to the vast majority of them, the main job of the registry is in translating the blocking conditions of a reactor into a sequence of concrete observables provided by the trigger table. The ultimate goal is to compute a *conservative* estimate of the subset of blocked reactors possibly affected by a store update (ie. a superset thereof).

Figure 2 illustrates the top-level structure of an OCP system.

4 Paradigmatics

Here we discuss the potential of OCP in three application domains in which constraints have played an important role: search, (constraint) databases, and concurrent programming.

4.1 Search

We consider a classic application of sequential CP, backtracking search. The problem at hand involves a constraint system of constraints over $\{0, 1\}$, and to find a binary assignment to the decision variables $X_1, \ldots, X_n$ so that some base conjunction of constraints $\mathcal{C}_0(X_1, \ldots, X_n)$ is satisfied, or report failure in case no such assignment exists. The constraints $\mathcal{C}_0$ need not be specified here, but we assume that the store contains just one conjunction of atomic constraints, represented as a set of its conjuncts. A search for a one solution for decision variables $X_1, \ldots, X_n$ can be realized by the C-like program below, which uses the updates `softell` and `retract`. The former was described above in an example, whilst `retract(`$\mathcal{C}$`)` has the effect of removing the constraint $\mathcal{C}$ from the store. In what follows, we assume the decision variables are organized into an array $X[i], 1 \leq i \leq n$. The procedure below is called with `solve(1)`.

```
solve(i) {
      if (i > n) return TRUE;
      if (softell(X[i] = 0)) {
            if (solve(X[i + 1])) return TRUE
            retract(X[i] = 0)
      }
      if (softell(X[i] = 1) {
            if (solve(X[i + 1])) return TRUE
            retract(X[i] = 1);
      }
      return FALSE
}
```

Thus the constraint system here acts with the above agent in functionally the same way a constraint programming library (eg. ILOG Solver) does to regular sequential programs.

The next example deals with concurrent agents acting to perform parallel search. We retain the same problem over $\mathcal{C}_0(X_1, \ldots, X_n)$. In addition, the constraint store also contains a constraint database for coordination purposes. The current setting involves a master agent which serves to divide the search space amongst any number of worker agents that volunteer at any time. Consider the search tree T for this problem. This is a full binary tree of $n + 1$ levels, whose left subtree represents the choice $x[1] = 0$ and the the right subtree $x[1] = 1$; similarly for the remaining nodes wrt the other decision variables. We identify the individual nodes of T using a string of bits so that the empty string represents T, 0 represents the left subtree of T whose right subtree is 0.1, etc.

At any time, there are a number of worker agents, each assigned a distinct node in T; we say that m is "marked" with a. The idea is that that agent a with node m is performing the search problem associated with m. We will maintain the property that the set of marked nodes, plus a selection of nodes which are "closed", ie: search below them is complete, forms a *frontier* for T. This ensures that the remaining search space is covered by the available worker agents.

An agent a, using its unique identifier id, starts by voluteering its service to the master agent. It then awaits the assignment of a node identifier m so that it can use the following program segment to start search on m. In what follows, we can extract from m its associated decision variable $x[i]$; the length of m is denoted $|m|$. The reactor `fail` returns TRUE when some solution (published by any agent) is found, and the reactor `succeed(m)` publishes a solution. The accessing of marks below is to be thought of as being updates to the store which contains a database of the current marks. (The logic behind the marks is explained later.) We assume without further ado that this agent injects into the store a failure notification when it has finished its search unsuccessfully. A final but important assumption: the update `softell`$_{\mathcal{C}_0}$`(id, x[i] = 0)` below is a satisfiability test of the formula $\mathcal{C}_0 \wedge C_m \wedge X_i = 0$ where $\mathcal{C}_0$ is the basic constraint, $\mathcal{C}_m$ are all the equations of the form $X_j = 0$ or $X_j = 1$, $1 \leq j < i$ *corresponding to the node* m. Similarly for the other update `softell`$_{\mathcal{C}_0}$`(id, x[i] = 1)`.

```
solve(id, m) {
      if (fail) exit();
      if (|m| > n) { succeed(m); exit() }
      if (softell_C0(id, x[i] = 0))
           if (!solve(id, m.0)) retract(id, x[i] = 0)
      if (mark(id) == m) move mark(id) to m.1
      if (mark(id) is not above m) exit();
      if (softell_C0(id, x[i] = 1))
           if (!solve(id, m.1)) retract(id, x[i] = 1)
      return FALSE
}
```

The master agent can be simply described as a multi-threaded process which accepts volunteers in one thread, and dispatches a job by assigning a marked node to a volunteer agent in the other thread. In another thread, it bookeeps which amongst the workers have not yet failed, and the last thread simply listens for a solution found.

In the program fragment below, we assume that `assign_work(id)` eventually causes the agent code above to be called, with a suitable node m. Call this newly voluteered agent a'. In order to determine such m, the master maintains a list of current assignments (a, m_a) indicating that the currently active worker a is responsible for the subtree rooted at m_a. If the list is empty, then we are just starting and so the incoming agent gets the whole tree. Otherwise, randomly choose one current assignment (a, m_a) and (a) re-assign $(a, m_a.0)$, and (b) assign $(a', m_a.1)$. That is, "split" the job of a and a'.

Note below that the actions of volunteering and then being assigned a node is performed atomically. The reactor `agent_failed(id)` simply records that agent id has failed. The reactor `check_all_failed` makes the flag `done` available when all agents have failed.

```
repeat
     volunteer(id) ⇛ assign_work(id)
     &
     agent_failed(id)⇛ check_all_failed
     &
     succeed(m) ⇛ done
until done
```

There are some subtle properties required on marks and the current search position of agents in order for this system to work. First, note that when an agent is just about move from the left subtree of its marked node to the right, that it moves the mark along with it. The logic here is that the left subtree is now closed, and so the mark can be moved deeper down the tree. More importantly, it means that when the master agent splits a node, it will be the case that the current agent working under that node will in fact be in the left subtree of the node.

Finally for this example, we observe that a practical challenge is to contain the bottleneck caused by the $\texttt{softell}_{\mathcal{C}_0}$ updates needed by the agents as they search. In practice, if each agent could access a local constraint solver which initially holds the base constraints, and which subsequently handles the constraint solving associated with the $\texttt{softell}_{\mathcal{C}_0}$ updates, then the only communication between a worker agent and the outside world would be the (far less frequent) access of marks.

Database Programming

Consider a constraint store representing a constraint database. We assume that the specific kinds of terms here are as in the real arithmetic-based CLP($\mathcal{R}$)system [9]. In this example, borrowed from [1], the database P contains current information about stocks and their prices, and further, there is a single predicate `time(`n`)` where n, a positive integer, denotes the current hour. We leave unspecified how the stock prices and the `time` predicate are updated. An trader seeks to do the following: *if the price of* `apple` *is less than that of* `ibm` *for a period of five hours or more, buy.* Consider a reactor of the general form:

$$
\begin{array}{l}
\texttt{repeat} \\
\qquad A_1, A_2, \cdots, A_n \Rrightarrow \delta_1, \delta_2, \cdots, \delta_m \\
\texttt{watching } C_1, C_2, \cdots, C_k
\end{array}
$$

whose intuitive semantics is as follows: if and when the goal $?-A_1, A_2, \cdots, A_n$ has answer constraints $\tilde{\theta}$ on the current P, then perform $\delta_1\theta_i, \delta_2\theta_i, \cdots, \delta_m\theta_i$ over all $\theta_i \in \tilde{\theta}$. These updates δ_i have been described in the CDB example in section 2, but we add one here: `return(t)`, where `t` is bound to a value whose type is understood by the agent. If, however, the goal $? - C_1, C_2, \cdots, C_k$, run on the current value of P, is known to succeed before any of the δ_i is performed, then the entire reactor may terminate. Our trader agent now can be written as follows:

```
WAIT:
      t =  (stock(apple, A), stock(ibm, B), A < B ⇛ time(T), return(T))
COUNT:
      b =  repeat
                 stock(apple, A), stock(ibm, B), A ≥ B ⇛ return true
           watching time(T), T − t > 5
      if (b) goto WAIT;
BUY:
      ··· buy the stock ···
```

It is folklore that there is some similarity between agent programming and that of active databases [1, 18]. OCP can make the similarity explicit.

Concurrent Programming

We have already seen an example of a binary semaphore. More friendly to programmers is the more structured framework of Linda [5] which provides a certain constraint server based upon a *tuplespaces*. Updates are the expected ones on tuples, and synchronization is achieved by awaiting a tuple of a certain form to appear in the tuplespace. The OCP formulation in our CDB examples essentially cover the same ground, but with a far greater expressive power in terms of data structuring and in terms of constraints on the underlying terms. (However, Linda has special features, eg *eval* which creates processes to fill in the fields of a tuple. Such features are beyond the scope of the OCP model.)

For the rest of this subsection, we present a special-purpose constraint system whose purpose is toward declarative concurrency. The idea is that the constraints specify how occurrences of certain events may run relative to one another, and this clearly has general application to parallel programming.

Consider *event classes* $e, f, \cdots$, and *event occurrences* (or simply *events*) $e[1], e[2], \cdots, f[1], f[2], \cdots$. The item of real interest is the event class, and for this example, it suffices to think of a class as a certain program point in an agent. The constraints in our server are of type integer, and event classes are represented as sequences $e[1], e[2], \cdots$ of integer variables. Where n and m are specific positive integers, the constraints are conjunctions and disjunctions of formulas of the form:

$$
\begin{array}{l}
e[n] = m \\
e[n] < f[m] \\
\forall i, j\,(e[i+n] < f[j+m])
\end{array}
$$

and there is also one special constraint $\texttt{time} = n$ which is assumed to behave like a clock. That is, we have quantified arithmetic formulas over array and integer expressions.

The intended model is one which assigns values to those events $e[i]$ which have executed already, and these values shall be the timestamp (as defined by the changing variable `time`. Thus in the model, at a given time, $\forall i(e[i-1] < e[i]$ is true, where i ranges up to number of times the e has been executed. We thus have notion of "current occurrence of e" being the largest $e[n]$ that has a value (which of course does not exceed that value of `time`). In other words, at any given time t (ie when `time =` t is in the store), there is a history of the event class e: a sequence of increasing timestamps $e[1], \ldots, e[k]$ where $k = current(e)$ and $e[1] \leq e[2] \leq \ldots \leq e[k] \leq t$ where $k = current(e)$.

Consider the standard producer/consumer problem, here with a bounded buffer of size 5, say. The producer and consumer agents are associated to the event classes p and c respectively. The server may thus contain the formula $\begin{array}{l} \forall i\ (p[i] < c[i]) \\ \forall i\ (c[i] < p[i+5]) \end{array}$

and the agents may be written as follows[7] .

```
PRODUCER:
     softell(current(p) = time) ⇛  !
     ...  produce ...
     goto PRODUCER;

CONSUMER:
     softell(current(c) = time) ⇛  !
     ...  consume ...
     goto CONSUMER;
```

Note that we may have an *arbitrary* number of producers and consumers as defined above. Next, consider mutual exclusion between two agents, the first has event classes e and e' corresponding to the entry and exit program points of its critical section. Similarly we have f and f' for the other agent. Consider the store

$$\begin{array}{l} \forall i(e[i] < e'[i]) \\ \forall i(f[i] < f'[i]) \\ (e'[1] < f[1] \bigvee f'[1] < e[1]) \\ \forall i,j\,(e'[i] < f[j] \longrightarrow e'[i+1] < f[j] \bigvee f'[j] < e[i+1]) \\ \forall i,j\,(f'[j] < e[i] \longrightarrow e'[i] < f[j+1] \bigvee f'[j+1] < e[i]) \end{array}$$

Each agent, say the one for e, enters and leaves its critical section through the following reactors, respectively:

```
WAIT(SOFTELL(current(e) = TIME))
WAIT(SOFTELL(current(e') = TIME))
```

[7]The update ! does nothing but terminates successfully.

References

[1] J. Bailey, M. Georgeff, D. Kemp, D. Kinny and K. Ramamohanarao, "Active Databases and Agent Systems - A Comparison", Proceedings of the Second International Workshop on Rules in Database Systems, Springer-Verlag LNCS 985, 342-356, 1995.

[2] Bertino, E., Ooi, B.C., Sacks-Davis, R., Tan, K.-L., Shidlovsky, B., and B. Catania, "Indexing Techniques for Advanced Database Systems", Kluwer Academic Publishers, 1997.

[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations", *IEEE Computer*, 29(2), 18–28, 1996.

[4] G. Béry, G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation", *Science of Computer Programming*, 19(2), 87–152, 1992.

[5] N. Carriero and D. Gelernter, "Linda in context", *Comm. of the ACM*, 32(4), 444–458, 1989.

[6] D. Harel and E. Gery, "Executable Object Modeling with Statecharts", *IEEE Computer*, 30(7), 31–42, 1997.

[7] P. van Hentenryck, "Constraint Satisfaction in Logic Programming", MIT Press, 1990.

[8] J.Jaffar and M.Maher, "Constraint Logic Programming: a Survey", *Journal of Logic Programming*, Special 10^{th} Anniversary Issue, Vols 19/20, May/July 1994.

[9] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap, "The CLP($\mathcal{R}$) Language and System", *ACM Transactions on Programming Languages and Systems*, 14(3), July 1992.

[10] P. Kanellakis, P.C. Kuper, and P. Revesz, "Constraint Query Languages", *Journal of Computer and System Sciences*, 51(1), 26-52, 1995.

[11] Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *Journal of Computer and System Sciences*, 55(2):241–272, October 1997.

[12] J.W. Lloyd, *Foundations of logic programming*, Springer-Verlag, 2nd ed., 1987.

[13] K. Marriott and P.J. Stuckey, *Programming with Constraints*, The MIT Press, 1998.

[14] *ILOG Solver 4.0, Reference Manual*, ILOG Inc., 1997.

[15] V. Saraswat, *Concurrent Constraint Programming*, MIT Press, 1993.

[16] G. Schlageter. Optimistic methods for concurrency control in distributed systems. In *Proceedings of 7th International Conference on Very Large Data Bases*, pages 125–130, 1981.

[17] G. Smolka, "The Oz Programming Model", in J. van Leeuwen (ed.), *Computer Science Today*, LNCS 1000, Springer Verlag, 1995.

[18] Jennifer Widom, Stefano Ceri, and Umeshwar Dayal, "Active Database Systems : Triggers and Rules for Advanced Database Processing", Morgan Kaufmann, 1995.

Appendix

We first extend slightly the language for reactors presented in section 2:

r	$::=$	$c \Rrightarrow_n \delta$	trigger immediately, n answers
	$\mid$	$c \Rightarrow_n r$	trigger eventually, n answers
	$\mid$	$r \ \& \ r$	interleave
	$\mid$	$r \mid r$	early commit
	$\mid$	$r : r$	late commit
	$\mid$	$r \ ; \ r$	sequencing
	$\mid$	$[r]$	"atomic" transaction
	$\mid$	`repeat` r `until` c	uninterruptable loop
	$\mid$	`repeat` r `watching` c	interruptable loop
	$\mid$	$\mathtt{idle}_i$	
	$\mid$	$\mathtt{xact}_a$	
	$\mid$	`abort` r	

The reactors $\mathtt{idle}_i$, $\mathtt{xact}_a$, and `abort` r are not part of the language seen by a programmer. They are intermediate expressions produced by our operational semantics and are mentioned here for convenience. The i in $\mathtt{idle}_i$ is a trigger identifier returned by our operational semantics when a trigger is noted for later execution. The a in $\mathtt{xact}_a$ is a transaction identifier returned by our operational semantics when an "atomic" transaction is noted for execution. We assume that given updates δ and δ' we can composed them into a compound update function $\delta \circ \delta'$. We use ! to denote the special do-nothing update function.

States and Transitions

We now extend the notion of a constraint system Δ so that it contains a state describing the environment. Every reactor has "access" to this state. Similarly we extend the notion of an update, to become a *transition*, now denoted γ. The definition of states and transition are mutually recursive, as follows.

A state Δ comprises three items:

- a "theory" $\mathtt{theory}(\Delta)$,
- a finite set of "registered registered" $\mathtt{registered}(\Delta)$, and
- a finite set of "pending transitions" $\mathtt{pending}(\Delta)$.

An transition γ has one of the following eleven forms:

`birth` r	δ	`abort` r
$i : \bullet\ c \Rightarrow_n r$	$i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} \bullet\ r$	$i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} r\ \bullet$
$i : \bullet\ c \Rrightarrow_n \delta$	$i : c \Rrightarrow_n^{\theta_1,\ldots,\theta_n} \delta\ \bullet$	$a : \bullet\ r, \Delta$
$a : r, \Delta \xrightarrow{\gamma} \bullet\ \Delta', r'$	$a : r, \Delta \xrightarrow{\gamma} \Delta'\ \bullet$	

An transition that has one of the forms $i : \bullet\ c \Rightarrow_n r$, $i : \bullet\ c \Rrightarrow_n \delta$, or $a : \bullet\ r, \Delta$ is called a registered transition. The i in a transition that has one of the forms $i : \bullet\ c \Rightarrow_n r$, $i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} \bullet\ r$, $i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} r\ \bullet$, $i : \bullet\ c \Rrightarrow_n \delta$, $i : c \Rrightarrow_n^{\theta_1,\ldots,\theta_n} \delta\ \bullet$ is called the "trigger identifier" of the transition. The a in a transition that has one of the forms $a : \bullet\ r, \Delta$, $a : r, \Delta \xrightarrow{\delta'} \bullet\ \Delta', r'$, or $a : r, \Delta \xrightarrow{\delta'} \Delta'\ \bullet$ is called the "transaction identifier" of the transition. In an update that has one of these forms $i : \bullet\ c \Rightarrow_n r$, $i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} \bullet\ r$, $i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} r\ \bullet$, $i : \bullet\ c \Rrightarrow_n \delta$, $i : c \Rrightarrow_n^{\theta_1,\ldots,\theta_n} \delta\ \bullet$, it is assumed that $FV(u) \subset FV(c)$ and $FV(r) \subset FV(c)$, where $FV(e)$ denotes the free variables in the expression e. Each trigger identifier or a transaction identifier is allowed to occur at most once in $\texttt{registered}(\Delta) \cup \texttt{pending}(\Delta)$. Only registered transitions are allowed in $\texttt{registered}(\Delta)$ and only transitions of the form $i : c \Rightarrow_n^{\theta_1,\ldots,\theta_n} \bullet r$ are allowed in $\texttt{pending}(\Delta)$.

We assume that there is an operation $\dagger$ for merging the theories of two states. For example, if the theory of a state is a finite set of ground facts as in PROLOG, $\dagger$ could be just the set union operation. We define $\texttt{clean}(\Delta)$ to be a state Δ' derived from the state Δ by copying the latter's theory and setting the other components to empty; that is, $\texttt{theory}(\Delta') = \texttt{theory}(\Delta)$, $\texttt{registered}(\Delta') = \{\}$, and $\texttt{pending}(\Delta') = \{\}$. Also, we define $\texttt{abort}_{I,A}(\Delta)$ to be a state Δ' derived from the state Δ by aborting all the triggers identified in I and all the transactions identified in A; that is, $\texttt{theory}(\Delta') = \texttt{theory}(\Delta)$, $\texttt{registered}(\Delta') = \{\gamma \mid \gamma \in \texttt{registered}(\Delta)$, the trigger identifier of γ (if any) is not in I, the transaction identifier of γ (if any) is not in $A\}$, $\texttt{pending}(\Delta') = \{\gamma \mid \gamma \in \texttt{pending}(\Delta)$, the trigger identifier of γ (if any) is not in I, the transaction identifier of γ (if any) is not in $A\}$.

Operational Semantics

A reactor system is composed of a state Δ and a finite set of reactors R. The dynamic behaviour of such a reactor system is defined by the transition relation $R, \Delta \xrightarrow{\gamma} \Delta', R'$. This reactor system transition relation is defined in terms of the reactor transition relation $r, \Delta \xrightarrow{\gamma} \Delta', r'$ and the reactor termination relation $r, \xrightarrow{gamma} \Delta'$ of individual reactors r and the state transition relation $\Delta \xrightarrow{\gamma} \Delta'$.

The basic version of the reactor system transition relation $R, \Delta \xrightarrow{\gamma} \Delta', R'$ is given below. We use the notation $X + x$ to mean a set Y such that $Y = X \cup \{x\}$ and $x \notin X$. We use the notation $X - x$ to mean a set Y such that $X = Y + x$.

***** Birth *****

$$\frac{\Delta \xrightarrow{\gamma} \Delta' \quad \gamma \text{ is } \texttt{birth}\ r}{R, \Delta \xrightarrow{\gamma} \Delta', R + r}$$

***** Death *****

$$\frac{r, \Delta \xrightarrow{\gamma} \Delta'}{R + r, \Delta \xrightarrow{\gamma} \Delta', R}$$

***** Progress *****

$$\frac{r, \Delta \xrightarrow{\gamma} \Delta', r'}{R + r, \Delta \xrightarrow{\gamma} \Delta', R + r'}$$

The "progress" transition rule given above is nondeterministic because a reactor r is randomly picked from the reactor set and allowed to progress one step further. This rule can be substituted with other rules to reflect a different way for choosing which reactor to run next. For example, we can add a side condition to this rule to say that r should preferrably contain a subexpression $\texttt{idle}_i$ such that the condition c of its corresponding noted trigger $i : \bullet\ c \Rightarrow_n r$ can be satisfied in the current state Δ.

Now we provide the rules for defining the reactor transition relaton $r, \Delta \xrightarrow{\gamma} \Delta', r'$ and the reactor termination relation $r, \Delta \xrightarrow{\gamma} \Delta'$. Both of these relations have fairly complex side conditions, which we abstracted out in the state transition relation $\Delta \xrightarrow{\gamma} \Delta'$ to be defined later.

***** Interleaving *****

$$\frac{r_1, \Delta \xrightarrow{\gamma} \Delta', r_1'}{r_1 \ \&\ r_2, \Delta \xrightarrow{\gamma} \Delta', r_1' \ \&\ r_2} \qquad \frac{r_2, \Delta \xrightarrow{\gamma} \Delta', r_2'}{r_1 \ \&\ r_2, \Delta \xrightarrow{\gamma} \Delta', r_1 \ \&\ r_2'}$$

$$\frac{r_1, \Delta \xrightarrow{\gamma} \Delta'}{r_1 \;\&\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_2} \qquad \frac{r_2, \Delta \xrightarrow{\gamma} \Delta'}{r_1 \;\&\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_1}$$

***** Sequencing *****

$$\frac{r_1, \Delta \xrightarrow{\gamma} \Delta', r_1'}{r_1;\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_1';\; r_2} \qquad \frac{r_1, \Delta \xrightarrow{\gamma} \Delta'}{r_1;\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_2}$$

***** Triggering eventually *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } i : \bullet\, c \;\Rightarrow_n\; r \quad i \text{ is fresh}\end{array}}{c \;\Rightarrow_n\; r, \Delta \xrightarrow{\gamma} \Delta', \texttt{idle}_i} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } i : c \;\Rightarrow_n^{\theta_1,\ldots,\theta_n}\; \bullet\, r\end{array}}{\texttt{idle}_i, \Delta \xrightarrow{\gamma} \Delta', \texttt{idle}_i}$$

$$\frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } i : c \;\Rightarrow_n^{\theta_1,\ldots,\theta_n}\; r\, \bullet\end{array}}{\texttt{idle}_i, \Delta \xrightarrow{\gamma} \Delta', r\theta_1 \;\&\; \cdots \;\&\; r\theta_n}$$

***** Triggering immediately *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } i : \bullet\, c \;\Rrightarrow_n\; \delta \quad i \text{ is fresh}\end{array}}{c \;\Rrightarrow_n\; \delta, \Delta \xrightarrow{\gamma} \Delta', \texttt{idle}_i} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } i : c \;\Rrightarrow_n^{\theta_1,\ldots,\theta_n}\; \delta\, \bullet\end{array}}{\texttt{idle}_i, \Delta \xrightarrow{\gamma} \Delta'}$$

***** Update and abortion *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is an update } \delta\end{array}}{\delta, \Delta \xrightarrow{\gamma} \Delta'} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } \texttt{abort}\ r\end{array}}{\texttt{abort}\ r, \Delta \xrightarrow{\gamma} \Delta'}$$

***** Delayed choice *****

$$\frac{r_1, \Delta \xrightarrow{\gamma} \Delta', r_1'}{r_1 \;:\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_1' \;:\; r_2} \qquad \frac{r_2, \Delta \xrightarrow{\gamma} \Delta', r_2'}{r_1 \;:\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_1 \;:\; r_2'}$$

$$\frac{r_1, \Delta \xrightarrow{\gamma} \Delta'}{r_1 \;:\; r_2, \Delta \xrightarrow{\gamma} \Delta', \texttt{abort}\ r_2} \qquad \frac{r_2, \Delta \xrightarrow{\gamma} \Delta'}{r_1 \;:\; r_2, \Delta \xrightarrow{\gamma} \Delta', \texttt{abort}\ r_1}$$

***** Early choice *****

$$\frac{\begin{array}{c}r_1, \Delta \xrightarrow{\gamma} \Delta', r_1' \\ \gamma \text{ is a registered transition}\end{array}}{r_1 |\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_1' |\; r_2} \qquad \frac{\begin{array}{c}r_1, \Delta \xrightarrow{\gamma} \Delta', r_1' \\ \gamma \text{ is not a registered transition}\end{array}}{r_1 |\; r_2, \Delta \xrightarrow{\gamma} \Delta', \texttt{abort}\ r_2 \;;\; r_1'} \qquad \frac{r_1, \Delta \xrightarrow{\gamma} \Delta'}{r_1 |\; r_2, \Delta \xrightarrow{\gamma} \Delta', \texttt{abort}\ r_2}$$

$$\frac{\begin{array}{c}r_2, \Delta \xrightarrow{\gamma} \Delta', r_2' \\ \gamma \text{ is a registered transition}\end{array}}{r_1 |\; r_2, \Delta \xrightarrow{\gamma} \Delta', r_1 |\; r_2'} \qquad \frac{\begin{array}{c}r_1, \Delta \xrightarrow{\gamma} \Delta', r_2' \\ \gamma \text{ is not a registered transition}\end{array}}{r_1 |\; r_2, \Delta \xrightarrow{\gamma} \Delta', \texttt{abort}\ r_1 \;;\; r_2'} \qquad \frac{r_2, \Delta \xrightarrow{\gamma} \Delta'}{r_1 |\; r_2, \Delta \xrightarrow{\gamma} \Delta', \texttt{abort}\ r_1}$$

***** Uninterruptable loop *****

$$\frac{(r \;;\; \texttt{repeat}\ r\ \texttt{until}\ c) \mid (c \Rrightarrow_0\ !), \Delta \xrightarrow{\gamma} \Delta', r'}{\texttt{repeat}\ r\ \texttt{until}\ c, \Delta \xrightarrow{\gamma} \Delta', r'}$$

$$\frac{(r \;;\; \texttt{repeat}\ r\ \texttt{until}\ c) \mid (c \Rrightarrow_0\ \texttt{perform}\ !), \Delta \xrightarrow{\gamma} \Delta'}{\texttt{repeat}\ r\ \texttt{until}\ c, \Delta \xrightarrow{\gamma} \Delta'}$$

***** Interruptable loop *****

$$\frac{(r\ ;\ \texttt{repeat}\ r\ \texttt{watching}\ c)\ :\ (c \Rrightarrow_0\ \texttt{perform}\ !), \Delta \xrightarrow{\gamma} \Delta', r'}{\texttt{repeat}\ r\ \texttt{watching}\ c, \Delta \xrightarrow{\gamma} \Delta', r'}$$

$$\frac{(r\ ;\ \texttt{repeat}\ r\ \texttt{watching}\ c)\ :\ (c \Rrightarrow_0\ \texttt{perform}\ !), \Delta \xrightarrow{\gamma} \Delta'}{\texttt{repeat}\ r\ \texttt{watching}\ c, \Delta \xrightarrow{\gamma} \Delta'}$$

***** Transactions *****

$$\frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } a : \bullet\ r, \Delta \quad a \text{ is fresh}\end{array}}{[r], \Delta \xrightarrow{\gamma} \Delta', \texttt{xact}_a} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } a : r, \Delta_o \xrightarrow{\gamma'} \bullet\ \Delta_n, r' \\ r, \Delta_o \xrightarrow{\gamma'} \Delta_n, r'\end{array}}{\texttt{xact}_a, \Delta \xrightarrow{\gamma} \Delta', \texttt{xact}_a} \qquad \frac{\begin{array}{c}\Delta \xrightarrow{\gamma} \Delta' \\ \gamma \text{ is } a : r, \Delta_o \xrightarrow{\gamma'} \Delta_n\ \bullet \\ r, \Delta_o \xrightarrow{\gamma'} \Delta_n\end{array}}{\texttt{xact}_a, \Delta \xrightarrow{\gamma} \Delta'}$$

As mentioned before, some complex side conditions for the above reactor transition and reactor termination relations are abstracted into a state transition relation $\Delta \xrightarrow{\gamma} \Delta'$. We provide this state transition relation below.

$$\frac{\gamma \text{ is } \texttt{birth}\ r}{\Delta \xrightarrow{\gamma} \Delta} \qquad \frac{\begin{array}{c}\gamma \text{ is an update } \delta \\ \texttt{theory}(\Delta') = u(\texttt{theory}(\Delta)) \\ \texttt{registered}(\Delta') = \texttt{registered}(\Delta) \\ \texttt{pending}(\Delta') = \texttt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'} \qquad \frac{\begin{array}{c}\gamma = \texttt{abort}\ r \\ I = \{i \mid \texttt{idle}_i \text{ appears in } r\} \\ A = \{a \mid \texttt{xact}_a \text{ appears in } r\}\end{array}}{\Delta \xrightarrow{\gamma} \texttt{abort}_{I,A}(\Delta)}$$

$$\frac{\begin{array}{c}\gamma = i : \bullet\ c\ \Rightarrow_n\ r \\ \texttt{theory}(\Delta') = \texttt{theory}(\Delta) \\ \texttt{registered}(\Delta') = \texttt{registered}(\Delta)\ + \gamma \\ \texttt{pending}(\Delta') = \texttt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = i : c\ \Rightarrow_n^{\theta_1,\ldots,\theta_n}\ \bullet\ r \\ \texttt{theory}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n \\ \texttt{theory}(\Delta') = \texttt{theory}(\Delta) \\ \texttt{registered}(\Delta') = \texttt{registered}(\Delta)\ - (i : \bullet\ c\ \Rightarrow_n\ r) \\ \texttt{pending}(\Delta') = \texttt{pending}(\Delta)\ + \gamma\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = i : c\ \Rightarrow_n^{\theta_1,\ldots,\theta_n}\ r\ \bullet \\ \texttt{theory}(\Delta') = \texttt{theory}(\Delta) \\ \texttt{registered}(\Delta') = \texttt{registered}(\Delta) \\ \texttt{pending}(\Delta') = \texttt{pending}(\Delta) - i : c\ \Rightarrow_n^{\theta_1,\ldots,\theta_n}\ \bullet\ r\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = i : \bullet\ c\ \Rrightarrow_n\ u \\ \texttt{theory}(\Delta') = \texttt{theory}(\Delta) \\ \texttt{registered}(\Delta') = \texttt{registered}(\Delta)\ + \gamma \\ \texttt{pending}(\Delta') = \texttt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = i : c\ \Rrightarrow_n^{\theta_1,\ldots,\theta_n}\ u\ \bullet \\ \texttt{theory}(\Delta) \models c\theta_j \text{ for all } 1 \le j \le n \\ \texttt{theory}(\Delta') = (u\theta_1\ \circ\ \cdots\ \circ\ u\theta_n)(\texttt{theory}(\Delta)) \\ \texttt{registered}(\Delta') = \texttt{registered}(\Delta) - (i : \bullet\ c\ \Rrightarrow_n\ \delta) \\ \texttt{pending}(\Delta') = \texttt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = a : \bullet\ r, \mathtt{clean}(\Delta)\\ \mathtt{theory}(\Delta') = \mathtt{theory}(\Delta)\\ \mathtt{registered}(\Delta') = \mathtt{registered}(\Delta)\ \ + \gamma\\ \mathtt{pending}(\Delta') = \mathtt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = a : r, \Delta_o \xrightarrow{\delta'} \bullet\ \Delta_n, r'\\ \mathtt{theory}(\Delta') = \mathtt{theory}(\Delta)\\ \mathtt{registered}(\Delta') = (\mathtt{registered}(\Delta) - (a : \bullet\ r, \Delta_o)) + (a : \bullet\ r', \Delta_n)\\ \mathtt{pending}(\Delta') = \mathtt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

$$\frac{\begin{array}{c}\gamma = a : r, \Delta_o \xrightarrow{\gamma'} \Delta_n\ \bullet\\ \mathtt{theory}(\Delta') = \mathtt{theory}(\Delta) \dagger \mathtt{theory}(\Delta_n)\\ \mathtt{registered}(\Delta') = \mathtt{registered}(\Delta) - a : \bullet\ r, \Delta_o\\ \mathtt{pending}(\Delta') = \mathtt{pending}(\Delta)\end{array}}{\Delta \xrightarrow{\gamma} \Delta'}$$

Finally, we remark that the transitions given above that involve "transactions" are very interesting. Each transaction $[r]$ is executing within their own private state and is isolated from the rest of the reactor system. The effect of such a transaction is then committed by merging its termination state with the current state of the entire reactor system. Our operational semantics let this transaction progress in a step-wise interleaved manner with the rest of the system without interference. This avoids the need of holding back other transactions. Such a transaction is, in a sense, analogous to an atomic transaction in an optimistic database system[16].