Interactive Interaction Constraints

José Proença, Dave Clarke

Interactive Interaction Constraints*

José Proença and Dave Clarke

iMinds-DistriNet, Department of Computer Science,
KU Leuven, Belgium
{jose.proenca,dave.clarke}@cs.kuleuven.be

Abstract. Interaction constraints are an expressive formalism for describing coordination patterns, such as those underlying the coordination language Reo, that can be efficiently implemented using constraint satisfaction technologies such as SAT and SMT solvers. Existing implementations of interaction constraints interact with external components only in a very simple way: interaction occurs only *between* rounds of constraint satisfaction. What is missing is any means for the constraint solver to interact with the external world *during* constraint satisfaction.

This paper introduces *interactive interaction constraints* which enable interaction during constraint satisfaction, and in turn increase the expressiveness of coordination languages based on interaction constraints by allowing a larger class of operations to be considered to occur atomically. We describe how interactive interaction constraints are implemented and detail a number of strategies for guiding constraint solvers. The benefit of interactive interaction constraints is illustrated using two examples, a hotel booking system and a system of transactions with compensations. From a general perspective, our work describes how to open up and exploit constraint solvers as the basis of a coordination engine.

Keywords: interaction constraints, constraint satisfaction, coordination, Reo

1 Introduction

Coordination languages facilitate the exchange of data between components (or services) externally to the operation of those components. One way of describing coordination patterns is by using *interaction constraints* [11], which originated as an approach to implementing Reo connectors [3]. In this approach, off-the-shelf constraint solvers such as Choco [18] and SAT and SMT solvers such as SAT4J [6] and Z3 [16] are used as the basis of the underlying coordination engine. The coordination engine operates in rounds, each of which proceeds by collecting constraints from components and the connector that coordinates them, and then solving the constraints. Components perform blocking reads and writes on ports, which are converted into constraints stating that they want to output or input data. A solution to the constraints describes how data flows between

* This research is supported by the KU Leuven BOF-START project STRT1/09/031 DesignerTypeLab, Belgium, and by the FCT grant SFRH/BPD/91908/2012.

the components, after which some reads and writes may succeed. Each round is considered to be an *atomic* (or *synchronous*) *step*. Between rounds the states of the components and connectors may change.

One problem with the current state-of-the-art is that interaction occurs only *between* constraint solving rounds, not *during* rounds. It is impossible in Reo, for instance, to send data to a component and receive a result from it within the round, as no interaction with external components occurs during a round. This means that all data involved in the constraint is known at the start of a round, and consequently that the kinds of interaction that can be expressed using interaction constraints are limited. The challenge of introducing interaction with external components during a coordination round is that such interaction can produce externally observable behaviour. However, after solving the coordination constraints, these external observations may not correspond to what the coordination pattern aims to achieve. This implies that such actions will need to be undone after a round using rollbacks or compensations.

This paper reports on our work on *interactive interaction constraints*, which enhance a coordination engine to enable interaction during rounds. The underlying constraint solver can invoke external components several times during a round while searching for a valid solution to the coordination constraints. This can be seen as a form of negotiation. This kind of interaction can be incorporated into the visual notation of Reo using special *filter* and *transformer* channels. Filters have a predicate that is used to determine whether data flows through the channel and transformers modify the data passing through the channel by invoking a unary function. In our approach these could involve external interaction. For instance the predicate could consult an external database or by engaging in interaction with a user. Because the actual data that flows through a connector involves the solution of a potentially complex set of constraints, it is never clear when invoking a filter or a transformer whether the connector will commit to the chosen data. Thus, these channels must be implemented using a try-and-compensate mechanism, and hence the whole constraint satisfaction process becomes transactional.

Although originally stemming from research on Reo—indeed, the same visual notation can be used to describe connectors—our approach is much more general, as a greater range of coordination patterns can be implemented. Interactive interaction constraints are based on information unknown at compile time, the expressing a wider variety of coordination patterns than constraints over a fixed set of data types and operators. More generally, our approach falls within the implicit programming paradigm [17], wherein constraints specify the computation and SAT and SMT solvers perform the computation. The contribution of our work to this field is the use of constraint satisfaction to implement coordination patterns. More specifically, this paper deals with the problem of increasing the kinds of external interaction possible during constraint satisfaction.

Organisation of the paper The next section motivates the need for richer interaction model and identifies the main challenges. Section 3 describes the language used to specify constraints, and Section 4 explains how they are solved.

Details of the constraint-solving engine are described in Section 5. Section 6 presents how transactions are achieved with interactive interaction constraints, Section 7 presents related work, and Section 8 concludes.

2 The Need for Interaction

Interaction constraints were introduced as a bridge to providing an efficient and flexible implementation of the Reo coordination language [11]. Previous implementations based on compilation to constraint automata suffered from the problem that the entire behaviour of the connector needed to be known in advance [5]. Implementations based on connector colouring got around this problem, but these were initially not very efficient and were ultimately not very flexible as they were insensitive to data values [9]. Changizi et. al [7] extended the automata-based compilation approach with filters and transformers. These are handled by a SAT/SMT solver, though the choice of filters and transformers is limited to those expressible in the language of the solver. When building an automaton, all solutions for all states need to be found, and thus more work than necessary needs to be done and the approach is inflexible. Jongmans et al. [14] integrated external functionality by generating Java code corresponding to the automata-with-data-constraints model of Reo. The resulting code has an exponential number of formulas, without data transformations, that are checked sequentially. Interaction constraints improved on these implementations by exploiting the flexibility of constraints—again, limited only by the underlying solver—, and by permitting constraints to change dynamically and to be evaluated concurrently and partially, thereby increasing scalability as well [10].

One remaining problem is that the kind of external interaction available in the current interaction constraints-base engine is limited to blocking reads and writes. That is, considering Fig. 1, component Request interacts with the connector by performing a blocking write of some value. At some future time, this value may be accepted by the system and the write will proceed (or it may timeout). Dually, component Confirmed interacts by preforming a blocking read. At some future time, a value will become available and the read proceeds.

From Reo's perspective synchronicity corresponds to atomicity. Thus in Fig. 1 there are three possible ways data can flow (synchronously) through the connector: a request flows from Request via exactly one of $\mathsf{SrchHotel}_i$, which return a possible hotel room booking. Then it flows through filter Approve, which seeks approval from the user, transformer Book, which performs the booking, transformer Invoice, which handles the payment, filter Paid, which occurs when the payment succeeds, and finally to component Confirmed. Due to semantics of the synchronous drain (sd), dataflow through the connector is permitted only if Paid allows the data to pass. Now it is clear that such an atomic step corresponds to all steps of the hotel booking process succeeding. If any step fails, such as when the user does not approve the selection or if the payment is not made, then no flow occurs in the connector at all. To actually implement this requires a lot more than interaction via blocking reads and writes, such as calling and getting

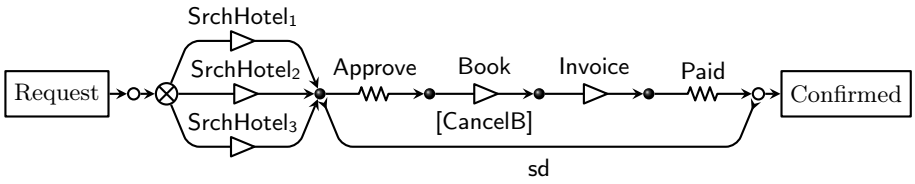


Fig. 1. Example of a Hotel Reservation workflow. Nodes (●) receive data from exactly one of their inputs, and replicate it to all of their outputs. The exclusive router (⊗) is a special node that forwards data to exactly one of its outputs. Transformer channels are represented using a triangle and filter channels using a zig-zag line.

results multiple times from the hotels, interacting with the user checking the payment, and retrying with different possible values from the hotels. These interactions can be realised using special functions and predicates that, whenever queried by the constraint solver, trigger calls to external components. Solving an interactive constraint might require such functions to be executed with different parameters until a valid solution is found. Whenever constraints imposed by individual elements of the connector are not satisfied, some externally observable actions, such as interaction with the user and calls to the booking site, might need to be undone, by performing a rollback or running compensation code.

Exploiting a constraint solver to implement a scenario such as the one above requires that a number of issues are properly handled:

- Constraints corresponding to the channels need to be evaluated in the right order, such as calling Approve before Book, to avoid situations where the solver guesses a value and wastefully calls some external function. External functions should only be given input that derives from concrete initial values.
- The constraint solver may (potentially, wastefully) invoke all of $\mathsf{SrchHotel}_1$, $\mathsf{SrchHotel}_2$, and $\mathsf{SrchHotel}_3$, although only one of them will be accepted. A better strategy is often to try one at a time.
- If an external function that produces an externally observable side-effect is called, such as Book, but subsequently the booking needs to be cancelled, then some compensation/rollback functionality (CancelB) needs to be performed to undo the side-effect.
- Functions that do not transform their data but do produce side-effects can be postponed until after a solution to the constraints has been found, as from the perspective of the constraint solver they are equivalent to a synchronous channel (what happens on both ends is the same).
- External functions and predicates need to be total to avoid blocking the constraint solver. Non-complying functions could easily be wrapped, such as by implementing a time-out and returning a 'no-result' value.
- Within a given round, external functions and predicates need to be deterministic to preserve the consistency of the constraint solving process. Non-complying functions can be memoized to behave as deterministic functions.

3 Coordination via Interaction Constraints

Our previous work on interaction constraints formed the basis of an efficient implementation of the Reo coordination language [11]. In this model, coordination patterns are described using logical formulas defined over two kinds of variables: *synchronisation* variables, which capture whether or not there is dataflow on a given port, and *data* variables, which describe the value that flows, when there is dataflow. Coordination takes place in rounds, and between rounds the constraints can change. From a high level perspective, each round corresponds to an atomic operation, and each solution to the interaction constraints gives the ports that synchronise and the data that flows between them.

Given a collection of ports $\mathcal{X}$, synchronisation variables $x \in \mathcal{X}$ range over booleans, and data variables $\widehat{x} \in \widehat{\mathcal{X}}$ range over a global data set $\mathbb{D}$. Formulas are defined as Dijkstra's guarded commands [12], given by the following grammar:

$$\begin{aligned}
\psi &::= \phi \to s \mid \psi_1\ \psi_2 \mid \top && \text{(formulas)}\\
\phi &::= x \mid P(\widehat{x}) \mid \phi_1 \wedge \phi_2 \mid \neg\phi && \text{(guards)}\\
s &::= \phi \mid \widehat{x} := d \mid \widehat{x}_1 := \widehat{x}_2 \mid \widehat{x}_1 := f(\widehat{x}_2) \mid s_1\,;\,s_2 && \text{(statements)}
\end{aligned}$$

$\top$ is *true*, P is a unary predicate over data variables, and f is a unary total function. Constraint $\psi\ \psi$ is interpreted as $\psi \wedge \psi$, $s; s$ as $s \wedge s$, and $x := y$ as $x = y$. Other logical connectives can be encoded as usual. The grammar for guarded commands enforces data assignments to always be in positive positions, and ensures that the assignment operator $:=$ is asymmetric to capture the direction of dataflow. These features are not exploited in this paper, but required by the predicate abstraction technique described in Section 4.1.

The novel addition of this paper is that functions and predicates need not be built-in functions of the logic and can be implemented externally. Thus, evaluating a function or predicate requires a call outside of the constraint solver. Furthermore, such functions and predicates may be side-effecting, and these side-effects may need to be undone. How this interactive and transactional evaluation of functions and predicates is handled is explained in detail in Section 4.

A Constraint-Based Encoding of Reo Channels As an example of how interaction constraints are used, Table 1 recalls the encoding of the semantics of the most common Reo primitives [11]. For example, the LossySync can have dataflow on b only if data flows on a ($b \to a$), and, whenever both ports have dataflow, the data flowing on a is copied to b ($b \to \widehat{b} := \widehat{a}$). Included in the table are writers and readers: these capture the essential interaction behaviour of components that perform blocking reads and writes. The semantics of connector composition is given by conjunctively joining the constraints of its constituents.

4 Solving Interactive Constraints

As functions and predicates are defined externally, the constraint solving process requires external interaction to provide an interpretation for them as logical

Channel	Representation	Constraints	Channel	Representation	Constraints
Sync	$a \longrightarrow b$	$a \leftrightarrow b$ $b \rightarrow \widehat{b} := \widehat{a}$	LossySync	$a \dashrightarrow b$	$b \rightarrow a$ $b \rightarrow \widehat{b} := \widehat{a}$
SyncDrain	$a \rightarrowtail\!\!\leftarrow b$	$a \leftrightarrow b$	FIFO-E	$a \longrightarrow \boxed{} \longrightarrow b$	$\neg b$
SyncSpout	$a \longleftrightarrow b$	$a \leftrightarrow b$	FIFO-F(d)	$a \longrightarrow \boxed{d} \longrightarrow b$	$\neg a$ $b \rightarrow \widehat{b} := d$
Merger	$a, b \rightarrow c$	$c \leftrightarrow (a \vee b)$ $\neg(a \wedge b)$ $a \rightarrow \widehat{c} := \widehat{a}$ $b \rightarrow \widehat{c} := \widehat{b}$	Replicator	$a \rightarrow b, c$	$a \leftrightarrow b$ $a \leftrightarrow c$ $a \rightarrow \widehat{b} := \widehat{a};$ $\widehat{c} := \widehat{a}$
Filter(P)	$a \xrightarrow{P} b$	$b \rightarrow \widehat{b} := \widehat{a}$ $(a \wedge P(\widehat{a})) \leftrightarrow b$	Transf(f)	$a \xrightarrow{f} b$	$a \leftrightarrow b$ $b \rightarrow \widehat{b} := f(\widehat{a})$
Writer(d)	$\boxed{\mathrm{W}(d)} \longrightarrow a$	$a \rightarrow \widehat{a} := d$	Reader	$\boxed{\mathrm{R}} \longleftarrow a$	$\top$

Table 1. Constraint-based encoding of Reo Channel.

formulas. This section provides a model of how this can be done. Firstly, before constraint solving, constraints are reduced to a boolean formula using a notion of predicate abstraction based on the data dependencies of each predicate. This avoids prematurely computing such predicates and functions. Then, during constraint solving the solver will evaluate predicates and functions on a per-need basis. In addition, compensations need to be performed, but as these are independent of constraint solving, their discussion is postponed until Section 5.

4.1 Pre-Processing

Formulas over boolean and data variables are encoded into formulas only over boolean variables using a notion of predicate abstraction in two steps (full details are available in a separate report [21]):

1. starting from predicates, calculate all data variables that may contribute to its value by tracing back to sources of data; and
2. replace data variables by new boolean variables, one for each predicate that is *reachable*, based on the sets calculated on the previous step.

These steps are illustrated using the examples in Fig. 2. In the first step, every path backwards from a predicate to a data source is determined based on the data assignments in formulas, yielding paths numbered from 1 to 5. Each of these paths is denoted by a new boolean variable. In the second step, data variables are replaced by boolean variables. Each new variable captures whether the associated predicate holds when applied to the given data. For instance, path 2 is represented by variable $\widehat{x}_{R.f}$, which denotes whether $R(f(\widehat{x}))$ holds. In the underlying constraints, $\widehat{x}$ is replaced by 4 new boolean variables $\widehat{x}_P$, $\widehat{x}_Q$,

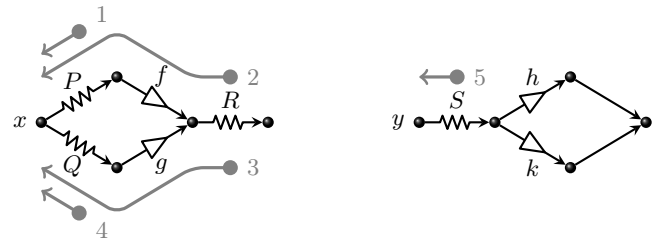


Fig. 2. Simple examples to illustrate predicate abstraction.

$\widehat{x}_{R.f}$ and $\widehat{x}_{R.g}$. Similarly $\widehat{y}$ is replaced by $\widehat{y}_S$, and so forth. Data assignments are modified to work on these new variables. For example, given variables $\widehat{x}_{P.f}$ and $\widehat{y}_{P.f}$, the encoding of $\widehat{y} := \widehat{x}$ includes $\widehat{y}_{P.f} := \widehat{x}_{P.f}$. Given variables $\widehat{x}_{P.f}$ and $\widehat{z}_P$, the encoding of $\widehat{z} := f(\widehat{x})$ includes $\widehat{z}_P := \widehat{x}_{P.f}$. Predicates in formulas are replaced by the corresponding variable: for instance, $P(\widehat{x})$ is replaced by $\widehat{x}_P$. Finally, data assignments $\widehat{x} := d$ are encoded as a conjunction of special constraints, called *external predicates*:

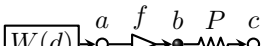
$$\bigwedge_{p \text{ reaches } x} \mathsf{XPred}(p, x, d) \qquad \text{(usage of external predicates)}$$

The reachability mentioned in the formula above is the same as in Fig. 2, and, in this case, p ranges over $R.f$, $R.g$, P and Q. Each $\mathsf{XPred}(p, x, d)$ in this formula is a wrapper guarding the evaluation of $p(d)$. This constraint is equivalent to the one below, but it is evaluated during constraint solving using dedicated functions.

$$\neg x \ \wedge \ \neg y \ \wedge \ \widehat{x}_p = \mathsf{eval}(p(d)) \qquad \text{(interpretation of an external predicate)}$$

where y is the port to which the predicate p is applied, after dropping all the associated functions, and eval performs the computations needed to evaluate a predicate and its associated functions. For instance, the external predicate $\mathsf{XPred}(R.f, x, d)$ is interpreted as $\neg x \ \wedge \ \neg r \ \wedge \ \widehat{x}_{R.f} = \mathsf{eval}(R(f(d))$, where r is the port between the transformer f and the filter R. This means that the value of $\widehat{x}_{R.f}$ only reflects the result of $R(f(d))$ when both x and r have dataflow.

For example, the original and abstracted constraints of the connector on the right are, respectively:



$$\begin{array}{c} a \to \widehat{a} := d \qquad b \to \widehat{b} := f(\widehat{a}) \\ a \leftrightarrow b \quad c \to \widehat{c} := \widehat{b} \quad (b \wedge P(\widehat{b})) \leftrightarrow c \end{array} \quad \Bigg| \quad \begin{array}{c} a \to \mathsf{XPred}(P, b, d) \qquad b \to \widehat{b}_P := \widehat{a}_{P.f} \\ a \leftrightarrow b \qquad (b \wedge \widehat{b}_P) \leftrightarrow c \end{array}$$

Since no predicate reaches c, no variables are created for c and $\widehat{c} := \widehat{b}$ is dropped.

An alternative to using predicate abstraction is to use an encoding into constraints over a fixed domain that are used as hashes for the actual values or functions that need to be computed. Following this line of thought, we are currently investigating the advantages of encoding into integer constraints, making a tradeoff between the number of variables used at the cost of a more expensive underlying constraint solver. These ideas are left as future work.

4.2 Evaluation Model

Fig. 3 presents a model of constraint solving over booleans, following the style of Apt [2], adapted to our setting. These rules rely on two core functions: *propagate* and *satisfy* ($\models$), which will be defined later for external predicates to control when functions and predicates are evaluated.

$$\frac{\langle c_1, \ldots, c_n \ ; \ V_1, x \mapsto \{\top, \bot\}\rangle}{\langle c_1^x, \ldots, c_n^x \ ; \ V_{n+1}^x\rangle \ \ \langle c_1^{-x}, \ldots, c_n^{-x} \ ; \ V_{n+1}^{-x}\rangle}\ (\textsc{branch}) \qquad \begin{array}{l} \textit{where, for } 1 \le i \le n: \\ (c_i^x \ , V_{i+1}^x) = \textit{propagate}(c_i, V_i^x) \ ; \\ (c_i^{-x}, V_{i+1}^{-x}) = \textit{propagate}(c_i, V_i^{-x}) \end{array}$$

$$\frac{\langle c, C \ ; \ V\rangle}{\langle c', C \ ; \ V'\rangle}\ (\textsc{propagate}) \quad \textit{where } (c', V') = \textit{propagate}(c, V)$$

$$\frac{\langle c, C \ ; \ V\rangle}{\langle C \ ; \ V\rangle}\ (\textsc{satisfy}) \quad \textit{if } V \models c$$

$$\frac{\langle c, C \ ; \ V\rangle \ \ \overline{\langle C' \ ; \ V'\rangle}}{\overline{\langle C' \ ; \ V'\rangle}}\ (\textsc{prune}) \quad \textit{if } V \not\models c$$

Fig. 3. Semantics of the constraint solver. $V^x = V[x \mapsto \top]$ and $V^{-x} = V[x \mapsto \bot]$, where $V[x \mapsto b]$ denotes the update of V by mapping x to $\{b\}$, for $b \in \{\bot, \top\}$.

Formally, a CSP over booleans is a pair $\langle C; V\rangle$ of constraints and variable domains. The constraints are initially the conjunctive set of guarded commands of a connector. The variable domains initially map each variable to $\{\top, \bot\}$. Constraint satisfaction proceeds by branching over variables and by simplifying constraints based on the current domains of variables. Branching over a variable x means creating two new CSPs, one assuming x is true and one that x is false. Simplification of constraints is performed by a *propagate* function, which takes a constraint c and a variable domain V and builds, when possible, a simpler constraint c' and a smaller domain variable V' where some variables become instantiated with $\{\top\}$ or $\{\bot\}$. Satisfaction of a constraint c is determined using the operator $\models$, based on the instantiated variables of domain V.

The implementation of *propagate* and *satisfaction* for external predicates $\mathsf{XPred}(P, x, d)$ are as follows:

$$\textit{propagate}(\mathsf{XPred}(P, x, d), V) = \begin{cases} (\top, V) & \textbf{if } V(x) = \bot \textbf{ or } V(y) = \bot \\ (\bot, V) & \textbf{if } V(x) = V(y) = \top \textbf{ and } \mathsf{eval}(P(d)) \notin V(\widehat{x}_P) \\ (\top, V[\widehat{x}_P \mapsto \top]) & \textbf{if } V(x) = V(y) = \top \textbf{ and } \top \in V(\widehat{x}_P) \textbf{ and } \mathsf{eval}(P(d)) \\ (\top, V[\widehat{x}_P \mapsto \bot]) & \textbf{if } V(x) = V(y) = \top \textbf{ and } \bot \in V(\widehat{x}_P) \textbf{ and } \neg\mathsf{eval}(P(d)) \\ (\mathsf{XPred}(P, x, d), V) & \textbf{otherwise} \end{cases}$$

$$V \models \mathsf{XPred}(P, x, d) = \begin{cases} \top & \textbf{if } V(x) = \bot \textbf{ or } V(y) = \bot \\ \widehat{y}_P = \mathsf{eval}(P(d)) & \textbf{if } V(x) = V(y) = \top \\ \textit{unknown} & \textbf{otherwise} \end{cases}$$

Observe that if $V \models c$ returns *unknown*, then neither $V \models c$ nor $V \not\models c$ hold.

The solver will still control when to instantiate any of the variables used by XPred, which are guided using a search strategy (see the next section).

5 Implementation

We have implemented a prototype coordination engine that handles external predicates based on the Choco constraint solver.[1] It focuses on the boolean satisfaction problem of the formulas obtained via predicate abstraction. Contrary to most other SAT and SMT solvers,[2] Choco allows user-defined function and predicates, and the customisation of strategies. These capabilities are exploited to control the evaluation of external predicates. As a running example we use a simple implementation of the Hotel Reservation system (Fig. 1). External predicates are implemented to read from and write to the command line.

5.1 Caching and Compensating

To avoid redoing complex calculations or performing the same query twice, whenever a function or predicate is evaluated its result is cached. This ensures that all functions are deterministic in any given round. Similarly, functions and predicates that require human interaction are executed at most once per argument, per round.

After each round of constraint solving the cache is cleared. During this process the engine checks which values contributed to the solution. Every cached value that did not contribute is reverted using its associated compensation, if it exists.

5.2 Strategies

By default Choco uses a branching strategy that selects the next variable to be analysed based on its current domain size, the number of uninstantiated constraints involving the variable, and the sum of some counters associated with these constraints [18]. For each selected variable, Choco starts by assigning the smallest value (*false* in the case of booleans), increasing it whenever necessary.

We propose the use of an alternative strategy, built using Choco strategy constructors,[3] that selects variables using a fixed order and assigns *false* values before assigning *true*. This order of variables is produced based on the possible paths of dataflow, which have been partially calculated during the pre-processing of the constraints (Section 4.1). The intuition is that if a path with dataflow that satisfies the constraints is found, the external predicates on the remaining paths do not need to be evaluated. More concretely, we use the following guidelines:

- Approximate the data dependency graph and linearise the resulting ordering.
- Branch first over synchronisation variables and only later over the rest.

[1] http://www.emn.fr/z-info/choco-solver/

[2] We also experimented with SAT4J and Z3 solvers, among others.

[3] These constructors are called `AssignVar`, `StaticVarOrder`, and `IncreasingDomain`.

In the Hotel Reservation system, this strategy can avoid the booking, invoicing, and payment of hotels that are not approved. The choice of which function and predicate is evaluated first is non-deterministic: it depends on the order of concatenation of traversals during the linearisation process. In this case, a possible linearisation is: $req \leftarrow h_1 \leftarrow h_1 out \leftarrow hs \leftarrow ap \leftarrow bk \leftarrow inv \leftarrow paid \leftarrow h_3 \leftarrow h_3 out \leftarrow h_2 \leftarrow h_2 out$. Thus the constraint solver will first select and branch the variable $h_2 out$, then the variable h_2, and so on. The data variables are selected only after the synchronous variables. The ports h_i and $h_i out$ are the two ports of the filter $\mathsf{SrchHotel}_i$, and the ports $ap, bk, inv, paid$ succeed the channels Approve, Book, Invoice, and Paid, respectively.

5.3 Scala Implementation

We have developed a set of libraries for the Scala language[4] to easily specify interactive constraints. Scala is fully interoperable with Java, hence our libraries can also be used to define and run connectors using Java.

```
object HotelReservation extends App {

  case class Req(val content:String)

  def srchHotel(i:Int) =
        Function("SearchHotel-"+i){
    case r:Req ⇒ i match {
      case 1 ⇒ List("F1","Ibis","Mercury")
      case 2 ⇒ List("B&B","YHostel")
      case _ ⇒ List("HotelA","HotelB")
    }
  }
  val approve = Predicate("approve"){
    case l:List[String] ⇒
      println("approve: "+l.mkString(",
            ")+". [y,n]")
      readChar() == 'y'
  }
  val book = Function("book"){
    case l : List[String] ⇒
      println("Options: "+l.mkString(", ")+
        ". Which one? (1.."+l.length+")")
      val res = readInt()
      l(res-1)
  }
  val cancelB = Function("cancelB"){
    case x ⇒ println("canceling "+x+".")
  }
  val invoice = Function("invoice"){
    case x ⇒ println("invoice for "+x+".")
  }
  val pay = Predicate("paid"){
    case x ⇒ if (x == "Ibis") {
      println("paid for Ibis")
      true
    }
    else {
      println("not paid for "+x)
      false
    }
  }



  // Connector definition
  val connector =
    writer("req",List(Req("req1"),
          Req("req2"))) ++
    nexrouter("req",List("h1","h2","h3")) ++
    transf("h1","h1o",srchHotel(1)) ++
    transf("h2","h2o",srchHotel(2)) ++
    transf("h3","h3o",srchHotel(3)) ++
    nmerger(List("h1o","h2o","h3o"),"hs") ++
    filter("hs","ap",approve) ++
    sdrain("hs","ap") ++
    transf("ap","bk",book,cancelB) ++
    monitor("bk","inv",invoice) ++
    filter("inv","paid",paid) ++
    reader("paid",5)

  connector.run()
}
```

Listing 1: Scala code for the Hotel Reservation system.

[4] http://www.scala-lang.org

The code for the Hotel Reservation system is presented in Listing 1. The code consists of a single object `HotelReservation`, which defines a `Request` inner class, a method or constant value that returns an instance of a `Predicate` or `Function` for each predicate and function, and a connector defined as the composition of 14 sub-connectors. The sub-connector `monitor` is a channel with a function that has side-effects but does not transform data. Instances of the `Predicate` and `Function` classes can be equally created using class inheritance, defining the methods `check` and `calculate`, respectively. The last line of the listing starts the connector running. This triggers the consecutive execution of rounds until a state with no solutions is reached. The code that interacts with the user via command line is highlighted. The documentation of the API can be found online.[5]

6 Example: Transactional Connectors

This section presents three example connectors that coordinate transactions with compensations. The examples are based on a chain of pairs (F_i, F_i^{-1}), where F_i is some operation and F_i^{-1} is a compensation that undoes the effect of F_i. A successful transaction will pass data through each F_i in succession. If any intermediate step fails, then all compensations up to that point need to be run, in reverse order. Thus, if F_1 and F_2 succeed, but F_3 fails, then F_3^{-1}, F_2^{-1}, and F_1^{-1} need to be run. The first example is based on traditional Reo connectors and external components. Traditionally in Reo, external components operate asynchronously and no external interaction occurs during the constraint solving process. The second example is an adaptation of the first where the asynchronous external components are replaced with synchronous transformer channels that encapsulate the external interaction. The third example internalises the compensation behaviour so that it is only accessible to the engine.

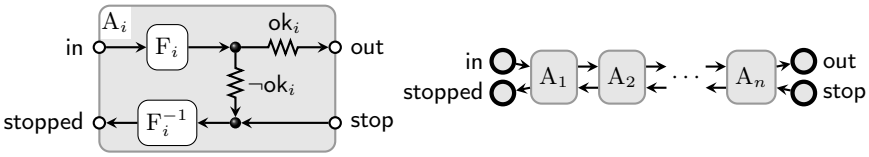


Fig. 4. Asynchronous transactions in Reo.

Fig. 4 presents a traditional Reo connector for coordinating the transaction and its compensation. The left-hand side shows how to coordinate a pair of components (F_i, F_i^{-1}), and the right-hand side shows how these can be composed sequentially to form a larger transaction. Each connector A_i passes data input on port in to F_i. In a subsequent step, F_i returns a result. If this satisfies the filter ok_i, the value is passed to out, otherwise it is passed to F_i^{-1}. In addition, a value

[5] http://people.cs.kuleuven.be/~jose.proenca/reopp/doc

can come from port stop and be passed to F_i^{-1} to indicate that the transaction failed upstream. They key point is that all F_i and F_i^{-1} are asynchronous as far as the Reo connector is concerned. Consequently, each part of the chain runs in a separate round, and nothing guarantees that all parts will run. Thus the connector does not really enforce that the transaction is atomic.

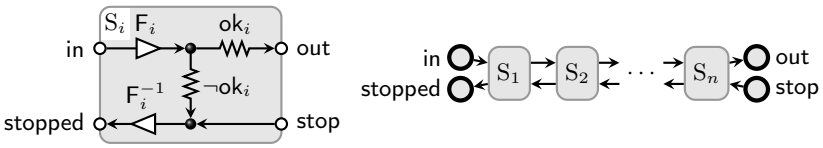


Fig. 5. Synchronous transactions via transformers.

Fig. 5 presents a revised version of the connector that uses synchronous transformer channels F_i and F_i^{-1} instead of components, as in the previous example. These transformers perform the same external operations as their counterparts above, but now they can be handled by the constraint engine. A consequence of the fact that they are synchronous is that the entire connector S_i is synchronous. Indeed, the entire chain in the right-hand side of the figure is synchronous, thus atomicity is regained.

Fig. 6. Synchronous transactions with built-in compensations.

But we can do better. Fig. 6 shows an improved version. In this version, each transformer is modified so that the compensation action F^{-1} is built into the transformer and is run by the engine whenever the transaction fails. The semantics of the connector is that it permits flow on all ports and only on all ports. So the only possible flows permitted are the ones where each transaction succeeds. In cases where this is not possible, such as when some ok_i is false, the engine rollbacks all transformers through which data has passed by running their compensations.

The main differences between the three approaches are summarised as follows: the first approach takes a multiple number of rounds to complete the transaction, while the second and third approaches take only one round; and the running of the compensation is handled by the connector in the first and second approaches, but by the engine in the third approach. Two consequences of having multiple rounds are that the intermediate steps are observable to the external world and the transaction may get stuck in the middle. By compressing everything into a single round, these problems are avoided, and the only

observables are completed transactions (modulo the fact that some actions are undone using compensations). Having the connector handle the running of the compensations is potentially error-prone, even though it introduces a degree of flexibility. Handling compensations within the engine simplifies the connector and shifts responsibility for correctness to the engine.

An alternative approach to using Reo to coordinate (long-running) transactions was presented by Kokash et al. [15]. This resembles the first approach above, though the connector was more complicated as it also permitted the transaction to be cancelled externally.

7 Related Work

Traditional approaches to implementing Reo [5,7,14] are based on pre-computing an automaton describing all future behaviour of the Reo connector. This typically performs more work than is necessary and is rather inflexibility, specifically because it eliminates all intensional information about the connector. Our approach is based on dynamically generating and solving logical constraints [11,8]. This permits more control over intensional aspects during runtime, which allows more refined interaction with external components than was previously possible.

Montanari and Rossi express coordination as a constraint satisfaction problem, in a similar and general way [20]. They view networks as graphs, and use the tile model to distinguish between synchronisation and sequential composition of the coordination pieces. In our approach, we explore a more concrete coordination model, which not only captures the semantics of Reo, but also extends it with external interaction, not found in Montanari and Rossi's work.

Minsky and Ungureanu introduced the Law-Governed Interaction (LGI) mechanism [19], implemented in the Moses toolkit. The mechanism targets distributed coordination of heterogeneous actors, enforcing laws that are defined using constraints in a Prolog-like language. The main innovation is the enforcement of laws by certified controllers that are not centralised. Their laws, as opposed to our approach, are not global, allowing them to achieve good performance, while compromising the scope of the constraints. Communication between actors governed by laws and actors outside LGI is possible, but not the execution of side-effecting code while checking constraints.

Abreu and Fiadeiro explore the coordination of interactions in service-oriented systems using SRML, a Service Modelling Language [1]. Services are linked with each other by connecting ports and referring to the protocol used to connect them. SRML operate at the abstraction level of business modelling, using asynchronous message passing and explicitly supporting service discovery. Our approach differs from theirs as in our work orchestration is guided by a constraint solving process, interaction with services is transactional, and our global constraints express more coordination patterns.

Our work falls within the implicit programming paradigm. Köksal et al. proposed similarly to integrate the power of declarative SAT/SMT solvers non-intrusively into sequential, imperative programs [17]. In contrast to this work,

our approach targets coordination languages, and depends upon a constraint solver enhanced with interaction as a side-effect.

Faltings et al. [13] explore interactive constraint satisfaction, which is a framework for open constraint satisfaction in a distributed environment that enables constraints to be added on-the-fly. There are a number of key differences between our work and theirs. The first is that our work focuses on the coordination of components, separating the computation and coordination aspects, whereas they aim purely at constraint satisfaction. Secondly, our work allow functions and predicates to be defined externally to the constraint solver, whereas their approach allows instead on-the-fly constraint generation. The former requires the management of compensations for any external interaction that is not committed to, where as the latter does not.

8 Conclusion and Future Work

This paper expanded upon the use of constraint solving as the basis of an engine for coordinating components by introducing support for external interaction during the constraint solving process. For this to make sense in terms of externally observable behaviour, certain calls to functions and predicates required rollback or compensation to undo their effect. This means that the rounds of constraint solving become transactional. In contrast to previous implementation approaches for Reo connectors, our approach increases the degree of external interaction possible in a connector, and transactional behaviour can be expressed much more concisely as a consequence. In effect, we have lifted Reo's notion of synchrony as atomicity to synchrony as transactional atomicity. This means that Reo's synchronous connector semantics can be used to (visually) express transactional behaviour, and interactive interaction constraints supply the bridge to the underlying implementation.

As future work we first plan to experiment with heuristics to better guide the constraint solver. One approach is to avoid the pre-processing phase by reducing the original constraints over arbitrary data values to an SMT problem of a simple theory. This approach will allow some external functions and predicates to be internalised within the engine, thereby avoiding the need for rollback/ compensation. Secondly, we will combine the techniques described in this paper with our earlier work on partial connector colouring [10], which optimises the constraint satisfaction process by admitting partial solutions to the constraints. This optimisation was experimentally demonstrated to increase scalability of the engine. Finally, we plan to integrate our implementation into the existing open source ECT tools for Reo [4], thereby making it available to developers.

References

1. J. Abreu and J. L. Fiadeiro. A coordination model for service-oriented interactions. In D. Lea and G. Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2008.

2. K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
3. F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.
4. F. Arbab, C. Koehler, Z. Maraikar, Y.-J. Moon, and J. Proença. Modeling, testing and executing Reo connectors with the Eclipse Coordination Tools. In *Proceedings of FACS*, 2008.
5. C. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
6. D. L. Berre and A. Parrain. The Sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010.
7. B. Changizi, N. Kokash, and Arbab. A constraint-based method to compute semantics of channel-based coordination models. In *ICSEA: proceedings of the International Conference on Software Engineering Advances*, 2012.
8. D. Clarke. Coordination: Reo, nets, and logic. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 5382 of *Lecture Notes in Computer Science*, pages 226–256. Springer, 2007.
9. D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Science of Computer Programming*, 66(3):205–225, May 2007.
10. D. Clarke and J. Proença. Partial connector colouring. In M. Sirjani, editor, *COORDINATION*, volume 7274 of *Lecture Notes in Computer Science*, pages 59–73. Springer, 2012.
11. D. Clarke, J. Proença, A. Lazovik, and F. Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76, 2011.
12. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.
13. B. Faltings and S. Macho-Gonzalez. Open constraint programming. *Artificial Intelligence*, 161(1-2):181–208, 2005.
14. S.-S. T. Q. Jongmans, F. Santini, M. Sargolzaei, F. Arbab, and H. Afsarmanesh. Automatic code generation for the orchestration of web services with reo. In F. D. Paoli, E. Pimentel, and G. Zavattaro, editors, *ESOCC*, volume 7592 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2012.
15. N. Kokash and F. Arbab. Applying Reo to service coordination in long-running business transactions. In S. Y. Shin and S. Ossowski, editors, *SAC*, pages 1381–1382. ACM, 2009.
16. A. S. Köksal, V. Kuncak, and P. Suter. Scala to the power of Z3: Integrating smt and programming. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 400–406. Springer, 2011.
17. A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. *SIGPLAN Not.*, 47(1):151–164, Jan. 2012.
18. F. Laburthe and N. Jussien. *CHOCO solver documentation*, August 2012. http://sourceforge.net/projects/choco/files/choco/2.1.5/choco-2.1.5/choco-doc-2.1.5.pdf.
19. N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, 2000.
20. U. Montanari and F. Rossi. Modeling process coordination via tiles, graphs, and constraints. In *3rd Biennial World Conference on Integrated Design and Process Technology*, volume 4, pages 1–8, 1998.
21. J. Proença and D. Clarke. Solving data-sensitive coordination constraints. CW Reports CW637, Department of Computer Science, KU Leuven, February 2013.