Connectors: Refinement and test case generation using Z3

Sihan Wu[1] and Xueyi Tan[2]

[1] Yuanpei College, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing 100871, China,
2300017743@stu.pku.edu.cn,
[2] School of Mathematical Sciences, Peking University, No.5 Yiheyuan Road, Haidian District, Beijing 100871, China,
2300010816@stu.pku.edu.cn

Abstract. test. test.

Keywords: test, test, test

1 Introduction

2 Preliminaries

2.1 The coordination language Reo

Reo is a channel-based exogenous coordination language where complex coordinators, called connectors, are compositionally built out of simpler ones [2]. Exogenous coordination imposes a purely local interpretation on each inter-component communication, engaged in as a pure I/O operation on each side, that allows components to communicate anonymously, through the exchange of untargeted passive data.

Complex connectors in Reo are organized in a network of primitive connectors with well-defined behavior, called *channels*, such as synchronous channels, FIFO channels, etc. A connector provides the protocol that controls and organized the communication, synchronization and cooperation among the components/services that they interconnect. Each channel has two *channel ends*: *source* ends and *sink* ends. A source channel end accepts data into the channel, and a sink channel end dispenses data out of the channel. It is possible for the ends of a channel to be both sinks or both sources. Reo places no restriction on the behavior of a channel and thus allows an open-ended set of different channel types to be used simultaneously together. Each channel end can be connected to at most one component instance at any given time. Some simple channel types in Reo are shown in sectionModeling Connectors.

Complex connectors are constructed by composing simpler ones via the *join* and *hiding* operations. Channels are joined together at nodes. A node consists of a set of channel ends. The set of channel ends coincident on a node A is disjointly

partitioned into the sets $\mathrm{Src}(A)$ and $\mathrm{Snk}(A)$, denoting the sets of source and sink channel ends that coincide on A, respectively. Nodes are categorized into *source*, *sink* and *mixed nodes*, depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of the two. The hiding operation is used to hide the internal topology of a component connector. The hidden nodes can no longer be accessed or observed from outside. A complex connector has a graphical representation, called a *Reo circuit*, which is a finite graph where the nodes are labeled with pair-wise disjoint, non-empty sets of channel ends, and the edges represent their connecting channels. The behavior of a Reo circuit is formalized by means of the data-flow at its sink and source nodes. Intuitively, the source nodes of a circuit are analogous to the input ports, and the sink nodes to the output ports of a component, while mixed nodes capture its hidden internal details.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a replicator. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected non-deterministically. A sink node, thus, acts as a non-deterministic merger. A mixed node nondeterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes.

2.2 Z3

Z3 [7] if an efficient SMT (Satisfiability Modulo Theories) solver freely available from Microsoft. It has been used in various software verification and analysis applications. Z3 expands to deciding the satisfiability (or dully the validity) of first order formulas with respect to combinations of theories such as: arithmetic, bit-vectors, arrays, and uninterpreted functions. Given the data and time constraints of connectors, it allows us to verify the satisfiability of properties or refinement relations. Z3 provides bindings for several programming languages. In this paper, we use *Z3 python-bindings* to construct the models and carry out refinement checking.

2.3 The UTP observational model

Specification and unification phases can be observed in every scientific discipline. During specification sicentists focus on some narrowly defined phenomenon and aim to discover the laws governing that phenomenon, which, typically, are special cases of a more general theory. Unification aims at unifying theory that clearly and convincingly exlains a broader range of phenomena. A proposed unification

of theories often receives spectacular confirmation and reward complementary to the prediction of new discoveries or by the development of new technologies. However, a unifying theory is usually complementary to the theories that it links, and does not seek to replace them. In [4] Hoare and He aim at unification in computer science. They saw the need for a comprehensive theory of programming that

- includes a convincing approach to the study of a range of languages in which computer programs may be expressed,
- must introduce basic concepts and properties that are common to the whole range of programming methods and languages,
- must deal separately with the additions and variations that are particular to specific groups of related languages,
- should aim to treat each aspect and feature in the simplest possible fashion and in isolation from all the other features with which it may be combined or confused.

Our theory of Reo connectors originated out of these motivations for unification. In this section we introduce the theory of UTP designs and the observational model for connectors briefly. More details about UTP can be found in [4].

A theory of designs UTP adopts the relational calculus as the foundation to unify various programming theories. All kinds of specifications, designs and programs are interpreted as relations between an initial observation and a subsequent (intermediate, stable or final) observation of the behavior of their execution. Program correctness and refinement can be represented by inclusion of relations, and all laws of the relational calculus are valid for reasoning about correctness.

Collections of relations form a theory of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

During observations it is usual to wait for some initial transient behavior to stabilize before making any further observation. In order to express this, we introduce two varaibles *ok*, *ok'*: **Boolean**. The variable *ok* stands for a successful initialization and the start of a communication. When *ok* is **false**, the communication has not started, so no observation can be made. The variable *ok'* denotes the observation that the communication has either terminated or reached an intermediate stable state. The communication is divergent when *ok'* is **false**.

In our semantic model, the observational semantics for a Reo connector is described by a design, i.e., a relation expressed as $P \vdash Q$, where P is the predicate specifying the relationship among the observations on the source nodes of the connector, and Q is the predicate specifying the condition that should be satisfied by the observations on the sink nodes of the connector. Such a design $P \vdash Q$ is defined as follows:

Definition 1. *A design is a pair of predicates $P \vdash Q$, where neither predicate contains ok or ok', and P has only unprimed variables. It has the following*

meaning:

$$P \vdash Q =_{df} (ok \wedge P \Rightarrow ok' \wedge Q)$$

A design predicate represents a pre/post-condition specification. The separation of precondition from post-condition allows us to write a specification that has a more general precondition than simply the domain of the relation used as a specification. Implementing a design, we are allowed to assume that the precondition holds, but we have to satisfy the post-condition. Moreover, we can rely on the system having been started, but we must ensure that it terminates. If the precondition does not hold, or the system does not start, we are not committed to establish the post-condition nor even to make the system terminate. Any non-trivial system requires a facility to select between alternatives according to the truth or falsehood of some guard condition b. The restriction that b contains no primed variables ensures that it can be checked before starting either of the actions. The conditional expression $P \lhd b \rhd Q$ describes a system that behaves like P if the initial value of b is **true**, or like Q otherwise. It can be defined as follows:

Definition 2. *The conditional expression is defined as follows:*

$$P \lhd b \rhd Q =_{df} (\boldsymbol{true} \vdash (b \wedge P \vee \neg b \wedge Q))$$

The sequential composition P; Q denotes a system that first executes P, and when P terminates executes Q. This system is defined via existential quantification to hide its intermediate observation, and to remove the variables that record this observation from the list of free variables of the predicate. To accomplish this hiding, we introduce a fresh set of variables v_0 to denote the intermediate observation. These fresh variables replace the input variables v of Q and the output variables v' of P, thus the output alphabet of P ($out_\alpha P$) and the input alphabet of Q ($in_\alpha Q$) must be the same.

Definition 3. *Let* $out_\alpha P = \{v'\}$, $in_\alpha Q = \{v\}$, *then*

$$P(in : u; out : v'); Q(in : v; out : w) =_{df} \exists v_0 \cdot P(in : u; out : v_0) \wedge Q(in : v; out : w)$$

If the conditional and sequential operators are applied to designs, the result is also a design. This follows from the laws below.

$$(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) = ((P_1 \lhd b \rhd P_2) \vdash (Q_1 \lhd b \rhd Q_2))$$
$$(P_1 \vdash Q_1); (P_2 \vdash Q_2) = (P_1 \wedge \neg(Q_1; \neg P_2) \vdash (Q_1; Q_2))$$

A reassuring result about a design is the notion of refinement, which is defined via implication. In UTP, we have the well-known property that under refinement, preconditions are weakened and post-conditions are strengthened. This is established by the following definition:

Definition 4. $[(P_1 \vdash Q_1) \sqsubseteq (P_2 \vdash Q_2)]$ ***iff*** $[P_1 \Rightarrow P_2] \wedge [P_1 \wedge Q_2 \Rightarrow Q_1]$

The theory of designs forms a complete lattice, with miracle $\top_{\mathbf{D}}$ as the top element, and abort $\bot_{\mathbf{D}}$ as the bottom element.

$$\top_{\mathbf{D}} =_{df} (\mathbf{true} \vdash \mathbf{false}) \text{ and } \bot_{\mathbf{D}} =_{df} (\mathbf{false} \vdash \mathbf{true})$$

The meet and join operations in the lattice of designs are defined as follows, which represent internal (non-deterministic, demonic) and external (angelic) choices.

$$(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) = (P_1 \wedge P_2 \vdash Q_1 \vee Q_2)$$
$$(P_1 \vdash Q_1) \sqcup (P_2 \vdash Q_2) = (P_1 \vee P_2 \vdash ((P_1 \Rightarrow Q_1) \wedge (P_2 \Rightarrow Q_2)))$$

Finally, iteration is expressed by means of recursive definitions. A recursively defined design has as its body a function on designs; as such, it can be seen as a (monotonic) function on pre/post-condition pairs (X, Y), and iteration is defined as the least fixed point of the monotonic function.

The theory of designs can be taken as a tool for representing specifications, programs, and, as in the following sections, connectors.

Observational model for connectors Connectors describe the coordination among components/services. We use in_R and out_R to denote what happens on the source nodes and the sink nodes of a connector $\mathbf{R}$, respectively, instead of using unprimed variables for initial observations (inputs) and primed variables for subsequent ones (outputs) as in [4]. Thus, the alphabet, i.e., the set of all observationcapturing variables, used in this paper is different from that for a design in [4]. The signature gives the rules for the syntax for denoting the elements of the theory. Note that in modeling of connectors not every possible predicate is useful. It is necessary to restrict ourselves to predicates that satisfy certain healthiness conditions which embody aspects of the model being studied: e.g., a predicate describing a connector that produces output without being started should be excluded from the theory ($\neg ok \wedge out_{\mathbf{R}} = \langle d, 1 \rangle$). In addition, the results of the theory must match the expected observations in reality, e.g., merging the sink node of a connector that fails to terminate with the source node of any other connector must always lead to non-termination of the whole composed connector (this is the technical motivation for introducing ok, ok'). The subset of predicates that meet our requirements are called designs.

For an arbitrary connector $\mathbf{R}$, the relevant observations come in pairs, with one observation on the source nodes of $\mathbf{R}$, and one observation on the sink nodes of $\mathbf{R}$. For every node N, the corresponding observation on N is given by a (finite or infinite) timed data sequence, which is defined as follows:

Let D be an arbitrary set, the elements of which are called data elements. The set DS of data sequences is defined as

$$DS = D^*$$

i.e., the set of all sequences $\alpha = (\alpha(0), \alpha(1), \alpha(2), \ldots)$ over D. Let $\mathbb{R}_+^*$ be the set of non-negative real numbers, which in the present context can be used to

represent time moments[3]. For a sequence s, we use $|s|$ to denote the length of s, and if s is an infinite sequence, then $|s| = \infty$. Let $\mathbb{R}_+^*$ be the set of sequences $a = (a(0), a(1), a(2), \ldots)$ over $\mathbb{R}_+$, and for all $a = (a(0), a(1), a(2), \ldots)$ and $b = (b(0), b(1), b(2), \ldots)$ in $\mathbb{R}_+^*$, if $|a| = |b|$, then

$$\begin{array}{lll} a < b & \text{iff} & \forall 0 \leq n < |a|.a(n) < b(n) \\ a \leq b & \text{iff} & \forall 0 \leq n < |a|.a(n) \leq b(n) \end{array}$$

For a sequence $a = (a(0), a(1), a(2), \ldots) \in \mathbb{R}_+^*$, and $t \in \mathbb{R}_+$, $a[+t]$ is a sequence defined as follows:

$$a[+t] = (a(0) + t, a(1) + t, a(2) + t, \ldots)$$

Furthermore, the element $a(n)$ in a sequence $a = (a(0), a(1), a(2), \ldots)$ can also be expressed in terms of derivatives $a(n) = a^{(n)}(0)$, where $a^{(n)}$ is defined by

$$a^{(0)} = a, a^{(1)} = (a(1), a(1), \ldots), a^{(k+1)} = (a^{(k)})^{(1)}$$

The set TS of time sequences is defined as

$$\begin{aligned} TS = \{a \in \mathbb{R}_+^* \mid & (\forall 0 \leq n < |a|.a(n) < a(n+1)) \\ & \wedge (|a| = \infty \Rightarrow \forall t \in \mathbb{R}_+^*.\exists k \in \mathbb{N}.a(k) > t)\} \end{aligned}$$

Thus, a time sequence $a \in TS$ consists of increasing and diverging time moments $a(0) < a(1) < a(2) < \cdots$.

For a sequence a, the two operators a^R and $\overrightarrow{a}$ denote the reverse and the tail of a, respectively, defined as:

$$a^R = \begin{cases} () & \text{if } a = () \\ (a')^{R^\frown}(a(0)) & \text{if } a = (a(0))^\frown a' \end{cases}$$

$$\overrightarrow{a} = \begin{cases} () & \text{if } a = () \\ a' & \text{if } a = (a(0))^\frown a' \end{cases}$$

where $^\frown$ is the concatenation operator on sequences. The concatenation of two sequences produces a new sequence that starts with the first sequence followed by the second sequence.

The set TDS of timed data sequences is defined as $TDS \subseteq DS \times TS$ of pairs $\langle \alpha, a \rangle$ consisting of a data sequence α and a time sequence a with $|\alpha| = |a|$.

[3] Here we use the continuous time model for connectors since it is expressive and closer to the nature of time in the real world. For example, for a FIFO1 channel, if we have a sequence of two inputs, the time moment for the output should be between the two inputs (The semantics we use for FIFO1 in this paper disallows output and input to happen at the same time moment.). If we use a discrete time model like $\mathbb{N}$, and have the first input at time point 1, then the second input can only happen at a time point greater than 2, i.e., at least 3. But in general, this is not explicit for the input providers.

Similar to the discussion in [3], timed data sequences can be alternatively and equivalently defined as (a subset of) $(D \times \mathbb{R}_+)^*$ because of the existence of the isomorphism

$$\langle \alpha, a \rangle \mapsto (\langle \alpha(0), a(0) \rangle, \langle \alpha(1), a(1) \rangle, \langle \alpha(2), a(2) \rangle, \ldots)$$

The occurrence (i.e., taking or writing) of a data item at some node of a connector is modeled by an element in the timed data sequence for that node, i.e., a pair of a data element and a time moment.

3 Modeling Connectors

3.1 Basic connectors

We first develop the design model for a set of basic Reo connectors, i.e., channels.

3.2 Timer connectors

3.3 Probabilistic connectors

The specification of channels with probabilistic behavior can be captured by the disjunction or conjunction of different predicates about time and data distributions. We consider four types of probabilistic channels: *message-corrupting synchronous channel*, *randomized synchronous channel*, *probabilistic lossy synchronous channel*, and *faulty FIFO1 channel*. Specifications of other primitive channels are ignored here and can be found at [1].

CptSync: The message-corrupting synchronous channel $-\not{p}\rightarrow$ is a synchronous channel which has an extra parameter p compared with the primitive synchronous channel. The delivered message can be corrupted with probability p. Hence, if a data item flows into the channel through the source end, then the correct data value will be obtained at the sink end with probability $1 - p$ and a corrupted data value $\perp$ will be obtained with probability p.

RandomSync: The randomized synchronous channel $\xrightarrow{rand(0,1)}$ can generate a random number $b \in \{0, 1\}$ with equal probability when it is activated through an arbitrary write operation on its source end, and this random number will be taken on the sink end simultaneously.

ProbSync: The message transmitted by the probabilistic synchronous channel $-\overset{q}{-}\rightarrow$ can get lost with a certain probability q. It can also act like a *Sync* channel and the message will be delivered successfully with probability $1 - q$.

FaultyFIFO1: The messages flowing into a faulty FIFO1 channel $\cdot\overset{r}{\cdot}\square\rightarrow$ can get lost with probability r when it is inserted into the buffer. In this case, the buffer remains empty. It can also behave as a normal *FIFO1* channel when the insertion of data into the buffer is successful with probability $1 - r$.

3.4 Composing connectors

3.5 Refinement of connectors

4 Refinement checking and test case generation in Z3

4.1 Test case for connectors

4.2 Refinement checking

References

1. The source code., https://github.com/Zhang-Xiyue/Prob-Reo
2. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (06 2004)
3. Arbab, F., Rutten, J.J.: A coinductive calculus of component connectors. In: International Workshop on Algebraic Development Techniques. pp. 34–55. Springer (2002)
4. Hoare, C.A.R.: Unified theories of programming. In: Mathematical methods in program development, pp. 313–367. Springer (1997)
5. Meng, S.: Connectors as designs: The time dimension. In: Margaria, T., Qiu, Z., Yang, H. (eds.) Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China. pp. 201–208. IEEE (2012), http://doi.ieeecomputersociety.org/10.1109/TASE.2012.36
6. Meng, S., Arbab, F., Aichernig, B.K., Aştefănoaei, L., de Boer, F.S., Rutten, J.: Connectors as designs: Modeling, refinement and test case generation. Science of Computer Programming 77(7), 799–822 (2012), https://www.sciencedirect.com/science/article/pii/S0167642311001006, (1) FOCLASA'09 (2) FSEN'09
7. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
8. Nawaz, M.S., Sun, M.: Using pvs for modeling and verification of probabilistic connectors. In: Hojjat, H., Massink, M. (eds.) Fundamentals of Software Engineering. pp. 61–76. Springer International Publishing, Cham (2019)
9. Sun, M.: Towards formal modeling and verification of probabilistic connectors in coq xiyue zhang (2018), https://api.semanticscholar.org/CorpusID:85541303
10. Zhang, X., Hong, W., Li, Y., Sun, M.: Reasoning about connectors using coq and z3. Science of Computer Programming 170, 27–44 (2019), https://www.sciencedirect.com/science/article/pii/S0167642318304076