Data abstraction in coordination constraints*

José Proença[1,2] and Dave Clarke[2]

[1] HASLab / INESC TEC, Universidade do Minho, Portugal
[2] iMinds-DistriNet, Dep. Computer Science, KU Leuven, Belgium
{jose.proenca,dave.clarke}@cs.kuleuven.be

Abstract. This paper studies complex coordination mechanisms based on constraint satisfaction. In particular, it focuses on data-sensitive connectors from the Reo coordination language. These connectors restrict how and where data can flow between loosely-coupled components taking into account the data being exchanged. Existing engines for Reo provide a very limited support for data-sensitive connectors, even though data constraints are captured by the original semantic models for Reo.
When executing data-sensitive connectors, coordination constraints are not exhaustively solved at compile time but at runtime on a per-need basis, powered by an existing SMT (satisfiability modulo theories) solver. To deal with a wider range of data types and operations, we abstract data and reduce the original constraint satisfaction problem to a SAT problem, based on a variation of predicate abstraction. We show soundness and completeness of the abstraction mechanism for well-defined constraints, and validate our approach by evaluating the performance of a prototype implementation with different test cases, with and without abstraction.

1 Introduction

Coordination languages describe how data can be exchanged among components, focusing on the glue code and abstracting away the computations performed by components. An ongoing trend for these languages over the last years leans towards more expressive coordination models, aiming at more compact and manageable representations of complex behaviour than basic models such as Linda.

This paper focuses on coordination models whose glue code is given by connectors, expressed as logical constraints. Using constraints to describe how data flows in a connector has been investigated, for example, for the BIP [3,4] and the Reo [1,7,15] coordination languages. Constraints have also been used to describe desirable properties of process algebras, such as Bruni's et al.'s compensable processes [5]. In order to keep the problem of producing and executing connectors tractable, only properties that bear no computation are captured by the constraints. These are then analysed using off-the-shelf constraint solvers.

Engines for BIP and Reo have incorporated various properties into their coordination constraints, such as history of the connector, some notions of priority, and simple data restrictions. These coordination-related properties are encoded

* This research is supported by the FCT grant SFRH/BPD/91908/2012.

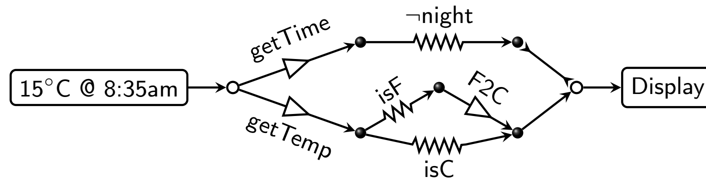

Fig. 1. Filtering communication between a sensor and a display based on data.

as boolean formulas or as formulas over a decidable theory, which can be analysed by a given off-the-shelf constraint solver. An implementation of BIP [4] relies on BDD libraries for constraint solving, and some Reo implementations rely on SAT solvers [7] and on Computer Algebra Systems [6]. Using constraint solvers to execute connectors brings more flexibility than compiling them into state machines that list all coordination patterns, since it supports larger connectors, and changes to the system have a low impact on performance.

This paper exploits the usage of constraints to describe coordination patterns that use complex (and possibly undecidable) data predicates. This is achieved by decoupling the evaluation of complex data predicates from the constraint solving problem. We propose a method that encodes formulas over data structures into a boolean formula, by incorporating in the final formula the results of operations over data that influence coordination. We show that this method is sound and complete with respect to a class of constraints that covers all Reo connectors over data that we encountered in the literature. An earlier version of this work with the detailed proofs can be found in a companion technical report [16]. Our technique has been recently exploited to introduce interaction between the solver and external components during constraint solving [15]. More generally, our approach falls within the implicit programming paradigm [14], wherein constraints specify the computation and SAT and SMT solvers perform the computation. Our contribution to this field is the use of constraint satisfaction to implement coordination patterns. More specifically, this paper deals with the problem of increasing the complexity of data used to coordinate components.

We use the Reo coordination language as the source of coordination constraints, based on our previous work [7]. Reo is a synchronous graph-based visual language wherein complex connectors are built out of simpler primitive connectors. Each primitive connector imposes restrictions on how and where data can flow, and the behaviour of a composite connector is given by the composition of the constraints of all primitives involved. A connector evolves on a per-round basis, and in each round data flows atomically through some of the ports of this connector, based on its combined constraints. After each round the state of the connector may change, resulting in new constraints.

The Reo connector depicted in Fig. 1 has a data producer and a data consumer that displays a given temperature value. The data producer tries to publish a temperature value of 15°C, measured at 8:35am. This producer is connected to two *transformer* channels, depicted with a triangle, that extract the

time and the temperature attributes from the data. These values are then filtered by *filter* channels, depicted with zig-zag lines, that allow data to flow if the associated predicate holds for the data flowing. For example, isF checks if the temperature is measured in Fahrenheit degrees. The result from the ¬night filter acts as a barrier to the temperature value, allowing data to flow to the display only if it is daytime. The connector evolves atomically, in the sense that data only flows from the producer to the consumer if it is daytime, if the temperature is in Fahrenheit or in Celsius, and if the display accepts data. If the predicates ¬night and isC hold but the display cannot receive data, then the producer is not allowed to publish the value. This reflects the role of data-constraints for coordination, where the mere attempt to send data influences dataflow.

Summarizing, we model and execute synchronous connectors where arbitrary data operations can influence dataflow—even when these cannot be handled by SMT solvers—, by using a SAT solver after a pre-analysis of the coordination constraints. These data operations can be described, for example, using Java methods. We show that our approach is sound and complete, and we compare the overhead cost of decoupling the data analysis against more traditional approaches where the data constraints are directly solved by an SMT solver. Our small benchmark shows that using data values and operations that can be encoded into complex integer calculations can be more efficiently handled using our methodology. However, when they can be encoded with simple integer expressions the performance of using the SMT solver directly is sometimes similar or better than our approach, depending on the number of new boolean variables introduced during the encoding into a boolean formula.

The rest of this paper is organised as follows. Section 2 motivates our approach. Section 3 presents a constraint-based semantics for Reo. Section 4 describes predicate abstraction. Section 5 evaluates the performance of abstracting over data. Section 6 discusses related work and Section 7 concludes our paper.

2 Motivation

When viewing coordination as constraints, the decision of what and where data can flow is made using constraint solving techniques. More precisely, a connector imposes a set of constraints, which evolve during the lifetime of the connector, whose solutions describe the dataflow through the connector. Finding these solutions is an NP problem, wich can be solved using off-the-shelf SAT and SMT solvers. By doing this the expressivity of coordination constraints becomes tighly coupled with the expressivity of the constraint solver. For example, if a connector wants to filter all time references based on a predicate night, the constraint solver needs to represent constraints over time references. This paper proposes an approach that allows operations over data values to be performed outside the underlying constraint solver, allowing functions and predicates to be defined in a more conventional programming language such as Java.

The main challenge when implementing an engine that executes connectors such as the one in Fig. 1 is to decide which data should flow in each of the ports,

taking into account operations over arbitrary data types, including temperature values and time references. We consider two main approaches to this challenge:

SMT The time references and temperature values are represented as integers that are used by an off-the-shelf SMT solver to find a solution. This encoding into integers can be done, for example, by representing the time in minutes and combining it with the value of the temperature using simple arithmetic operations. This restricts the expressivity of the constraints to the language of the SMT solver.

SAT The data constraints are reduced to a simpler constraint over boolean variables using an abstraction technique, and solved using an off-the-shelf SAT solver. The actual values flowing through the ports are calculated based on solutions from the SAT solver. In our example a possible boolean solution could say that only the predicates $\neg\mathsf{night}$ and isC hold, and that there is flow in all three ports. Based on this solution we can infer that the value $\mathsf{8{:}35am}$ flows on the port connected to the display.

Not all solutions to an abstract constraints are guaranteed to produce a solution to the original constraints. Therefore we focus on a subset of constraints that provide this guarantee. For example, we consider the connector from Fig. 1 to be ill-defined without the data producer, since a solution for the abstract constraints would not clearly map to a solution in the original formula.

3 Coordination as constraints

Connectors are viewed as a set of constraints representing valid coordination patterns, following our previous work [7]. Each port has a boolean variable $x \in \mathcal{X}$ indicating presence of dataflow and a data variable $\widehat{x} \in \widehat{\mathcal{X}}$ indicating what data flows. Coordination evolves in rounds: in each round the coordinated components contribute to the constraints of the connector, a solution for these constraints is found, and both the connector and components are updated accordingly.

3.1 Guarded commands

When compared with the original formulation of coordination constraints for Reo [7], we use an (asymmetric) attribution operator for data variables instead of equality, and we allow attributions to be only in positive positions (they can never be negated). The resulting data causality is exploited in our abstraction technique and in the definition of well-defined connectors, but it is does not modify the semantics of connectors. The requirement of having assignments in positive positions facilitates the analysis of connectors, while reflecting the concept of connectors as structures where data flows through. Formulas are represented by Dijkstra's guarded commands [8].

$$\begin{array}{llr} \psi & ::= \ \phi \rightarrow s \ \mid \ \psi_1 \ \psi_2 \ \mid \ \top & \text{(formulas)} \\ \phi & ::= \ x \ (\in \mathcal{X}) \ \mid \ P(\widehat{x}) \ \mid \ \phi_1 \wedge \phi_2 \ \mid \ \neg \phi & \text{(guards)} \\ s & ::= \ \phi \ \mid \ s_1 \wedge s_2 \ \mid \ \widehat{x} := d \ (\in \mathbb{D}) \ \mid \ \widehat{x} := \widehat{y} \ \mid \ \widehat{x} := f(\widehat{y}) & \text{(statements)} \end{array}$$

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

Table 1. Channel Encodings.

Synchronous variables $x \in \mathcal{X}$ range over booleans and *data variables* in $\widehat{\mathcal{X}} = \{\widehat{x} \mid x \in \mathcal{X}\}$ range over a global data set $\mathbb{D}$. Each synchronous variable corresponds to exactly one port of a Reo connector. $\top$ is *true*, $P \in \mathbb{P}$ is a unary predicate over data variables, and $f \in \mathbb{F}$ is a unary total function. A guarded command $\phi \to s$ is interpreted as $\neg\phi \vee s$, $\psi\ \psi'$ as $\psi \wedge \psi'$, and $\widehat{x} := \widehat{y}$ as $\widehat{x} = \widehat{y}$. The other logical connectives for guards can be encoded as usual.

Definition 1 (solution). *A* solution *to a formula ψ defined over ends $\mathcal{X}$ is a mapping σ from $\mathcal{X}$ to $\{\top, \bot\}$, and from $\widehat{\mathcal{X}}$ to data values $\mathbb{D}$, such that σ satisfies ψ, regarded as a boolean expression, according to the satisfaction relation $\sigma \models \psi$ defined below. Each predicate symbol P and function symbol f have an associated interpretation, denoted by $\mathcal{I}(P)$ and $\mathcal{I}(f)$, such that $\mathcal{I}(P) \subseteq \mathbb{D}$ and $\mathcal{I}(f) \subseteq \mathbb{D}^2$.*

$\sigma \models \top$	*always*	$\sigma \models x$	*iff* $\sigma(x) = \top$
$\sigma \models \widehat{x} := d$	*iff* $\sigma(\widehat{x}) = d$	$\sigma \models \neg\psi$	*iff* $\sigma \not\models \psi$
$\sigma \models \widehat{x} := \widehat{y}$	*iff* $\sigma(\widehat{x}) = \sigma(\widehat{y})$	$\sigma \models \psi_1 \wedge \psi_2$	*iff* $\sigma \models \psi_1$ *and* $\sigma \models \psi_2$
$\sigma \models \widehat{x} := f(\widehat{y})$	*iff* $(\sigma(\widehat{y}), \sigma(\widehat{x})) \in \mathcal{I}(f)$	$\sigma \models P(\widehat{x})$	*iff* $\sigma(\widehat{x}) \in \mathcal{I}(P)$

3.2 Reo as constraints

Table 1 presents the formulas of some of the most common Reo primitives [7]. It includes a writer that produces a data value d and reader that receives any data value, which are used to abstract away the behaviour of more complex components. We write ψ_c to denote the current formula imposed by a connector c. Composition of a connector is simply given by the conjunction of their formulas.

The formula ψ_{ni} below constrains the connector on the left of Fig. 2, a simplified version of the connector in Fig. 1.

$$x \to \widehat{x} := \mathsf{8{:}35am} \quad x \leftrightarrow y \quad y \to \widehat{y} := \mathsf{DST}(\widehat{x}) \quad (y \wedge \neg \mathsf{night}(\widehat{y})) \leftrightarrow z \quad z \to \widehat{z} := \widehat{y}$$

A possible solution for ψ_{ni} is $\{x, y, z \mapsto \top; \widehat{x} \mapsto \mathsf{8{:}35am}; \widehat{y}, \widehat{z} \mapsto \mathsf{9{:}35am}\}$, assuming DST adds one hour, and that $\neg\mathsf{night}(\mathsf{9{:}35am})$ holds. This solution states that x, y, z have dataflow, $\mathsf{8{:}35am}$ flows through x, and $\mathsf{9{:}35am}$ flows throw y and z.

3.3 Well-defined formulas

A well-defined formula is a formula to which our predicate abstraction can be applied. More precisely, a well-defined formula must have solutions that produce only *well-defined routes*, where each route is a set of data assignments derived from a given solution. Well-definedness of a route reflects (1) the absence of loops, (2) the absence of multiple assignments to a single variable, and (3) the existence of a data value at the end of each tree of assignments. For example, the following two formulas are ill-defined: $(a \wedge b) \to \widehat{a} := \widehat{b} \quad a \to \widehat{a} := 5$ and $a \to (\widehat{a} := \widehat{b} \wedge \widehat{b} := \widehat{a})$. The first assigns $\widehat{a}$ to $\widehat{b}$ and to 5 when $a \wedge b$ holds, which could be fixed by replacing the second guard by $a \wedge \neg b$. The second assigns $\widehat{a}$ and $\widehat{b}$ to each other, creating a loop of data assignments. Both formulas have routes that violate condition (3), which could be fixed by extending them with the guarded command $\top \to \widehat{b} := 7$.

Definition 2 (route). *A* route r *of a formula* ψ *is a set of assignments associated to a solution* $\sigma \models \psi$*, given by* $\mathsf{route}_\sigma(\psi)$ *defined below.*

$$\begin{array}{ll}
\mathsf{route}_\sigma(\phi \to s) & = \begin{cases} \mathsf{route}_\sigma(s) & \text{if } \sigma^*(g) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathsf{route}_\sigma(\psi_1 \ \psi_2) & = \mathsf{route}_\sigma(\psi_1) \cup \mathsf{route}_\sigma(\psi_2) \\
\mathsf{route}_\sigma(\phi) & = \emptyset \\
\mathsf{route}_\sigma(\widehat{x} := d) & = \{\widehat{x} \mapsto d\} \\
\mathsf{route}_\sigma(\widehat{x} := \widehat{y}) & = \{\widehat{x} \mapsto \widehat{y}\} \\
\mathsf{route}_\sigma(\widehat{x} := f(\widehat{y})) & = \{\widehat{x} \mapsto \widehat{y}\} \\
\mathsf{route}_\sigma(s_1 \wedge s_2) & = \mathsf{route}_\sigma(s_1) \cup \mathsf{route}_\sigma(s_2)
\end{array}
\qquad
\begin{array}{ll}
\textit{where:} & \\
\sigma^*(x) & = \sigma(x) \\
\sigma^*(\neg\phi) & = \neg(\sigma^*(\phi)) \\
\sigma^*(\phi_1 \wedge \phi_2) & = \sigma^*(\phi_1) \wedge \sigma^*(\phi_2) \\
\sigma^*(P(\widehat{x})) & = \sigma(\widehat{x}) \in \mathcal{I}(P)
\end{array}$$

Notation. $\mathsf{routes}(\psi)$ represents the set of all $\mathsf{route}_\sigma(\psi)$ for any σ, and $\mathsf{route}_\top(\psi)$ the set of all assignments in ψ. Then for every $r \in \mathsf{routes}(\psi)$, $r \subseteq \mathsf{route}_\top(\psi)$.

Definition 3 (well-definedness). *A route* r *is* well-defined *if the conditions below hold. A formula* ψ *is* well-defined *if all its routes are well-defined.*

1. *The transitive closure of* r *is not reflexive (no loops).*
2. *Each variable* $\widehat{x}$ *is assigned at most once in* r *(single assignment).*
3. *If* $(\widehat{x} \mapsto t) \in r$*, then* $t \in \mathbb{D}$ *or exists* t' *such that* $(t \mapsto t') \in r$ *(data source).*

Given a well-defined route it is always possible to calculate the data values flowing on this route. This is intuitively done by copying data starting from the data values, and using the functions extracted from guarded commands with

guards that evaluate to true. In the formula ψ_{ni} defined in Section 3.2, and using the solution σ presented there, $\mathsf{route}_\sigma(\psi_{ni})$ returns $\{\widehat{z} \mapsto \widehat{y}\,;\, \widehat{y} \mapsto \widehat{x}\,;\, \widehat{x} \mapsto \mathsf{8{:}35am}\}$. This route can be used to retreive back the values of $\widehat{y}$ and $\widehat{z}$, knowing that $\widehat{y} := \mathsf{DST}(\widehat{x})$, which can be inferred from ψ_{ni} and σ.

In practice, connectors with well-defined formulas need to explicitly mention what data values can be sent by producers; data cannot "created" during constraint solving. Two concerns emerge from this formulation of well-definedness. First, it seems unnatural to build a route from a given solution σ, and to use this route later to discover what values should flow through the route, since this is already given by σ. Second, checking well-definedness (as it is) requires iterating over all solutions. Our first observation is that $\mathsf{route}_\sigma(\cdot)$ does not use the values flowing on the ports: only the synchronisation variables and the validity of the predicates. The abstraction technique described later will provide exactly this information. Regarding the cost of verifying well-definedness, we chose to test sufficient (yet not necessary) conditions for being well-defined. We dedicate the next subsection to this. Furthermore, our data abstraction technique (cf. Section 4) will not produce invalid behaviour from ill-defined connectors; in the worse case it may fail to find the next step.

3.4 Verifying well-definedness

We provide a simple procedure to guarantee well-definedness, which does not cover all well-defined formulas. We address each of the three conditions in Definition 3 separately, and informally discuss the correctness of our procedure.

Loop free Instead of searching for loops in routes from $\mathsf{routes}(\psi)$, we do so in $\mathsf{route}_\top(\psi)$. Since every route is a subset of $\mathsf{route}_\top(\psi)$, these will also be loop free. An example of a loop-free formula that will be wrongly identifed as having loops is $a \to \widehat{b} := \widehat{c} \quad \neg a \to \widehat{c} := \widehat{b}$, since the mutual data dependency between $\widehat{b}$ and $\widehat{c}$ is guarded by a variable a that guarantees that the loop never occurs.

When considering formulas from traditional Reo primitive connectors, the direction of dataflow is fixed. It is still possible to create Reo connectors that yield wrongly identifed loops, but we find these to be complex and unnatural.

Single assignment We guarantee each variable to be uniquely assigned by construction. More precisely, we provide a condition that guarantees that the composition (conjunction) of two formulas preserves the single-assignment property. Intuitively two formulas are *pluggable* if they assign different variables.

Definition 4 (read and write variables, pluggable). *We say $\widehat{x}$ is a* read *variable in ψ if either $(\widehat{y} := \widehat{x}) \in \mathsf{route}_\top(\psi)$ or $(\widehat{y} := f(\widehat{x})) \in \mathsf{route}_\top(\psi)$, and is a* write *variable in ψ if $(\widehat{x} := t) \in \mathsf{route}_\top(\psi)$, for some f, $\widehat{y}$, and t. Write $?\psi$ and $!\psi$ to denote all read and write variables of ψ, respectively. Two formulas ψ_1 and ψ_2 are* pluggable *if:*

$$!\psi_1 \cap !\psi_2 \;=\; \emptyset.$$

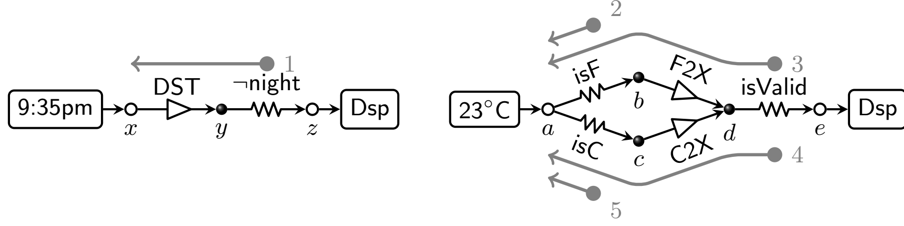

Fig. 2. Calculating dependencies of predicates; DST updates the time according to the daylight saving time, and F2X and C2X create a structure X that it verified by isValid.

By composing only pluggable formulas the effort of verifying the single-assignment property is restricted to only smaller formulas of primitive connectors. All formulas from Table 1 obey the single-assignment property.

Data source We guarantee routes of a formula to always end up in a data value also by construction, by requiring (1) formulas to be pluggable and (2) each primitive formula ψ_p to use only data variables with dataflow. More precisely, every solution $\sigma \models \psi_p$ must obey $\widehat{x} \in \mathsf{var}(\mathsf{route}_\sigma(\psi_p)) \Rightarrow \sigma(x)$, where $\mathsf{var}(\cdot)$ returns the variables present in a route. Finally, we also require (3) all read variables to be write variables in the global formula ψ, that is, $\widehat{x} \in ?\psi \Rightarrow \widehat{x} \in !\psi$.

All formulas in Table 1 obey requirement (2): in all solutions of these formulas if a variable $\widehat{x}$ is written or read then x is set to true. Dropping the guard b in the formula of the Sync channel, for example, would break this property, since $\widehat{b}$ could be read even when b is false. The third requirement is violated every time the SyncSpout is connected to a channel via a shared port x, since $\widehat{x}$ will be a read variable but not a write variable. This can be solved without violating other requirements simply by using a variation of the SyncSpout channel that always outputs a constant value. In fact, we do not know any system modelled in Reo that uses the data value produced by the SyncSpout channel.

4 Data abstraction

This section describes how to encode formulas over data into boolean formulas. This is done in two phases: (1) the dependencies for each predicate are calculated by tracing back the provenience of data, and (2) new boolean variables replace the existing data variables, used to dictate which predicates hold.

Fig. 2 illustrates the dependency analysis for predicates. From trace 3, for example, we deduce that isValid depends on the evaluation of isValid(F2X(23°C)). By evaluating the traces 1 to 5 the data values are no longer needed when searching for valid solutions. This section will describe how to transform formulas—such as the one in Section 3.2—into formulas over booleans—such as the one below. The expression within square brackets is replaced by its evaluation. Observe that z does not have any data variable, since it does not affect any predicate.

$$x \rightarrow \widehat{x}_{\mathsf{ni.dst}} := [\mathsf{night}(\mathsf{DST}(8{:}35\mathrm{am}))] \quad x \leftrightarrow y \quad y \rightarrow \widehat{y}_{\mathsf{ni}} := \widehat{x}_{\mathsf{ni.dst}} \quad (y \wedge \neg\widehat{y}_{\mathsf{ni}}) \leftrightarrow z$$

4.1 Precomputed domain invariants

Write $P.f_1.f_2 \ldots f_n$ to denote a predicate $P \in \mathbb{P}$ with an associated sequence of functions that have to be evaluated before the predicate. Define:

$$(P.f_1.\cdots.f_n) \circ f = \begin{cases} \mathsf{Error} & \text{if } f \in \{f_1, \ldots, f_n\} \\ P.f_1.\cdots.f_n.f & \text{otherwise} \end{cases}$$

and write $\{P_1, \ldots, P_n\} \,\bar{\circ}\, f$ to denote $\{P_1 \circ f, \ldots, P_n \circ f\} \setminus \{\mathsf{Error}\}$. Note that every function in a connector is considered unique.

For each port $x \in \mathcal{X}$ in a formula ψ we define its *domain invariant* $\mathcal{D}_x$ as the set of predicates and functions that can be reachable, intuitively captured by the 5 traces in Fig. 2. More precisely, each $\mathcal{D}_x$ is the smallest set of predicates such that $\rho(\psi)$ holds, where $\rho(\cdot)$ is defined as:

$$\begin{array}{rcc} \rho(P(\widehat{x})) = \mathcal{D}_x \supseteq \{P\} & \rho(\phi \to s) = \rho(\phi) \land \rho(s) & \rho(\psi_1\ \psi_2) = \rho(\psi_1) \land \rho(\psi_2) \\ \rho(\widehat{x} := \widehat{y}) = \mathcal{D}_y \supseteq \mathcal{D}_x & \rho(\phi_1 \land \phi_2) = \rho(\phi_1) \land \rho(\phi_2) & \rho(\neg\psi) = \rho(\psi) \\ \rho(\widehat{x} := f(\widehat{y})) = \mathcal{D}_y \supseteq (\mathcal{D}_x \,\bar{\circ}\, f) & \rho(s_1 \land s_2) = \rho(s_1) \land \rho(s_2) & \rho(_) = \mathit{true}. \end{array}$$

Domain invariants are always finite sets because the definition of $\bar{\circ}$ prevents the application of the same function twice. Well-definedeness does not prevent this duplication because it relies on $\mathsf{routes}(\cdot)$, while $\rho(\cdot)$ relies on all assignments.

The formula ψ_{ni} for the left connector of Fig. 2, presented in Section 3.2, yields the following domain invariants.

$$\mathcal{D}_x = \{\mathsf{ni.dst}\} \qquad \mathcal{D}_y = \{\mathsf{ni}\} \qquad \mathcal{D}_z = \emptyset$$

We write ni and dst as shorthands for night and DST, respectively. These domain invariants are indeed the smallest solution for the constraints given by $\rho(\psi_{ni})$, namely $\mathcal{D}_x \supseteq (\mathcal{D}_y \,\bar{\circ}\, \mathsf{dst})$ and $\mathcal{D}_y \supseteq \{\mathsf{ni}\}$. Applying the same reasoning for the connector on the right of Fig. 2 we can conclude that $\mathcal{D}_a = \{\mathsf{isF}, \mathsf{F2X.isValid}, \mathsf{C2X.isValid}, \mathsf{isC}\}$. The remaining domain invariants can be calculated in a similar way.

4.2 Predicate abstraction

This subsection formalises the encoding from a formula ψ into a new boolean formula, such as the one exemplified right before Section 4.1.

Let $[P.f_1.\cdots.f_n(d)] = P(f_1(\ldots(f_n(d))))$, where $n \geq 0$ and $d \in \mathbb{D}$.[3] The function $[\cdot]$, defined below, receives a formula ψ over arbitrary data types in $\mathbb{D}$ and returns a new formula over booleans, i.e., data variables in $[\psi]$ range over booleans. This transformation is a variant of *predicate abstraction* [9].

$$\begin{array}{rl} [\phi \to s] = [\phi] \to [s] & [x] = x \qquad [P(\widehat{x})] = \widehat{x}_P \qquad [\neg\psi] = \neg[\psi] \\ [\psi_1\ \psi_2] = [\psi_1]\ [\psi_2] & [\widehat{x} := d] = \bigwedge_{P \in \mathcal{D}_x} \widehat{x}_P := [P(d)] \\ [\phi_1 \land \phi_2] = [\phi_1] \land [\phi_2] & [\widehat{x} := \widehat{y}] = \bigwedge_{P \in (\mathcal{D}_x \cap \mathcal{D}_y)} \widehat{x}_P := \widehat{y}_P \\ [s_1 \land s_2] = [s_1] \land [s_2] & [\widehat{x} := f(\widehat{y})] = \bigwedge_{P \in \mathcal{D}_x, (P \circ f) \in \mathcal{D}_y} \widehat{x}_P := \widehat{y}_{P \circ f} \end{array}$$

[3] More precisely, $[P.f_1.\cdots.f_n(d)]$ iff $(n = 0) \land (d \in \mathcal{I}(P))$ **or** $\exists_{d_1,\ldots,d_n \in \mathbb{D}} \cdot (d_1 \in \mathcal{I}(P)) \land (\forall_{i \in \{1,\ldots,n\}} \cdot (d_{i+1}, d_i) \in \mathcal{I}(f_i))$, where $d_{n+1} = d$.

Predicates and functions are computed during the encoding of data assignments $\widehat{x} := d$. Each of these assignments originates a new variable $\widehat{x}_P$ for every $P \in \mathcal{D}_x$ given by the domain invariants, explained before, such that $\widehat{x}_P \leftrightarrow P(\widehat{x})$. Hence the number of new variables depends on the size of the domain invariants. Ports with an empty domain invariant will not have variables in the abstract formula, and ports that can affect n predicates will have at least n new variables.

4.3 Soundness and completeness

Our main claim is that every solution for a well-defined formula ψ can be found by finding a solution for its predicate abstraction $[\psi]$. This requires the abstraction function $[\cdot]$ to be *sound* and *complete*. Soundness means that every solution σ of ψ must also be a solution of $[\psi]$, after mapping each data assignment to the assignments of the new data variables as follows.

$$\begin{array}{rcll} [\sigma] & = & \{\widehat{x}_P \mapsto [P(\sigma(\widehat{x}))] & \mid \widehat{x} \in \mathsf{dom}(\sigma) \cap \widehat{\mathcal{X}}, P \in \mathcal{D}_x\} \\ & \cup & \{x \quad \mapsto \sigma(x) & \mid x \in \mathsf{dom}(\sigma) \cap \mathcal{X}\} \end{array}$$

Completeness means that every solution of $[\psi]$ must be the abstraction of at least one solution in ψ. Both proofs of soundness and completeness rely on the definition of $\rho(\cdot)$ and $[\cdot]$, and completness requires formulas to be well-defined.

Theorem 1 (Soundness). $\sigma \models \psi \;\Rightarrow\; [\sigma] \models [\psi]$.

Proof. Start by fixing the domain invariant of every port. The proof follows by induction on the structure of formulas, applied to guards, statements, and to guarded commands. Soundness of the conjunction of guarded commands follows directly from the soundness of guarded commands and the definition of $\models$. □

Theorem 2 (Completeness).
*ψ is well-defined **and** $\sigma' \models [\psi] \quad \Rightarrow \quad \exists_\sigma \cdot (\sigma \models \psi) \wedge (\sigma' = [\sigma])$.*

Proof. We build a solution σ for ψ based on σ', knowing that ψ is well defined.

1. Start with the smallest σ such that $\forall x \in (\mathsf{dom}(\sigma') \cap \mathcal{X}) \cdot \sigma(x) = \sigma'(x)$.
2. Assume (so far) that, for every $\widehat{x}_P \in \mathsf{dom}(\sigma')$, $\sigma'(\widehat{x}_P) = \top \Rightarrow \sigma(\widehat{x}) \in \mathcal{I}(P)$. Calculate $r = \mathsf{route}_{\sigma'}(\psi)$ using the assumption above to resolve $\sigma^*(P(\widehat{x}))$.
3. The route r is well-defined (based on the assumption mentioned above), hence it is possible to calculate the data flowing in every port along these routes. Starting from each data value in r, apply the assignments and functions induced by r to calculate these data values.

Observe that not all x and $\widehat{x}$ need to have a value assigned by σ. Extending σ with assignments of variables not in σ will not modify $\sigma \models \psi$, since the validity of the route is enough to guarantee satisfaction.

The assumption introduced in (2) can be shown based on the the construction of σ and on the routes induced by σ' on both ψ and its abstraction. □

5 Evaluation

We validate our approach by applying predicate abstraction to five connectors with varying sizes. All but the last connector use integers as the data domain, allowing us to compare the performance of our techniques against the direct usage of an SMT solver. The goal of this evaluation is to understand the overhead of pre-computing the operations over data before invoking a SAT solver, possibly introducing a larger number of variables. The last connector uses a Java data structure instead of integers, showing that the performance is not compromised when dealing with other data domains, and to emphasise that our abstraction technique supports more expressive data-sensitive connectors.

Our prototype implementation uses the Z3 SMT solver[4] to solve expressions with booleans and integers. Z3 is a high-performance theorem prover with an incorporated SMT solver being developed by Microsoft. In our experiments we use only integer arithmetic, although Z3 supports many other theories.

We evaluate our test cases using the following solver configurations.

Z3 Z3 is used to solve the original data constraints.

[**Z3**] The original constraints are encoded into boolean constraints using predicate abstraction, and solved with Z3; and a solution for the original constraint is produced.

Our prototype implementation is developed using the Scala language,[5] which produces Java binary classes, can import Java libraries, and supports functional programming. The source code and our benchmarks can be found online.[6] To integrate Z3 with Scala we use the Scala^Z3 libraries developed at EPFL [13].

5.1 Test cases

Our approach is evaluated using five test cases: the temperature connector from our motivating example, a set of transactional functions in sequence and in parallel, and two variants of an approval system.

Temperature This connector (Fig. 3) is based on our motivating example from Section 1. The data value is regarded as an integer, the transformer channels perform simple arithmetic operations, and the predicates use simple inequalities.

Transactional functions We define a transactional function to be a tuple $\langle \mathsf{pre}, \mathsf{f}, \mathsf{f}^{-1}, \mathsf{post} \rangle$, where f is the main function, f^{-1} is a compensation that must be applied to undo f, and pre and post are pre- and post-conditions of f. The test cases consist of the sequential and parallel composition of transactional functions (Fig. 4). Data enters the connector via the *in* port and exits either via *out* if both conditions hold, or via *stopped* otherwise. The *stop* port propagates the stopping signal in the sequential composition. Predicates and functions use again simple arithmetic operations and inequalities, and are setup so that all transactions succeed except the last transaction in the sequence.

[4] http://research.microsoft.com/projects/z3

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

[6] http://is.gd/reopp

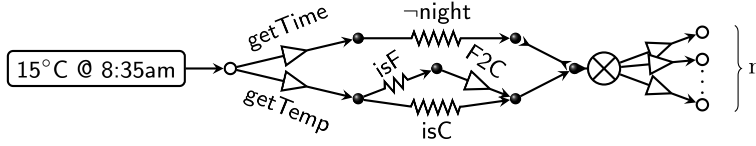

Fig. 3. Temperature connector connected to n outputs.

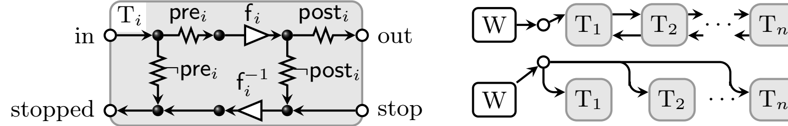

Fig. 4. Connectors with transactional functions.

Approval system The approval system (Fig. 5) captures the merging from several applicants, each publishing their classification. Each applicant provides a tuple of 5 integers, consisting of a unique identifier and 4 classifications from 0 to 20. The predicates isApproved and isDenied check if these ratings are within a certain thresholds, encoded in two variants: (1) as expressions that require arithmetic operations to convert back and forward tuples (based on conversions to and from base 21), and (2) as Java methods over tuples of elements.

5.2 Results and discussion

The constraints for our test cases are solved using a 8-core 2.4 GHz Intel Xeon desktop with 16 GB RAM running Ubuntu Linux. Each measurement was performed 10 times, and the average was used (Fig. 6). The time covers the building of formulas, the solving of constraints, and the calculation of the dataflow, performed at runtime. In the first and last two graphs a log-log scale is used.

Z3 uses SAT solving to iteratively search for solutions to more complex theories, in our case the theory of integers. Our abstraction also reduces a more complex problem to a SAT problem. Probably due to internal optimisations in Z3, and the usage of more efficient memory operations, its performance is in some cases similar or better than predicate abstraction.

The transactional functions running in parallel exhibit the best results for predicate abstraction compared to Z3. This is partially justified by the small

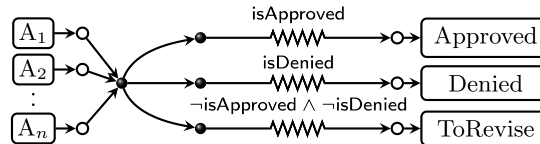

Fig. 5. Approval n-ary connector.

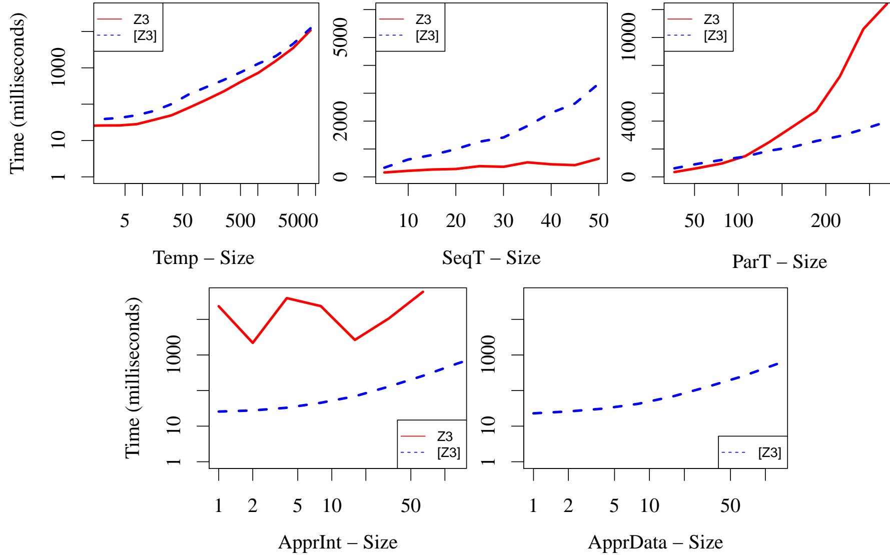

Fig. 6. Performance evaluation of our five parameterised test case connectors.

number of variables added during predicate abstraction, and because pre-compilation of the predicates is not more expensive since all predicates need to be evaluated also for Z3. Conversely, the number of variables in the sequence of transactional functions is very high, reducing the performance of [Z3]. The unexpected variations of time for Z3 in the approval system are probably a consequence of the complexity of its predicates and of the high valued integers involved.[7] This complexity has little impact when using predicate abstraction, which performs faster and more consistently. Furthermore abstraction allows the usage of Java data structures and operations, allowing a reimplementation of the approval system in a more structured way and without loss of performance.

Summarising, we conjecture that scenarios with complex data functions and predicates benefit from our predicate abstraction mechanism, scenarios with a large number of simple functions and predicates and no complex calculations benefit from using SMT solvers, and in scenarios with a smaller number of data operations the difference of performance is small. Using predicate abstraction can also be beneficial in scenarios with a large number of predicates and functions, provided the encoding does not produce a large number of variables, as in ParT.

6 Related work

A recent attempt to coordinate Erlang actors uses special actors with associated Reo connectors [11]. That work illustrates the need to support data con-

[7] A number of runs for Z3 timed out after 5 min and were left out of this benchmark.

straints, since there was no automatic tool to generate coordination code from Reo connectors. From the verification perspective, model checking techniques for Reo connectors exist based on mCRL2 and on its representation of data structures [12]. Regarding implementations of Reo, Changizi et. al [6] 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. Their process of building an automaton searches for all solutions for all states. Our work is more flexible by considering only one state and solution at a time during execution, and it supports formulas with data operations outside the underlying solver. Jongmans et al.[10] orchestrated web services based on Reo, and 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. Our approach improved on these implementations by exploiting the flexibility of constraints, not limited by the expressivity of the underlying constraint solver, and by identifying a suitable set of connectors for our abstraction techniques.

Predicate abstraction is a technique used to reduce complex problems to simpler ones while preserving some relevant properties [9]. This technique is commonly used for model checking [2], where concrete states of a system are mapped to a smaller set of abstract states based on a set of predicates. New predicates can be added to expand the set of abstract states, in a process called *abstraction refinement*. Our variation of predicate abstraction modifies an original system by replacing operations over data by boolean variables that reflect properties over this data. Instead of refining the abstraction until a solution is found (also experimented outside this paper), we identify systems that do not require abstraction refinement.

Our work falls within the implicit programming paradigm. Köksal et al. proposed to integrate the power of SAT/SMT solvers non-intrusively into sequential, imperative programs [14]. In contrast, our approach targets coordination languages, and addresses the expressivity of data-sensitive synchronous systems.

7 Conclusions

This paper explores an execution model for data-sensitive connectors based on predicate abstraction. We exploit the fact that the vast majority of connectors includes concrete data values to precompute the predicates used by the connector before solving the data constraints. A simple analysis of the constraints yields which predicates should be computed for each variable, and the original predicates are abstracted to boolean variables holding the precomputed results. Our approach is shown to be sound and complete for well-defined connectors. As a result, one can specify and run the coordination layer between components using high-level constraints that inspect and manipulate data offered by producers.

This abstraction technique has been exploited to investigate new interaction mechanisms between the solver and external components during constraint solv-

ing, by using functions and predicates that perform interaction [15]. An interesting direction for future work is to encode generic data constraints into formulas over simple theories, instead of boolean formulas, making a tradeoff between relying on more powerful solvers and avoiding the potential increase of variables.

References

1. 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.
2. T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In J.-P. Katoen and P. Stevens, editors, *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 2280, pages 158–172. Springer, 2002.
3. S. Bliudze and J. Sifakis. Synthesizing Glue Operators from Glue Constraints for the Construction of Component-Based Systems. In S. Apel and E. Jackson, editors, *Software Composition*, LNCS, pages 51–67, Berlin / Heidelberg, 2011. Springer.
4. M. Bozga, M. Jaber, N. Maris, and J. Sifakis. Modeling dynamic architectures using dy-bip. In T. Gschwind, F. D. Paoli, V. Gruhn, and M. Book, editors, *Software Composition*, LNCS 7306, pages 1–16. Springer, 2012.
5. R. Bruni, C. Ferreira, and A. K. Kauer. First-order dynamic logic for compensable processes. In Sirjani [17], pages 104–121.
6. B. Changizi, N. Kokash, and Arbab. A constraint-based method to compute semantics of channel-based coordination models. In *International Conference on Software Engineering Advances*, 2012.
7. D. Clarke, J. Proença, A. Lazovik, and F. Arbab. Channel-based coordination via constraint satisfaction. *Science of Computer Programming*, 76, 2011.
8. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, Aug. 1975.
9. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification*, LNCS 1254, pages 72–83. Springer, 1997.
10. 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 *European Conference on Service-Oriented and Cloud Computing*, pages 1–16, 2012.
11. R. Khosravi and H. Sabouri. Using coordinated actors to model families of distributed systems. In Sirjani [17], pages 74–88.
12. N. Kokash, C. Krause, and E. P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing*, 24(2):187–216, 2012.
13. 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, *Conference on Automated Deduction*, LNCS 6803, pages 400–406. Springer, 2011.
14. A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. *SIGPLAN Not.*, 47(1):151–164, Jan. 2012.
15. J. Proença and D. Clarke. Interactive interaction constraints. In R. De Nicola and C. Julien, editors, *COORDINATION*, LNCS 7890, pages 211–225. Springer, 2013.
16. J. Proença and D. Clarke. Solving data-sensitive coordination constraints. CW Reports CW637, Department of Computer Science, KU Leuven, February 2013.
17. M. Sirjani, editor. *Coordination Models and Languages, COORDINATION, Stockholm, Sweden, June, 2012. Proceedings*, LNCS 7274. Springer, 2012.