Compiling Protocols to Promela and Verifying their LTL Properties

Benjamin Lion
CWI, Amesterdam, The Netherland
B.Lion@cwi.nl

Samir Chouali
Univ. Bourgogne Franche-Comté,
FEMTO-ST Institute/CNRS, Besançon,
France
schouali@femto-st.fr

Farhad Arbab
CWI, Amesterdam, The Netherland
Farhad.Arbab@cwi.nl

ABSTRACT

Component-based systems can be modeled as black-box, standalone components, coordinated by an interaction protocol. In this paper we focus on developing reliable protocols, specified in a coordination language where their design, implementation, and property verification rely on the exact same model. In this context, to construct complex protocols compositionally, we use Reo, a powerful channel-based coordination language with well-defined semantics. We extend the current set of back-end languages supported by Reo's compiler with Promela, the specification language of the model checker SPIN. The automatic generation of Promela code from Reo specification enables system simulation and verification of LTL properties of Reo models using SPIN.

KEYWORDS

Component-based systems, Reo circuits, verification, Promela/SPIN

1 INTRODUCTION

Development of a complex distributed system relies, generally, on assembling existing COTS (Commercial off-the-shelf) components (or services), such that their interactions after the composition manifest that of the required system. A major difficulty in developing such systems involves the correctness of the interaction protocol among their constituent heterogeneous components. As such, correct behavior of individual component is insufficient to imply the correctness of the composed system, because such system's behavior depends on the protocol that coordinates its components. Precise specification and correctness of coordination protocols, thus, play a crucial role in construction of complex distributed systems.

Coordination languages, like Reo [3], offer powerful "glue-code" to specify interaction protocols. Reo offers a non-traditional interaction centric paradigm for design of concurrent system. It is a coordination language, that supports exogenous composition of components and services via explicit, composable, pure interaction specifications. The Reo model of a concurrent system consists of a set of components that interact with each other through a set of externally specified protocol constructs, called, connectors, that constrain synchronization and exchanges of data among those components. Primitive and complex connectors are specified in a graphical or textual syntax [11]. Many semantic formalisms exist for formal specification of the behavior of Reo connectors [14].

In this paper we propose to use Reo to guarantee a precise specification of complex protocols, and our main purpose is to propose a formal approach for verifying their correctness. For that, we propose to exploit the SPIN model checker [13] to verify safety and liveness properties of the specified protocols.

Our choice of SPIN is motivated by the fact that it is one of the most employed model checkers, in both industrial and academic communities, for detecting software defects in concurrent system designs. SPIN has been applied to everything from the verification of complex call processing software that is used in telephone exchanges, to the validation of intricate control software for interplanetary spacecraft. In SPIN, a formal specification is built using Promela, which is an imperative language that resembles C in, .e.g, variable and type declaration.

To ensure the correctness of translation from Reo to Promela, we present sufficient conditions to compile a protocol into a Promela program. We first introduce the notion of input/output designation, to direct the data flow in the protocol. Then, we define an environment as a set of processes that the protocol coordinates, and write the functional response of the protocol from the environment as a formula in guarded commands form. We finally present the translation of the guarded commands to Promela, to verify their LTL properties. We show the correctness of our translation by considering the behavior of the protocol specified in Reo and Promela. We demonstrate the relevance of our work in a case study involving design and verification of a railway component-based system.

Proofs of propositions and theorem are accessible at [1].

2 REO

Reo is a channel-based coordination language used for compositional construction of coordinated systems. In this section, we explain the terminology used in Reo, together with a graphical syntax used to draw Reo circuit. We then present a logical syntax for representation of exogenous constraints. We define a guarded command form for logic formula, and use this structure to generate executable and verifiable code.

Terminology. The basic constructs in Reo consist of *components*, *ports*, *channels* and *nodes*. A *port* is a unidirectional means of synchronous exchange of data between two parties. Two operations can be performed on a port: put and get. The unidirectionality of a port means that each of the two parties on its two sides can use the port exclusively either as input or as output. A party that uses a port as output, can perform only put operations on that port, and a party that uses a port as input can perform only get operations on that port. A port *fires* whenever it has a get and a put operation pending on its respective sides. Firing of a port transfers the data item from the put operation to the get operation. A component is defined with a set of ports at its boundary, called its *interface*. Each port of an interface is either an input or an output port. When a port is shared by two components, we say that those two components are in *composition*. The port must be used as input in one component, and as output in the other component. A component can internally contain several components in composition, in which case we say that the component is *composite*. A component is called *atomic* if its behavior is defined in a specific formal semantic. A binary component, meaning that one whose interface consists of only two ports, is called a channel.

As pictured in Fig. 1, we represent graphically some standard Reo component. More details can be found in [3]. We define in the next subsection a syntax to write the constraint relations among the ports of a component. We later define a semantic of those constraints as the set of data streams accepted by the constraint on their respective ports of the component.

2.1 Exogenous constraints as formula

Logical constraints. We can express the externally observable behavior of a component as a relation over the sequences of data items that it exchanges with its environment through its ports. In this view, an interaction protocols is a relation that constrains the exchanges of data among the ports of various components. We can use logic formulas to express these constraints. We usually separate two types of constraints. A *synchronization constraint* relates firing of ports, and a *data constraint* relates the values exchanged among ports. We follow the work of [10], where they used a special data item NO-FLOW (that we represent here as $*$) to encode synchronization as data constraint.

We use D to denote the set of data items that flow through ports and are stored in memories. We use a special symbol $* \notin D$, that represent no-data, to encode synchronization. We use D_* to denote the set $D \cup \{*\}$. We use P to denote the set of variables that represent values exchanged through their homonym ports, M the set of variables that represent the current values stored in their homonym memory cells, and M' the set of variables that represent the next values to be stored in their unprimed homonym memory cells. We denote the set of variables as V, which consists of the union of P, M and M'. Variables take their values from the domain D_*. Q and F respectively denote the set of n-ary predicate symbols and n-ary function symbol.

A *term* is either a variable $v \in V$ (which can be a port or a memory variable), an n-ary function $f \in F$, or a constant $d \in C$ (where C is the set of constant symbols).

$$t_1, t_2 \quad ::= \quad d \quad | \quad f(t_1, \dots, t_n) \quad | \quad v$$

The set of term expression is denoted by $\mathcal{E}$

A *formula* is built inductively by:

$$\phi \quad ::= t_1 = t_2 \quad | \quad B(t_1, \dots, t_n) \quad | \quad \phi_1 \wedge \phi_2 \quad | \quad \neg\phi \quad | \quad \exists p \phi$$

Where $B \in Q$ is a predicate symbol. The set of formula expressions is denoted by $\mathcal{F}$. We use the shorthand notation $t_1 \neq t_2$ for $\neg(t_1 = t_2)$, $\bot$ for $t_1 \neq t_1$, and we get $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$ as well as $\forall t \phi = \neg(\exists t \neg\phi)$ and $\phi \implies \psi = \neg\phi \vee \psi$. Quantifiers range over port variables only. We call *atomic* a formula that is either an equality, an inequality, or a predicate. We call V_ϕ the set of free variables occurring in ϕ.

Solution of constraints. We use $\gamma : F \to D_*^n \to D_*$, the interpretation function for function symbols, and $\Gamma : V \to D_*$ to denote the interpretation function for variable symbols. We identify a constant c with its homonym value in D_*. We define the interpretation of terms $[\![\cdot]\!]_{(\Gamma,\gamma)} : C \cup F \cup V \to D_*$, over the functions Γ and γ, inductively as:

$$[\![d]\!]_{(\Gamma,\gamma)} = d \in D_*$$
$$[\![f(t_1, \dots, t_n)]\!]_{(\Gamma,\gamma)} = \gamma(f)([\![t_1]\!]_{(\Gamma,\gamma)}, \dots, [\![t_n]\!]_{(\Gamma,\gamma)}) \in D_*$$
$$[\![v]\!]_{(\Gamma,\gamma)} = \Gamma(v) \in D_*$$

Each n-ary predicate symbol $B \in Q$ is assumed to have an interpretation denoted by $\mathcal{I}(B)$, such that $\mathcal{I}(B) \subseteq D_*^n$ Given an interpretation function γ for the function symbols, a *solution* to a formula ϕ is an assignment Γ such that Γ *satisfies* ϕ, written as $\Gamma \models \phi$, defined inductively on ϕ as:

$\Gamma \models \top$ always
$\Gamma \models t_1 = t_2$ iff $[\![t_1]\!]_{(\Gamma,\gamma)} = [\![t_2]\!]_{(\Gamma,\gamma)}$
$\Gamma \models \phi_1 \wedge \phi_2$ iff $\Gamma \models \phi_1$ and $\Gamma \models \phi_2$
$\Gamma \models \neg\phi$ iff $\Gamma \not\models \phi$
$\Gamma \models \exists p \phi$ iff there exists $d \in D_*$ such that $\Gamma \models \phi[d/p]$
$\Gamma \models B(t_1, \dots, t_n)$ iff $([\![t_1]\!]_{(\Gamma,\gamma)}, \dots, [\![t_n]\!]_{(\Gamma,\gamma)}) \in \mathcal{I}(B)$

We extend the domain definition of an n-ary function from D^n to D_*^n by defining $f(t_1, \dots, t_n) = * \iff t_1 = * \vee \dots \vee t_n = *$.

2.2 Permanent and ephemeral constraints

Components and permanent constraints. A Reo component, as introduced earlier, constrains the flow of data through its boundary ports. We call the logical formula used to represent this relation on the data flow through the ports of a component the *permanent constraint* of the component. Fig.1 shows several examples of permanent constraints of component.

Given a port $p \in P$, the proposition of whether or not p fires is encoded as an equality between p and elements of D_*. We say that p fires if $p \neq *$. On the other hand, p does not fire if $p = *$. Taking $a, b \in P$, we can now express that a and b fire synchronously, as the formula $a = b$.

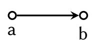
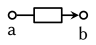
sync(a, b)		$a = b$
fifo(a, b)		$(m' = a \wedge a \neq * \wedge b = * \wedge m = b) \vee$ $(m' = a \wedge a = * \wedge b \neq * \wedge m = b) \vee$ $(m' = m \wedge a = * \wedge b = *)$

Figure 1: From left to right: textual syntax, graphical syntax and permanent constraints. From top to bottom: sync channel, and fifo channel.

The permanent constraint for a *fifo* channel has three clauses. The first one corresponds to filling the buffer with the data item observed at port a; the second one empties the buffer through port b; and the last one corresponds to the case where no port fires, in which case the value in the buffer must remain unchanged.

As hinted previously, protocols can be built by composing primitive. In the case of a composite component, the resulting permanent constraint is defined as the conjunction of the permanent constraints of the underlying components.

The logic is agnostic regarding the direction of data flow and merely represents the constraint on the data observed at each port. We introduce in the next paragraph the designation of input and output for variables, and the notion of ephemeral constraint.

I/O designation and ephemeral constraints. Given a formula, ϕ, and its set of variable $V_\phi = P \cup M \cup M'$, we write the function $io : V_\phi \to \{0, 1\}$ as a designation for input and output variables such that:

$$io : v \mapsto \begin{cases} 1 & v \in M \\ 0 & v \in M' \\ io(v) \in \{0, 1\} & v \in P \end{cases}$$

We define V_ϕ^{in} as the set of variables v such that $io(v) = 1$ and V_ϕ^{out} the set of variables such that $io(v) = 0$. We also extend the notation to P^{in} and P^{out}, referring to the set of input and output port variables as designated by the *io* function.

Since a port is a shared entity between two components, we introduce the relative notion of environment for a component as the constraint imposed by the external world on its boundary ports.

A component knows about the constraint imposed by the environment only by sensing the state of its boundary ports, and cannot access the permanent constraint that describes the environment. We model an *environment* Δ by:

$$\Delta : P_\Delta \to D_*,\ v \ \mapsto \begin{cases} * & v \in P_\phi^{out} \cap P_\Delta \\ \Delta(v) \in D_* & v \in P_\phi^{in} \end{cases}$$

where $P_\phi^{in} \cap P_\Delta = P_\phi^{in}$, and $P_\phi^{out} \cap P_\Delta \subseteq P_\phi^{out}$. We introduce the notion of *ephemeral* constraints ϵ and μ as opposed to a *permanent* constraint ϕ. In contrast to a permanent constraint, an ephemeral constraint may change in time. An environment Δ imposes an external ephemeral constraint $\mu = \bigwedge_{v \in P_\Delta}(v = \Delta(v) \vee v = *)$ on the free variables of the permanent constraint ϕ of a component. The external ephemeral constraint μ represents the fact that, if the environment assigns $*$ to a variable, then the component does not have a choice but to assign $*$ to the same variable; however, if the environment has a data item pending at port v (described by the constraint $\Delta(v) \neq *$), the component can still chose to assign $*$ to the variable and let the environment wait. An assignment Γ such that $\Gamma \models \phi$ imposes an internal ephemeral constraint on the next state of the component with $\epsilon = \bigwedge_{m \in M}(m = \Gamma(m'))$. In other words, the next assignment Γ' must then satisfy the constraint $\phi \wedge \epsilon$, where ϵ represents the internal state of the component, i.e., its memory values.

Intuitively, we can understand external ephemeral constraints as a current state of the ports on the boundary of a component (whether or not the environment has some I/O requests pending on a subset of ports), and the internal ephemeral constraint reflects the current state of the memory in a component. In composition with the permanent constraint, we get the current constraint of the component in the context of its internal and external states.

2.3 Behavior and language of protocols

Operational semantic. Operationally, we consider formulas as labels for *states*, and assignments Γ as labels for *transitions* between states. Each state is labeled by an ephemeral constraint $\epsilon \wedge \mu$. We assume an initial state $\epsilon_0 \wedge \mu_0$, given $\epsilon_0 = \bigwedge_{m \in M} m = c_m$ where $c_m \in D_*$, and $\mu_0 = \bigwedge_{p \in P} p = *$. In the initial state, the constraint μ_0 ensures that all port variables have the value $*$, and the constraint ϵ_0 gives an initial value for all memory variables. We say that two states $\epsilon_1 \wedge \mu_1$ and $\epsilon_2 \wedge \mu_2$ under a permanent constraint ϕ are related by an assignment Γ if and only if $\Gamma \models \phi \wedge \epsilon_1 \wedge \mu_1$ and $\epsilon_2 = \bigwedge_{m \in M}(m = \Gamma(m'))$. In this case, we write $\epsilon_1 \wedge \mu_1 \xrightarrow{\Gamma_\phi} \epsilon_2 \wedge \mu_2$, and drop the subscript ϕ when it is clear from the context.

Note that there exists an assignment Γ that relates each state with itself. Indeed, choosing the assignment Γ such that $\Gamma(m) = \Gamma(m')$ for all $m \in M$ and $\Gamma(p) = *$ for all $p \in P$ is a solution for any ephemeral constraint. Multiple distinct assignments may relate the same two states, in which case the labeled transition system contains multiple labeled edges between the same pair of states.

We refer to an infinite path in the labeled transition system $\epsilon_1 \wedge \mu_1 \xrightarrow{\Gamma_1} \epsilon_2 \wedge \mu_2 \xrightarrow{\Gamma_2} \epsilon_3 \wedge \mu_3 \xrightarrow{\Gamma_3} \dots$ by the infinite sequence of its labels $\Gamma_1, \Gamma_2, \dots$. By $\mathcal{T}$, we designate the set of infinite label sequences of the LTS. Given $\tau \in \mathcal{T}$, we write $\tau(i)$ for the i-th assignment in the sequence τ, and $\tau(i)(v)$ for the value of the variable $v \in V_\phi$ in the i-th assignment of the sequence τ.

Language of protocols. Analogously, an infinite sequence $\Gamma_0, \Gamma_1, \dots \in \mathcal{T}$ can also be transformed into a tuple of data streams. For each variable $v \in V$, we define the data stream $\sigma_v : \mathbb{N} \to D_*$ such that $\sigma_v(i) = \Gamma_i(v)$ for all $i \in \mathbb{N}$. We call σ_V the tuple $(\sigma_{v_1}, \sigma_{v_2}, \dots, \sigma_{v_N})$ where $\{v_1, \dots, v_N\} = V$

We define the language of a formula ϕ with initial memory values defined by ϵ_0 as the set of tuples of data streams for the variables $v \in V$, i.e.:

$$L_\phi = \bigcup_{\tau \in \mathcal{T}} \{(\sigma_{v_1}, \dots, \sigma_{v_N}) \mid \sigma_{v_i}(t) = \tau(t)(v_i)\ for\ all\ v_i \in V\ and\ t \in \mathbb{N}\}$$

a	$*$	1	1	$*$	...
b	$*$	1	1	$*$	...

a	$*$	1	$*$	1	...
b	$*$	$*$	1	$*$	...

Figure 2: Examples of data streams on a ports a and b of a synchronous (left) and asynchronous (right) channel.

Each component defines a constraint on the exchanges of data items through its own boundary ports and two resulting time data streams are shown in Fig. 2. This constraint is represented by a formula such that the interpretation of the formula describes the intended behavior of the component. We represent the behavior of a component as a set of data streams, as also described in [11]. The symbol $*$ used in the table represents the case where no data flow is observed at its respective port.

This overview suffices to understand the Reo compiler, comprehend the behavior of independent components, and that of composite components. In the next subsection, we explain the logical form which the compiler uses to optimize the composition of components. We define the notion of the guarded command form for a formula, and relate it to code generation. The formal development in the next subsection is supported by a tool [2] developed at CWI. Currently, the tool consists of a compilation suite from Treo, a textual language for Reo [12], to executable code (Java) or verifiable code (Promela). We explain the formal steps in the compilation process in the next subsection.

2.4 Compilation

The language of constraints expressed previously is not yet sufficient to define compilation into a program. We need to restrict the expressiveness of the constraints such that we can compile the resulting protocol into a program. We provide in this section a restriction of the language of constraints and define a generic constraint solver for it.

Guarded commands. Given a formula ϕ representing a constraint, and an *io* designation, we define sufficient conditions for ϕ to be written in the guarded command form. We first introduce a fragment of the logic of constraints, give some requirement on ϕ to be expressible as a guarded command, and show some properties if ϕ satisfied those requirements.

We denote by v^{in} and v^{out} a variable v such that $io(v) = 1$ and $io(v) = 0$, respectively. We denote a term by t^{in} and t^{out} such that:

$$t^{in} ::= \quad d \quad | \quad f(t_1^{in}, \dots, t_n^{in}) \quad | \quad v^{in} \qquad t^{out} ::= v^{out}$$

The language of guards g is defined by the following grammar:

$$\begin{aligned} l &::= B(t_1^{in}, \dots, t_n^{in}) \mid t_1^{in} = t_2^{in} \mid t^{out} = * \mid \neg l \\ g &::= l_1 \wedge l_2 \end{aligned}$$

The language of commands c is defined by the following grammar:

$$c ::= t^{out} = t^{in} \mid c_1 \wedge c_2$$

We say that a formula ϕ is in *guarded command form* if

$$\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$$

where g_i are formulas in the language of guards and c_i are formulas in the language of commands. Given ϕ in guarded command form, we call an implication of the type $g \implies c$ in ϕ, a guarded command of ϕ. We call the number of guarded commands in such a formula, the size of that formula. We denote the set of guards by $\mathcal{G}$, and the set of commands by $\mathcal{C}$.

We say that a quantifier free formula $\phi = \phi_1 \vee ... \vee \phi_n$ in disjunctive normal form and expressed in the language of constraints is *deterministic* if ϕ can be written with the grammar of guarded commands such that $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$, where $\wedge$ has precedence over $\vee$, g_i are guards, c_i are commands, and $g_i \wedge g_j \equiv \bot$ for all i, j where $i \neq j$.

We assume that if a term t^{out} is involved in an equality, then the other term is either $*$, or an input term t^{in}. In other words, if an equality $t_1^{out} = t_2^{out}$ appears in the constraint ϕ, there must exist an input term t^{in} such that $t_1^{out} = t^{in}$ and $t_2^{out} = t^{in}$. Moreover,

Proposition 1. *A deterministic formula $\phi = g_1 \wedge c_1 \vee ... \vee g_n \wedge c_n$ can be written as a guarded command where:*

$$\phi = (g_1 \implies c_1) \wedge ... \wedge (g_n \implies c_n) \wedge (\bigvee_i g_i)$$

.

Given two guarded commands $\phi = \bigwedge_i (g_i \implies c_i) \wedge (\bigvee_j g_j)$ and $\psi = \bigwedge_i (g_i' \implies c_i') \wedge (\bigvee_j g_j')$, we write the composition $\phi \wedge \psi$ as the formula:

$$\phi \wedge \psi = \bigwedge_i (g_i \implies c_i) \bigwedge_i (g_i' \implies c_i') \wedge (\bigvee_{i,j} g_i' \wedge g_j)$$

Proposition 2. *Given two deterministic formulas ϕ and ψ, their product $\phi \wedge \psi$ is also deterministic.*

As the construction in the proof of Proposition 2 shows, writing the composition of two deterministic formulas as a deterministic formula may increase the size of the resulting formula. Some optimizations that can be applied to formulas before forming their product, but we don't detail them here for lack of

Example. We now consider an example of guarded commands for the fifo primitive. The *fifo* primitive has a permanent constraint written in Fig. 1, with the *io* designation of $io(a) = 1$ and $io(b) = 0$, i.e., a is an input port and b is an output port. The formula of a *fifo* is deterministic, and with the result of Proposition 1, we can write its permanent constraint as a guarded command:

$$\begin{aligned}\phi_{fifo} = &((a \neq * \wedge b = * \wedge m = *) \implies (m' = a \wedge m = b)) \wedge \\ &((a = * \wedge b \neq * \wedge m \neq *) \implies (m' = a \wedge m = b)) \wedge \\ &((a = * \wedge b = *) \implies m' = m) \wedge \\ &((a \neq * \wedge b = * \wedge m = *) \vee (a = * \wedge b \neq * \wedge m \neq *) \vee \\ &\qquad (a = * \wedge b = *))\end{aligned}$$

The formula for a *fifo* channel has two different guards. The first guard checks whether its source port a is active, in which case the command fills the buffer with the data observed at port a; the second guard checks whether the channel's sink port b is active, in which case its command empties the buffer through port b.

Proposition 3. *Given a deterministic formula ϕ in guarded command form, and an ephemeral constraint $\epsilon \wedge \mu$, Γ is a solution of $\phi \wedge \epsilon \wedge \mu$ if and only if there exists a unique guarded command $g \implies c$ of ϕ such that:*

$$\Gamma \models g \wedge \epsilon \wedge \mu \quad \text{and} \quad \Gamma \models c$$

We now look into the implementation and verification of a protocol described by a formula in guarded command form. As Proposition 3 shows, given a state of an environment and an internal state of our component, each solution is described by a unique guarded command. Therefore, given an environment and an internal state, a program implementing a protocol checks the set of guards that are satisfied by the state of the environment and the internal state, and nondeterministically satisfies one and only one corresponding command. We develop in the next section the steps leading from a protocol specification to a program written in Promela.

3 FROM REO TO PROMELA

In the previous section, we detailed the requirement and the procedure to write a protocol as a formula in the guarded command form. The implication of having such a form for a protocol makes it possible and easier to translate it into a program. In this section, we define a translation from a formula written as a guarded command to a Promela program. We show the correctness of the translation, by comparing the semantic of a Reo specification, and that of its target program in Promela.

Throughout this section, we assume a generic data type denoted as Data for data flowing in the protocol, since data type is specific in the application that employs the protocol.

3.1 Ports and environment in Promela

Ports. In Reo, as defined in the previous section, a port is a location where two components synchronize and exchange data. In Promela, we implement a Reo port as a pair of two Promela channels, each with a buffer size of one. We show that our Promela implementation of a port simulates Reo's synchronized message passing between components.

```
typedef port {
        chan data = [1] of {Data};
        chan synch = [1] of {int}; }
```

Listing 1: definition of a Reo port in Promela

As expressed in Listing 1, a port has a data channel and a synchronization channel. The data channel is responsible of the data flow between input and output ends of a port. The Promela synch channel ensures synchronous exchange between these two ends.

As described in Listing 2, two actions can be performed on such a constructed port: put and take.

```
    inline put(q,a) {             inline take(q,a) {
      int x;                        atomic{q.synch!-1; q.data?a} }
      atomic{q.data!a;
      q.synch?x} }
```

Listing 2: put **and** take **functions**

The action put has two arguments: a port q and a datum a. The function call put(q, a) atomically fills the data channel of q with the datum a, and blocks on the synch channel, waiting to synchronize with the component on the output side of q. The integer x is used to empty the synch channel, but its value does not matter.

The action take has a port q and a variable a as arguments. The function call take(q, a) atomically notifies, by filling the synch channel, that there is a component willing to take data, and blocks on the data channel, until a datum can be taken into the variable a. The integer value of -1 written into the synchronization channel is arbitrary, as synch is used only for signaling.

Environment. We call an external component that interacts with the main protocol, an *agent*. For each port q in the protocol's interface, we assume an *agent* connected to that port. More precisely, if q is used as an input port by the protocol, the *agent* connected to q must use q as an output port, and vice versa. We give in Listing 3 an example of a definition of an agent with two ports as its arguments.

```
proctype agent(port p1; port p2){
        /*  p1: input, p2: output */
        do
          :: /* action */
        od }
```

Listing 3: A generic structure for an agent

Each agent is defined as a proctype in Promela, and runs concurrently with the main protocol. We represent the generic behavior of an agent as an infinite sequence of non deterministic actions (with the do – od loop), but the definition of the precise behavior of an agent is left for the user. Since agents and protocol share ports, it is possible that an agent blocks until a datum is delivered at its port.

We assume a set of agents given by the user. Our compiler generates only the skeleton of an agent including the set of ports in its interface, with the direction of each port (either input or output). As an extension, we intend to make the input/output restriction of ports direction more strict, using the Promela assertion xr and xs to prevent misuse of port directionality. We later specify some properties of the desired observable behavior.

3.2 Translation of a Reo specification

We define formally the translation of a formula written in guarded command form into a Promela specification.

Logical terms. Given a set of term expressions $\mathcal{E}$, we define a mapping of $\mathcal{E}$ to a set of Promela specification $\mathcal{P}$. We assume a function $\mathcal{I}$ that maps every constant, function, and predicate symbol to its interpretation in Promela. We call $[\![\cdot]\!]_\mathcal{E} : \mathcal{E} \to \mathcal{P}$ the function that maps a term to a unique name expression in Promela, such that:

$$[\![f(t_1,\dots,t_n)]\!]_\mathcal{E} = \mathcal{I}(f)([\![t_1]\!]_\mathcal{E},\dots,[\![t_n]\!]_\mathcal{E})$$
$$[\![c]\!]_\mathcal{E} = \mathcal{I}(c) \qquad [\![m]\!]_\mathcal{E} = m$$
$$[\![m']\!]_\mathcal{E} = m \qquad [\![p]\!]_\mathcal{E} = p$$

Memory variables m and m' are mapped to the same name in Promela, since they refer to the same memory location, subject to different actions.

Given a formula ϕ, its set of port variables P, and memory variables M, the function $[\![\cdot]\!]_\mathcal{I} : P \cup M \to \mathcal{P}$ maps each port variable $p \in P$ and memory variable $m \in M$ to a definition in Promela, such that:

$$[\![p]\!]_\mathcal{I} = \textbf{port } [\![p]\!]_\mathcal{E}; \quad \text{and} \quad [\![m]\!]_\mathcal{I} = \textbf{chan } [\![m]\!]_\mathcal{E} = \textbf{[1] of \{Data\};}$$

Each port variable is mapped to an instance of the port structure in Promela. Every port name is unique, and given by $[\![p]\!]_\mathcal{E}$. Each memory variable is defined as a channel of data type **Data** and of size 1. Every memory name is also unique, and given by $[\![m]\!]_\mathcal{E}$.

Besides the mapping of terms to Promela, we introduce Promela variables to access formula variables' value. We use $[\![\cdot]\!]_\mathcal{V} : \mathcal{E} \to \mathcal{P}$ as the function that takes a variable expression and returns a Promela expression such that:

$$[\![m]\!]_\mathcal{V} = \_[\![m]\!]_\mathcal{E} \qquad [\![p]\!]_\mathcal{V} = \_[\![p]\!]_\mathcal{E} \qquad [\![m']\!]_\mathcal{V} = \_[\![m]\!]_\mathcal{E}$$

We use $[\![m]\!]_\mathcal{V}$ and $[\![p]\!]_\mathcal{V}$ to refer to the current value in Promela of the memory variable m or of the port variable p.

Guarded commands. Because we have assumed the source formula to be deterministic, we define the target Promela program by induction on the syntax of the formula in guarded command form.

For a set of guards $\mathcal{G}$, we define $[\![\cdot]\!]_\mathcal{G} : \mathcal{G} \to \mathcal{P}$ such that:

$$[\![g_1 \wedge g_2]\!]_\mathcal{G} = [\![g_1]\!]_\mathcal{G}\texttt{\&\&}[\![g_2]\!]_\mathcal{G}$$
$$[\![B(t_1^{in},\dots,t_n^{in})]\!]_\mathcal{G} = \mathcal{I}(B)([\![t_1^{in}]\!]_\mathcal{V},\dots,[\![t_n^{in}]\!]_\mathcal{V})$$
$$[\![t_1^{in} = t_2^{in}]\!]_\mathcal{G} = \begin{cases} [\![t_1^{in}]\!]_\mathcal{V}\texttt{==}[\![t_2^{in}]\!]_\mathcal{V} & \text{if } t_1^{in} \neq * \wedge t_2^{in} \neq * \\ \texttt{empty(}[\![t_1^{in}]\!]_\mathcal{E}\texttt{)} & \text{if } t_1^{in} \neq * \wedge t_2^{in} = * \text{ and } t_1^{in} \in M \\ \texttt{true} & \text{otherwise} \end{cases}$$
$$[\![\neg(t_1^{in} = t_2^{in})]\!]_\mathcal{G} = \begin{cases} \texttt{full(}[\![t_1^{in}]\!]_\mathcal{E}\texttt{.data)} & \text{if } t_1^{in} \neq * \wedge t_2^{in} = * \\ & \quad \text{and } t_1^{in} \in P \\ \texttt{!(}[\![t_1^{in} = t_2^{in}]\!]_\mathcal{G}\texttt{)} & \text{otherwise} \end{cases}$$
$$[\![\neg(t^{out} = *)]\!]_\mathcal{G} = \texttt{full(}[\![t^{out}]\!]_\mathcal{E}\texttt{.synch)}$$

Note that guards of the shape $p = *$, where $p \in P^{in}$, are trivially mapped to **`true`** in Promela, since such condition can always be satisfied (a port could always have a silent behavior). However, guards of the shape $\neg(p = *)$ must distinguish the case where $p \in P^{in}$, since the port p must have a pending input from the environment.

Before defining the translation from commands to Promela program, we must ensure that the conjunction is properly ordered. We consider the case where a memory variable $m \in M$ is used as input and $m' \in M'$ as output in the same command. In this case, we first give precedence to the command involving m over the command involving m', since m refers to the current value of m, and m' refers to the new value of m.

For a set of commands C, we define the function $[\![\cdot]\!]_C : C \to \mathcal{P}$ such that:

$$[\![c_1 \wedge c_2]\!]_C = [\![c_1]\!]_C\texttt{;}\ [\![c_2]\!]_C\texttt{;}$$
$$[\![t^{out} = t^{in}]\!]_C = \begin{cases} \texttt{put(}[\![t^{out}]\!]_\mathcal{E}\texttt{,}[\![t^{in}]\!]_\mathcal{V}\texttt{)} & \text{if } t^{out} \in P \\ [\![t^{out}]\!]_\mathcal{E}\texttt{!}[\![t^{in}]\!]_\mathcal{V} & \text{if } t^{out} \in M \end{cases}$$

The Promela program for a command is directly followed by an update of the state of the protocol. Since command execution consumes certain values from some port locations, we notify each input port involved in the command to release its value. Therefore, given a command c, for each t^{in} occurring in c, we append at the end of the program $[\![c]\!]_C$ the following statements: $\texttt{take(}[\![t^{in}]\!]_\mathcal{E}\texttt{,}[\![t^{in}]\!]_\mathcal{V}\texttt{)}$ if $t^{in} \in P$ or $[\![t^{in}]\!]_\mathcal{E}\texttt{?}[\![t^{in}]\!]_\mathcal{V}$ if $t^{in} \in M$.

Given a deterministic formula ϕ in the guarded command form, we inductively define the function $[\![\cdot]\!]_\mathcal{F} : \mathcal{F} \to \mathcal{P}$ such that:

$$[\![g \implies c]\!]_\mathcal{F} = [\![g]\!]_\mathcal{G}\texttt{-> atomic\{}[\![c]\!]_C\texttt{\}}$$

The generic structure of a Promela program obtained from a deterministic formula with n guarded commands is shown in Listing 4.

```
proctype Protocol(port p1;...){        /* Guarded commands */
  /* p1: input, ... */                 do
  /* Memory declaration */             :: (guard_1) ->
  chan m = [1] of {int};                  atomic{command_1}
  /* Initial state */                  :: ...
  m!0;                                 :: (guard_n) ->
  /* Local variables */                   atomic{command_n}
  int _m; int _p1 ;                    od }
  ...
```

Listing 4: a generic structure of a protocol

Example. We show as example the protocol of a *fifo* channel. The corresponding deterministic formula for a *fifo* is presented in the example in Section 2.4. Listing 5 shows the generated code obtained from compiling the deterministic formula representing a fifo.

```
proctype Fifo(port a; port b){
    /* a: input, b: output */
    chan m = [1] of {int};
    int _m; int _a ; int _b ;
    do
      :: (empty(m) && full(a.data)) ->
                       atomic{take(a,_a);m!_a;}
      :: (full(m) && full(b.synch)) ->
                       atomic{m?_m; put(b,_m);}
    od }
```

Listing 5: Promela code generated for a Reo fifo channel with a String buffer

Note that the last guarded command is removed from the generated code, since it would result in the trivial statement: **`true -> atomic{m?_m;m!_m}`**, which essentially, merely copies the value of the memory **`m`** to itself.

Initialization. The init process in Promela defines the only active process in the initial state. Ports are initialized and shared between processes that run concurrently, i.e., we apply the instantiation function $[\![\cdot]\!]_\mathcal{I}$ for all $p \in P$. We later reason on the set of processes in the init process only. All agents and protocol are run concurrently.

Correctness. We use the semantic of Promela defined in [21] to show the correctness of our translation. The operational semantics of a Promela program $\mathcal{P}$ composed of processes P_i is defined as a graph $T = (Q, \rightarrow, q_0)$ where Q is a set of states and $\rightarrow$ is a binary relation on states. A state $q \in Q$ is a tuple $q = (l_0, ..., l_m, lv1, ..., lvm, gv)$ where each l_i is a location in process P_i, lvi is the vector of local variable values in process P_i, and gv is the vector of global variables in P. The state $s_0 \in Q$ refers to the initial state.

Proposition 4. *For every gv vector of global variable values, there exists an environment Δ such that $\Delta \equiv gv$, and vice versa.*

We use $[\![g \implies c]\!]_{\mathcal{F}}$ to designate the translation of the guarded command $g \implies c$ to a Promela program. We denote the set of guarded commands as S_{gc}.

Theorem 5. *Given a permanent constraint ϕ, where $[\![\phi]\!]_{\mathcal{F}}$ is its translation, a state $q = (l, lv, gv)$ and an environment Δ such that $gv \equiv \Delta$, for every guarded command $gc \in S_{gc}$ and a state q' such that $q \xrightarrow{gc} q'$, there exists an assignment such that $\Gamma \models g \wedge c$, where $[\![g \implies c]\!]_{\mathcal{F}} = gc$.*

Completeness can be derived by showing that every solution of the guarded command corresponds to an implementation where the set of agents simulate the environment of the solution. By taking an implementation that unifies the agents, and using the non-deterministic properties of Promela, we can simulate any solution of the formula as a statement and a gv vector in Promela. Elaboration of this part remains as future work.

Adding our Promela code generator as a back-end for the Treo compiler significantly extends the application range of the current compiler. From the same Reo specification, the compiler can generate either an executable code in Java, or a verifiable code in Promela.

4 CASE STUDY

In this section, we consider specification of a railway protocol as a Reo circuit, its compilation into Promela, and verification of some of its properties using SPIN.

4.1 Railway case study

We consider the simplified trains protection in CBTC (Communications Based Train Control) systems. The system is illustrated in the Figure 3 and detailed in depth in previous work of one of the author [19]. To illustrate our approach, we propose a slight adaptation of the original version of the system.

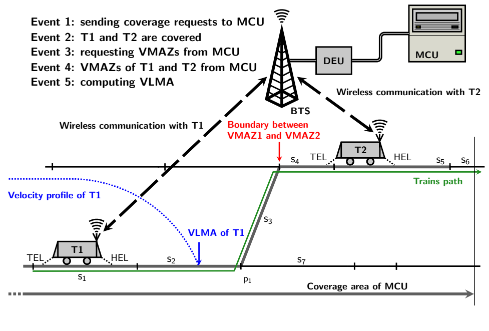

Event 1: sending coverage requests to MCU
Event 2: T1 and T2 are covered
Event 3: requesting VMAZs from MCU
Event 4: VMAZs of T1 and T2 from MCU
Event 5: computing VLMA
DEU
MCU
BTS
Wireless communication with T2
Wireless communication with T1
Boundary between VMAZ1 and VMAZ2
T2
TEL
HEL
Velocity profile of T1
Trains path
VLMA of T1
T1
TEL
HEL
Coverage area of MCU

Figure 3: Simplified trains protection functions.

The studied system consists of a set of components located inside and outside trains, that interact continuously to calculate and exchange parameters necessary to guarantee safe circulation of trains.

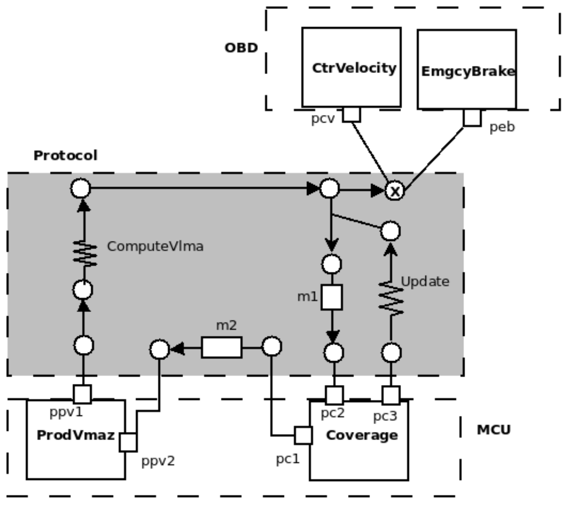


Figure 4: Reo circuit

In this paper, we focus on modeling and verifying the interaction protocol of the main OBD component (On-Board Device) that interacts continuously with the *Movement Control Unit* component (MCU). This protocol implies constraints on the subcomponents of OBD (CtrVelocity, EmgcyBrake) and those of MCU (Coverage, ProdVmaz). Before modeling this protocol as a Reo circuit, in the next section, we describe it informally by showing the chaining of component actions that OBD and other components enable during their interactions.

The OBDs of T1 and T2 initiate the protection process by asking if they are visible to the MCU component, by sending the message *covReq* to Coverage subcomponent. There are several MCUs covering the entire line, with overlapping coverage sections, useful for information exchange between them. Locations are sent by wireless communication to the nearest *Base Transmission Station* (BTS) and saved in its *Data Exchange Unit* (DEU), which in turn transfers them to MCU (event 1). The internal subcomponent Coverage of MCU determines whether or not the zone between TEL and HEL (tail and head external locations) is completely included within its coverage area, and responds to T1 and T2 by sending the messages *covered* or *uncovered* (event 2). Next, each train asks from its covering MCUs its *Vital Movement Authority Zone* (VMAZ), by sending the message *vmazReq*. A VMAZ is an area (sequence of segments) in which the train can safely circulate (event 3). MCU ensures that VMAZs of successive trains never overlap to avoid collisions. VMAZs are computed, by the MCU subcomponent ProdVmaz, through chaining of segments according to route information. Chaining terminates at the nearest obstacle on the train trajectory: the end of MCU coverage area, an uncontrolled switch, or the beginning of a segment containing TEL of the next train, etc. Finally, the train computes the *Vital Limit of Movement Authority* (VLMA) by incorporating a fixed safety margin within the limit of its VMAZ (event 5).

Depending on VLMA an OBD component will either enable its subcomponent CtrVelocity, or EmgcyBrake. The subcomponent CtrVelocity gradually reduce the train velocity down to zero (via its internal action $ctrVelocity$) before HEL reaches VLMA. The activation of the subcomponent EmgcyBrake results into an emergency brake action (*emgcyBrake* action).

4.2 Reo specification

The railway example presented above is translated into a Reo circuit, described graphically in Fig. 4. Six different type of components are used to model the protocol. Two *fifo* channels are used to represent the asynchronous communication while Coverage sends an update and expects an answer, and when Coverage communicates with ProdVmaz. Transformer channels ComputeVlma and Update are used to apply functions on the data flowing at their ports. Therefore, the transformer ComputeVlma returns the speed value, responding to the message sent by ProdVmaz, and Update updates the data of the message sent by Coverage. The *xrouter* component (circle with a cross) models the routing replication of data to CtrVelocity or EmgcyBrake. Once a data enters the xrouter component, the data is sent either to CtrVelocity or to EmergcyBrake but not to both. *Merger*, *replicator*, and *sync* components have the usual behavior and are used to model synchronization properties. For more details on Reo components behavior, please refer to [3].

The protocol component (dashed line and grey font) represents the permanent constraint of the system. The four components CtrVelocity, EmgcyBrake, ProdVmaz, and Coverage, correspond to the environment of the protocol. In the next section, we generate the Promela code for this protocol, link the protocol with its agents, and verify some synchronization properties.

4.3 Code generation

Initialization. To generate the Promela code corresponding to Reo circuit described in Fig. 4, we exploit our compiler that accepts as input Reo specification and generates as output Promela code by applying the rules described in Section 4. We obtain the following Promela proctypes described in Listing 6.

```
init {
port pcv; port pc1; port pc2;
...
atomic{
run prodVmaz(ppv2,ppv1);run emergencyBrake(peb);
run ctlVelocity(pcv); run coverture(pc2,pc3,pc1);
run Protocol(pcv,pc3,pc1,ppv1,ppv2,pc2,peb) } }
```

Listing 6: Generated proctypes

Ports are first initialized with a respective unique named. Then, the Protocol proctype implements the Reo specification of the interaction protocol, with its corresponding interface. In addition, the proctypes ProdVmaz, EmgcyBrake, CtrlVelocity, Coverage, and EmgcyBrake implement the boundary agent, considered as an environment for the main protocol proctype. Each proctype accepts as parameters, ports associated with the interface of its respective component.

Protocol. In Listing 7, we show a partial implementation of the Protocol proctype corresponding to Reo specification described in Figure 4. To illustrate the result obtained from the compiler, we describe, for example, the actions *coveReq*, and *covered*, that are implemented as two guarded commands. The first action models the message sent by the OBD component to the agent Coverage. The second one models the message received by the OBD component from the agent Coverage. For example, as shown in Listing 7 in the implementation of *coveReq*, the guard part of the action specifies that the buffer m_1 is full because there is a message to send, and the port p_{c2} (instantiation of the parameter $p6$ in Protocol) associated to the component Coverage is free. The command part of the action specifies that the message will be sent in P_{c2} and the buffer will be available.

```
proctype Protocol(port p1; ... ; port p7){
...
do
::(full(m1) && full(p6.sync)) -> atomic{ m1?_m1; put(p6,_m1);}
        /* covereq!*/
...
:: (full(p3.data) && empty(m2)) -> atomic{take(p3,_p3);m2!_p3;}
        /*covered*/
od }
```

Listing 7: Guarded commands for covereq and covered actions

Simulation and verification. The Promela program generated by our compiler allows to run random or guided simulation with SPIN for analyzing the protocol. It allows also to verify safety properties, such as the absence of deadlock states. The verification performed on the Promela program, corresponding to the case study, shows that the protocol behavior does not reach any deadlock state. Using the approach described in [9], we define, in the next subsection, some new flag variables used to verify liveness properties. The experiment detailed in the use case can be reproduced by downloading the Reo compiler [2].

4.4 Verification of LTL properties

Flags. LTL properties, in our case, describe properties of the run in the generated code. As presented in previous section, the state of a generated Promela program is entirely captured in the global variables value, and a vector of local variables value for each concurrent proctype. We decide, to abstract system's state by defining flags to describe the main state we interested in. As shown in Listing 8

```
/*flags declaration */
bit hassend=0, hasreceive=0, msg_covReq=0,msg_vmazReq=0,
msg_covered=0, msg_uncovered=0, msg_vmaz=0,brake=0,ctrvelo=0;
...
do
::(full(m1) && full(p6.sync)) -> atomic{ m1?_m1; put(p6,_m1);
d_step{hassend=1; msg_covReq=1; hasreceive=0; msg_vmazReq=0;
msg_covered=0; msg_uncovered=0; msg_vmaz=0; brake=0;
ctrvelo=0;}} /* covereq!*/
```

Listing 8: Flags of the generated code

Mainly, each sending and receiving action from the protocol point of view has a dedicated flag. This allows to keep track of the actions performed by the components and the reactions of their environment. These flags are updated together with each send/receive event, using a *d_step* statement to ensure the values assignment in one execution step. For example in Listing 8, we present the Promela code of the guarded command *covereq* enriched with its corresponding flags.

LTL properties. To verify the correctness of the specified protocol, we propose to verify the following LTL properties presented in Listing 9.

```
ltl p1{[]((hassend==1 && msg_covReq==1) -> <> (hasreceive==1
&& (msg_uncovered==1 || msg_covered==1)))} /*satified */

ltl p2{[](msg_covered==1 -> <> (hasreceive==1 && msg_vmaz==1
&& brake==1))} /*not satisfied */
```

Listing 9: Flags of the generated code

The property p1 specifies that, always if the component *OBD* sends the message *covReq* then it will receive a message *covered* or *uncovered*. This property is satisfied. The property p2 specifies that always if the *OBD* receives the message *covered* then it will receive the message *vmaz* and the action *emgcyBrake* will be enabled. This property is not satisfied, because there is the option of enabling the action *ctrVelocity* instead of *emgcyBrake*.

5 RELATED WORK

In this paper, we propose compiling Reo circuits into Promela to formally verify the LTL properties of protocols of interactions among components. Formal verification of Reo specifications using a number of tools has already appeared in the literature (see below). Complementing this existing set of tools, in our work, we opted to use the SPIN model checker, because (i) it is one of the most advanced analysis and verification tools available, and (ii) its Promela language resembles our guarded command formal specification of Reo circuits. We are not aware of a comprehensive work on verification of Reo circuits using SPIN.

We proposed a formal framework for translation of Reo models into Promela. We use a transition-systems-based operational semantics of Promela to express the formal semantics of Reo, through which we establish the correctness of our translation. Most of the existing work uses constraint automata, instead, as operational semantics.

Similar to our concerns, the CHARMY framework [20] also addresses verifying LTL formulas expressing properties of component-based systems. CHARMY is a UML based specification, where state diagrams are counterparts of our agents, and sequence diagrams are that of our protocols. CHARMY defines a system as a set of state diagrams, coordinated by a sequence diagram. The CHARMY framework offers the possibility to translate its specification into a Promela program, and thereby allows the verification of properties using SPIN. However, CHARMY allows composition of state diagram specifications only, and not that of sequence diagrams. In our case, agents and protocol differ only in their implementations. In Reo, protocols are constructed compositionally, and their composition preserves and propagates synchrony. CHARMY does not seem to support such compositions.. Moreover, Reo can express nesting of components, which is disallowed in CHARMY. Finally, properties of protocols are specified using property sequence chart (PSC) in CHARMY. In our approach, we require the specification of LTL properties directly in Promela, and leave as future work the integration of property specification at the design level.

Vereofy [5–7] is a model checking tool developed at the University of Dresden to analyze and verify Reo connectors. Vereofy has two input languages: the Reo Scripting Language (RSL), used to specify the coordination protocol, and a guarded command language called Constraint Automata Reactive Module Language (CARML), a textual version of constraint automata used to specify the behavior of components. Vereofy allows the verification of temporal properties expressed in LTL and CTL-like logics. In contrast to Vereofy, in our work we use only one language, Promela, to specify both component behavior and coordination protocols.

The constraint automata semantics for Reo was also considered in [8] for defining and verifying bisimulation and language equivalence between Reo connectors. In [3, 4], the authors considered time constraints, and proposed a timed version of constraint automaton to verify by model checking timed CTL properties. In [15, 16], the authors use timed constraint automata and present a SAT-based approach for bounded model checking of real-time component connectors. Another verification approach based on SAT solvers was proposed in [11], exploiting Alloy. This work allows analysis of Reo circuits and verification of their properties expressed as predicates in the lightweight modeling language of Alloy, which is based on first-order relational logic. In [17, 18], Kokash et al. proposed a framework for the verification of Reo circuits using the mCRL2 toolset (developed at the TU of Eindhoven). Their tool automatically generates mCRL2 specifications from Reo graphical models. The translation from Reo to mCRL2 uses the constraint automata semantics of Reo.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented sufficient conditions to compile a protocol specified in Reo into a program implemented in Promela. We introduced the notions of ephemeral and permanent constraints; the former consist of constraints that change through time, reflecting the states of the environment or internal memories, whereas the latter consist of structural constraints that must hold invariably. We defined sufficient conditions for expressing the permanent constraint of a component in the guarded command form. We then defined a translation of a formula in guarded command form to a Promela program. We show the correctness of our translation, which we have implemented as a back-end for the current Reo compiler.

We demonstrate the relevance of our work in a case study involving design and verification of a railway component-based system. We use our compiler to translate the Reo specification of a protocol in this system into Promela, and use SPIN to verify LTL-specified properties, such as safety and liveness. In future work, we would like to support specification of LTL properties at design level as an extension of constraint formula. This direction is justified by the argument that in Reo, we can deduce the states of a protocol entirely from the current state of its boundary ports and its internal memory. A designer, thus, has enough information at the level of Reo to specify not just a protocol, but also its required LTL properties, in some suitable constraint language, which our extended compilation can subsequently translate into Promela LTL statement on protocol states, for verification.

REFERENCES

[1] 2018. Link to the proofs. https://cloud.benjaminlion.fr/s/PtH6RtW056tuFJ4

[2] 2018. Reo github webpage. https://github.com/ReoLanguage/Reo

[3] Farhad Arbab. 2004. Reo: A Channel-based Coordination Model for Component Composition. *Mathematical. Structures in Comp. Sci.* 14, 3 (June 2004), 329–366. https://doi.org/10.1017/S0960129504004153

[4] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. 2007. Models and temporal logical specifications for timed component connectors. *Software & Systems Modeling* 6, 1 (01 Mar 2007), 59–82. https://doi.org/10.1007/s10270-006-0009-9

[5] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. 2009. Formal Verification for Components and Connectors. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 82–101.

[6] Christel Baier, Tobias Blechmann, Joachim Klein, and Sascha Klüppelholz. 2009. A Uniform Framework for Modeling and Verifying Components and Connectors. In *Coordination Models and Languages*, John Field and Vasco T. Vasconcelos (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 247–267.

[7] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolfgang Leister. 2010. Design and Verification of Systems with Exogenous Coordination Using Vereofy. In *Leveraging Applications of Formal Methods, Verification, and Validation*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–111.

[8] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. 2006. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming* 61, 2 (2006), 75 – 113. https://doi.org/10.1016/j.scico.2005.10.008 Second International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA'03).

[9] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. 2013. Incremental Modeling of System Architecture Satisfying SysML Functional Requirements. In *Formal Aspects of Component Software - 10th International Symposium, FACS 2013, Nanchang, China, October 27-29, 2013, Revised Selected Papers*. 79–99. https://doi.org/10.1007/978-3-319-07602-7_7

[10] Dave Clarke, Jose Proenca, Alexander Lazovik, and Farhad Arbab. 2011. Channel-based coordination via constraint satisfaction. *Science of Computer Programming* 76, 8 (2011), 681 – 710. https://doi.org/10.1016/j.scico.2010.05.004 Special issue on the 7th International Workshop on the Foundations of Coordination Languages and Software Architectures (FOCLASA'08).

[11] Kasper Dokter and Farhad Arbab. 2018. Rule-Based Form for Stream Constraints. In *Coordination Models and Languages*, Giovanna Di Marzo Serugendo and Michele Loreti (Eds.). Springer International Publishing, Cham, 142–161.

[12] Kasper Dokter and Farhad Arbab. 2018. Treo: Textual Syntax for Reo Connectors. 272 (06 2018), 121–135.

[13] Gerard Holzmann. 2003. *Spin Model Checker, the: Primer and Reference Manual* (first ed.). Addison-Wesley Professional.

[14] Sung-Shik T. Q. Jongmans and Farhad Arbab. 2012. Overview of Thirty Semantic Formalisms for Reo. *Sci. Ann. Comp. Sci.* 22 (2012), 201–251.

[15] Stephanie Kemper. 2010. Compositional Construction of Real-Time Dataflow Networks. In *Coordination Models and Languages*, Dave Clarke and Gul Agha (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 92–106.

[16] S. Kemper. 2012. SAT-based verification for timed component connectors. *Science of Computer Programming* 77, 7 (2012), 779 – 798. https://doi.org/10.1016/j.scico.2011.02.003 (1) FOCLASA'09 (2) FSEN'09.

[17] Natallia Kokash, Christian Krause, and Erik de Vink. 2012. Reo + mCRL2 : A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing* 24, 2 (01 Mar 2012), 187–216. https://doi.org/10.1007/s00165-011-0191-6

[18] Natallia Kokash, Christian Krause, and Erik P. de Vink. 2010. Verification of Context-Dependent Channel-Based Service Models. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael Leuschel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg,

21–40.
[19] Sebti Mouelhi, Khalid Agrou, Samir Chouali, and Hassan Mountassir. 2015. Object-Oriented Component-Based Design using Behavioral Contracts: Application to Railway Systems. In *Proceedings of the 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering, CBSE 2015, Montreal, QC, Canada, May 4-8, 2015*. 49–58. https://doi.org/10.1145/2737166.2737171
[20] P. Pelliccione, P. Inverardi, and H. Muccini. 2009. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *IEEE Transactions on Software Engineering* 35, 3 (May 2009), 325–346. https://doi.org/10.1109/TSE.2008.104
[21] Stavros Tripakis and Costas Courcoubetis. 1996. Extending promela and spin for real time. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–348.