Compiling Protocols to Promela and Verifying their LTL Properties

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

Samir Chouali
Univ. Bourgogne Franche-Comté,
France
schouali@femto-st.fr

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

ABSTRACT
Component-based systems can be modeled as black-box, stand-alone 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 protocol 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 \( \ast \in D \), that represent no-data, to encode synchronization. We use \( D_0 \), to denote the set \( D \cup \{ \ast \} \). 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\( D_1, Q \) 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).

\[ t_1, t_2 \ ::= \ d \mid f(t_1, \ldots, t_n) \mid v \]

The set of term expression is denoted by \( \mathcal{E} \).

A **formula** is built inductively by:

\[ \phi \ ::= t_1 = t_2 \mid B(t_1, \ldots, t_n) \mid \phi_1 \land \phi_2 \mid \neg \phi \mid \exists \phi \]

Where \( B \in Q \) is a predicate symbol. The set of formula expressions is denoted by \( \mathcal{T} \). We use the shorthand notation \( t_1 \neq t_2 \) for \( \neg(t_1 = t_2) \). For \( t_1 \neq t_2 \), and we get \( \phi \lor \psi = \neg(\neg \phi \land \neg \psi) \) as well as \( A \phi = \neg(\exists \phi) \) and \( \phi \implies \psi = \neg \phi \lor \psi \). Quantifiers range over port variables only. We call atomic a formula that is either an equality, an inequality, or a predicate. We call \( V_\phi \) the set of free variables occurring in \( \phi \).

**Solution of constraints.** We use \( \gamma : F \rightarrow D^\phi_0 \rightarrow D_\ast \), the interpretation function for function symbols, and \( \Gamma : V \rightarrow D_\ast \), to denote the interpretation function for variable symbols. We identify a constant \( c \) with its homonym value in \( D_\ast \). We define the interpretation of terms \([ t \] \( ) \( , \gamma \) : \( C \cup F \cup V \rightarrow D_\ast \), over the functions \( \Gamma \) and \( \gamma \), inductively as:

\[ [d]_{\Gamma, \gamma} = \ Gamma \mid f(t_1, \ldots, t_n)]_{\Gamma, \gamma} = \gamma(f([t_1]_{\Gamma, \gamma}, \ldots, [t_n]_{\Gamma, \gamma})) \in D_\ast \]

Each n-ary predicate symbol \( B \in Q \) is assumed to have an interpretation denoted by \( I(B) \), such that \( I(B) \subseteq D^\phi_0 \). Given an interpretation function \( \gamma \) for the function symbols, a solution to a formula \( \phi \) is an assignment \( \Gamma \) such that \( \Gamma \ satisifies \phi \), written as \( \Gamma \models \phi \), defined inductively on \( \phi \) as:

\[ \Gamma \models \forall \phi \iff \Gamma \models \phi \]

We extend the domain definition of an n-ary function from \( D^\ast \) to \( D^\phi_0 \) by defining \( f(t_1, \ldots, t_n) \) \( = * \) \( \iff \) \( t_1 = * \lor \ldots \lor t_n = * \).

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.

![Figure 1: From left to right: textual syntax, graphical syntax and permanent constraints. From top to bottom: sync channel, and fifo channel.](image)

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 \( \phi \), and its set of variable \( V_\phi = P \cup M \cup M' \), we write the function \( io : V_\phi \rightarrow \{0, 1\} \) 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 \( V^{in}_\phi \) as the set of variables \( v \) such that \( io(v) = 1 \) and \( V^{out}_\phi \) 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 $\Delta$ by:

$$\Delta : P_\Delta \rightarrow D_\Delta, \nu \mapsto \{\ast \in P^\ast \cap P_\Delta \mid (\nu \in P^\ast \cap P_\Delta) \land (\nu \in P^\ast_\phi) \}$$

where $p^\ast_\phi \subseteq P_\phi \cap P^\ast \phi$ and $P^\ast \cap P_\Delta \subseteq P^\ast_\phi$. We introduce the notion of ephemeral constraints $\epsilon$ and $\mu$ as opposed to a permanent constraint $\phi$. In contrast to a permanent constraint, an ephemeral constraint may change in time. An environment $\Delta$ imposes an external ephemeral constraint $\mu = \bigwedge_{i \in \mathbb{N}} (m = \Delta(v) \forall \nu = \ast)$ on the free variables of the permanent constraint $\phi$ of a component. The external ephemeral constraint $\mu$ represents the fact that, if the environment assigns $\ast$ to a variable, then the component does not have a choice but to assign $\ast$ to the same variable; however, if the environment has a data item pending at port $v$ (described by the constraint $\Delta(v) \neq \ast$), the component can still choose to assign $\ast$ to the variable and let the environment wait. An assignment $\Gamma$ such that $\Gamma \models \phi$ imposes an internal ephemeral constraint on the next state of the component with $\epsilon = \bigwedge_{m \in \mathbb{M}} (m = \Gamma(m'))$. In other words, the next assignment $\Gamma'$ must then satisfy the constraint $\phi \land \epsilon$, where $\epsilon$ 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 $\Gamma$ as labels for transitions between states. Each state is labeled by an ephemeral constraint $\epsilon \land \mu$. We assume an initial state $\epsilon_0 \land \mu_0$, given $e_0 = \bigwedge_{m \in \mathbb{M}} (m = e_m)$ where $e_m \in D_m$, and $\mu_0 = \bigwedge_{p \in \mathbb{P}} p = \ast$. In the initial state, the constraint $\mu_0$ ensures that all port variables have the value $\ast$, and the constraint $e_0$ gives an initial value for all memory variables. We say that two states $\epsilon_1 \land \mu_1$ and $\epsilon_2 \land \mu_2$ under a permanent constraint $\phi$ are related by an assignment $\Gamma$ if and only if $\Gamma \models \phi \land \epsilon_1 \land \mu_1$ and $\Gamma_0 = \epsilon_0 \land \mu_0$, and drop the subscript $\phi$ when it is clear from the context.

Note that there exists an assignment $\Gamma$ that relates each state with itself. Indeed, choosing the assignment $\Gamma$ such that $\Gamma(m) = \Gamma(m')$ for all $m \in \mathbb{M}$ and $\Gamma(p) = \ast$ for all $p \in \mathbb{P}$ 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 $\epsilon_1 \land \mu_1 \xrightarrow{\Gamma_1} \epsilon_2 \land \mu_2 \xrightarrow{\Gamma_2} \epsilon_3 \land \mu_3 \xrightarrow{\Gamma_3} \ldots$ by the infinite sequence of its labels $\Gamma_1, \Gamma_2, \ldots$. By $\mathcal{T}$, we designate the set of infinite label sequences of the LTS. Given $\tau \in \mathcal{T}$, we write $r(i)$ for the $i$-th assignment in the sequence $\tau$, and $r(i)(v)$ for the value of the variable $v \in \nu$ in the $i$-th assignment of the sequence $\tau$.

**Language of protocols.** Analogously, an infinite sequence $\Gamma_0, \Gamma_1, \Gamma_2, \ldots \in \mathcal{T}$ can also be transformed into a tuple of data streams. For each variable $v \in V$, we define the data stream $\sigma(v) : \mathbb{N} \rightarrow D_v$ such that $\sigma(v)(i) = \Gamma_i(v)$ for all $i \in \mathbb{N}$. We call $\nu_{\epsilon_0}$ the tuple $(\sigma_{v_1}, \sigma_{v_2}, \ldots, \sigma_{v_N})$ where $\{v_1, \ldots, v_N\} = V$.

We define the language of a formula $\phi$ with initial memory values defined by $\epsilon_0$ as the set of tuples of data streams for the variables $v \in V$, i.e.:

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

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

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

$$t^{in} := \delta | f(t^{in}_1, \ldots, t^{in}_n) | \nu^{in} \land \nu^{out} \quad := \nu^{out}$$

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

$$l ::= B(t^{in}_1, \ldots, t^{in}_n) | t^{in}_i | t^{out} | \ast \mid \neg l$$

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

$$c ::= \nu^{in} \land \nu^{out} \land [c_1 \land c_2]$$

We say that a formula $\phi$ is in guarded command form if

$$\phi = \bigwedge_j (g_i \Rightarrow c_j) \land (\bigvee_j g_j)$$

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

---

**Figure 2:** Examples of data streams on a ports a and b of a synchronous (left) and asynchronous (right) channel.
We say that a quantifier free formula $\phi = \phi_1 \lor \ldots \lor \phi_n$ in disjunctive normal form and expressed in the language of constraints is deterministic if $\phi$ can be written with the grammar of guarded commands such that $\phi = g_1 \land c_1 \lor \ldots \lor g_n \land c_n$, where $\land$ has precedence over $\lor$, $g_i$ are guards, $c_i$ are commands, and $g_i \land c_i \equiv \bot$ for all $i$, 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 $t_{out} = t_{in}$ appears in the constraint $\phi$, there must exist an input term $t_{in}$ such that $t_{out} = t_{in}$ and $t_{out} = t_{in}$ Moreover.

Proposition 1. A deterministic formula $\phi = g_1 \land c_1 \lor \ldots \lor g_n \land c_n$ can be written as a guarded command where:

$$\phi = (g_i \implies c_i) \land \ldots \land (g_n \implies c_n) \land \left( \bigvee_i g_i \right)$$

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

$$\phi \land \psi = \bigwedge_i (g_i \implies c_i) \land \left( \bigvee_j g'_j \land g_j \right)$$

Proposition 2. Given two deterministic formulas $\phi$ and $\psi$, their product $\phi \land \psi$ 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 space.

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:

$$\phi_{fifo} = ((a \neq \ast \land b = \ast \land m = \ast) \implies (m' = a \land m = \ast)) \land ((a = \ast \land b \neq \ast \land m = \ast) \implies (m' = a \land m = \ast)) \land ((a = \ast \land b = \ast) \implies m') \land ((a \neq \ast \land b = \ast \land m = \ast) \lor (a = \ast \land b \neq \ast \land m = \ast) \lor (a = \ast \land b = \ast))$$

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 $\phi$ in guarded command form, and an ephemeral constraint $e \land \mu$, $\Gamma$ is a solution of $\phi \land e \land \mu$ if and only if there exists a unique guarded command $g \implies c$ of $\phi$ such that:

$$\Gamma \models g \land e \land \mu \quad \text{and} \quad \Gamma \models c$$

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.
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 \( E \), we define a mapping of \( E \) to a set of Promela specification \( P \). We assume a function \( I \) that maps every constant, function, and predicate symbol to its interpretation in Promela. We call \( [\cdot]_E : E \rightarrow P \) the function that maps a term to a unique name expression in Promela, such that:

\[
[f(t_1, \ldots, t_n)]_E = I(f)[t_1]_E, \ldots, [t_n]_E
\]

We use \( J \) to define formally the translation of a formula written in guarded command form into a Promela specification.

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 \( m \), 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 \( [\cdot]_C : C \rightarrow P \) such that:

\[
[c_1 \land c_2]_C = [c_1]_C; [c_2]_C;
\]

\[
t'_{\text{out}} = t'_{\text{in}}]_C = \begin{cases} \text{put}(\text{in}_{\text{out}})_E, [\text{in}_{\text{in}}]_E \big) & \text{if } t'_{\text{out}} \in P \\ \text{ref}(\text{in}_{\text{out}})_E, [\text{in}_{\text{in}}]_E \big) & \text{if } t'_{\text{out}} \in M \end{cases}
\]

The Promela program for a command is directly followed by an update of the state of the state. 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 \( t'_{\text{in}} \) occurring in \( c \), we append at the end of the program \( [c]_C \) the following statements: take \( ([\text{in}_{\text{out}}]_E, [\text{in}_{\text{in}}]_E) \) if \( t'_{\text{in}} \in P \) or \( ([\text{in}_{\text{out}}]_E, [\text{in}_{\text{in}}]_E) \) if \( t'_{\text{in}} \in M \).

Given a deterministic formula \( \phi \) in the guarded command form, we inductively define the function \([\cdot]_F : F \rightarrow P \) such that:

\[
[g \implies c]_F = [g]_F \implies [c]_C
\]

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

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.

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 \( \implies \text{atomic}(\text{m[7][3]; put(b._m);}) \) which essentially, merely copies the value of the memory \( m \) to itself.

Initialization. The init process in Promela defines the only active environment. We later specify some properties of the desired observable behavior.

3.2 Translation of a Reo specification

For a Reo specification, we have a set of ports \( P \), a set of variables \( V \) and a set of guards \( G \). We define the mapping of \( E \) to a set of Promela specification \( P \) as follows:

\[
\begin{align*}
[f(t_1, \ldots, t_n)]_E & = I(f)[t_1]_E, \ldots, [t_n]_E \\
[c_1 \land c_2]_E & = [c_1]_E; [c_2]_E \\
t'_{\text{out}} = t'_{\text{in}}]_E & = \begin{cases} \text{put}(\text{in}_{\text{out}})_E, [\text{in}_{\text{in}}]_E \big) & \text{if } t'_{\text{out}} \in P \\
\text{ref}(\text{in}_{\text{out}})_E, [\text{in}_{\text{in}}]_E \big) & \text{if } t'_{\text{out}} \in M 
\end{cases}
\end{align*}
\]

The Promela program for a command is directly followed by an update of the state of the state. 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 \( t'_{\text{in}} \) occurring in \( c \), we append at the end of the program \( [c]_C \) the following statements: take \( ([\text{in}_{\text{out}}]_E, [\text{in}_{\text{in}}]_E) \) if \( t'_{\text{in}} \in P \) or \( ([\text{in}_{\text{out}}]_E, [\text{in}_{\text{in}}]_E) \) if \( t'_{\text{in}} \in M \).

Given a deterministic formula \( \phi \) in the guarded command form, we inductively define the function \([\cdot]_F : F \rightarrow P \) such that:

\[
[g \implies c]_F = [g]_F \implies [c]_C
\]

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

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.

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 \( \implies \text{atomic}(\text{m[7][3]; put(b._m);}) \) which essentially, merely copies the value of the memory \( m \) to itself.

Initialization. The init process in Promela defines the only active environment. We later specify some properties of the desired observable behavior.
Correctness. We use the semantic of Promela defined in [21] to show the correctness of our translation. The operational semantics of a Promela program $P$ composed of processes $P_i$ is defined as a graph $\Gamma = (Q, \rightarrow, q_0)$ where $Q$ is a set of states and $\rightarrow$ is a binary relation on states. A state $q \in Q$ is a tuple $q = (l_1, \ldots, l_m, \text{gv}, \text{gv})$ where each $l_i$ is a location in process $P_i$, $\text{gv}$ is the vector of local variable values in process $P_i$, and $\text{gv}$ is the vector of global variables in $P$. The state $q_0 \in Q$ refers to the initial state.

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

We use $[g \models c]_\Delta$ to designate the translation of the guarded command $g \models c$ to a Promela program. We denote the set of guarded commands as $\mathcal{S}_g$.

**Theorem 5.** Given a permanent constraint $\phi$, where $[\phi]_\Delta$ is its translation, a state $q = (l, \text{gv}, \text{gv})$ and an environment $\Delta$ such that $\text{gv} \models \Delta$, for every guarded command $gc \in \mathcal{S}_g$ and a state $q'$ such that $q \models gc \Rightarrow q' \models gc$, there exists an assignment such that $\Gamma \models g \land c$, where $[g \models c]_\Delta \models gc$.

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 $\text{gv}$ 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.

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.

![Diagram of simplified trains protection functions](image)

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 (CtrlVelocity, 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 cov/Req 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 ver/Req. 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 (VMA) 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 CtrlVelocity, or EmgcyBrake. The subcomponent CtrlVelocity gradually reduce the train velocity down to zero (via its internal action ctrl/Velocity) before HEL reaches VLMA. The activation of the subcomponent EmgcyBrake results into an emergency brake action (emgcy/Brake 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 \texttt{fif0} 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 ComputeVima and Update are used to apply functions on the data flowing at their ports. Therefore, the transformer ComputeVima returns the speed value, responding to the message sent by ProdVmaz, and Update updates the data of the message sent by Coverage. The \texttt{xrouter} component (circle with a cross) models the routing replication of data to Ctrvelocity or EmygBrake. Once a data enters the \texttt{xrouter} component, the data is sent either to Ctrvelocity or to EmergencyBrake 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, EmergencyBrake, 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 proctype described in Listing 6.

```
intit {
  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 name. Then, the Protocol proctype implements the Reo specification of the interaction protocol, with its corresponding interface. In addition, the proctypes ProdVmaz, EmygBrake, Ctrvelocity, Coverage, and EmergencyBrake 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 \texttt{coveReq}, and \texttt{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 \texttt{coveReq}, the guard part of the action specifies that the buffer \texttt{m1} is full because there is a message to send, and the port \texttt{p2} (instantiation of the parameter \texttt{p6} in Protocol) associated to the component Coverage is free. The command part of the action specifies that the message will be sent in \texttt{p2} and the buffer will be available.

```
proctype Protocol(port p1; ... ; port p7){
  ...
  do 
  
  \texttt{:(\texttt{full}(m1) \&\& \texttt{full}(p6.sync))} -> \texttt{atomic(} m1\texttt{; m1; put(p6, }\_\texttt{m1);} 
  /\texttt{covered}/

  \texttt{:(\texttt{full}(p2.data) \&\& \texttt{empty}(m2))} -> \texttt{atomic(take(p3, }\_\texttt{p3);m21;}_p3; 
  /\texttt{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 
  \texttt{:(\texttt{full}(m1) \&\& \texttt{full}(p6.sync))} -> \texttt{atomic(} m1\texttt{; m1; put(p6, }\_\texttt{m1);} 
  \texttt{d_step(hassend=1; msg_covReq=1; hasreceive=0; msg_vmazReq=0;}
  \texttt{msg_covered=0; msg_uncovered=0; msg_vmaz=0; brake=0;}
  \texttt{ctrvelo=0;}); /\texttt{covered}/
```

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 \texttt{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 \texttt{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=0))]/\!\!/not satisfied /

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

Listing 9: Flags of the generated code

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

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 lan-
guage 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 se-
manitics 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 seman-
tics.

Similar to our concerns, the CHARMY framework [20] also ad-
dresses 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 prop-
ties 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 com-
position 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.
Veerefy [5–7] is a model checking tool developed at the Uni-
versity of Dresden to analyze and verify Reo connectors. Veere-
ofy 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. Veerefy allows the verification of tem-
poral properties expressed in LTL and CTL-like logics. In contrast
in Veerefy, 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 equiva-
ience 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 con-
nectors. 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 automati-
cally 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 proto-
col 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, reflect-
ing the states of the environment or internal memories, whereas
the latter consist of structural constraints that must hold invari-
ably. 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 involv-
ing 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
memories. A designer then has the freedom to bound part of a
protocol to specify not just a protocol, but also its required LTL
properties, in some suitable constraint language, which our ex-
tended compilation can subsequently translate into Promela LTL state-
ment on protocol states, for verification.

REFERENCES
https://doi.org/10.1016/S0960-1577(04)004153
[4] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. 2007. Models and tem-
poral logical specifications for timed component connectors. Software & Systems
Formal Verification for Components and Connectors. In Formal Methods for Com-
ponents and Objects, Frank S. de Boer, Marcello M. Bonsangue, and Eric Vale
A Uniform Framework for Modeling and Verifying Components and Connectors.
[7] Christel Baier, Tobias Blechmann, Joachim Klein, Sascha Klüppelholz, and Wolf-
gang Leutner. 2010. Design and Verification of Systems with Exogenous Coordi-
nation Using Vereey. In Leveraging Applications of Formal Methodology: Verification,
Validation, and Validation, Tiziana Margaria and Bernhard Steffen (Eds.). Springer Berlin
component connectors in Reo by constraint automata. Science of Computer
Second International Workshop on Foundations of Coordination Languages and
Software Architectures (FOCLASA’03).
[9] Oscar Carrillo, Samir Chouali, and Hassan Mountassir. 2013. Incremental Mod-
eling of System Architecture Satisfying SysML Functional Requirements. In Formal
Aspects of Component Software - 10th International Symposium, FACS 2013, Nan-
chang, China, October 27-29, 2013, Revised Selected Papers. 79–99.
https://doi.org/10.1007/978-3-319-07602-7_7
Channel-
ated 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).
straints. In Coordination Models and Languages, Giavanna Di Marzo Serugendo
and Michele Loreti (Eds.). Springer International Publishing, Cham, 142–161.
272 (08 2018), 121–135.
(first ed.). Addison-Wesley Professional.
2011.02.003 (1) FOCLASA’09 (2) FSEN’09
[17] Nataliia Koshak, Christian Krause, and Erik de Vink. 2012. Reo + mCRL2:
A framework for model-checking dataflow in service compositions. Formal
[18] Nataliia Koshak, Christian Krause, and Erik de Vink. 2010. Verification of
Context-Dependent Channel-Based Service Models. In Formal Methods for Com-
ponents and Objects, Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerste
de, and Michael Leuschel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg,
