Semantic Analysis of Dataflow Control Systems

C. Bernardeschi° and A. Bondavalli°

°Università di Pisa
Dipartimento di Ingegneria dell' Informazione
I-56100 Pisa, Via Diotisalvi 2, Italy
cinzia@iet.unipi.it

° CNUCE - Istituto del CNR
Via S. Maria 36, I-56126, Pisa, Italy
andrea.bondavalli@cnuce.cnr.it

Abstract

In this paper we assume a control system be designed following the dataflow computational model. The dataflow network corresponding to the system design is then transformed into a network of synchronous processes to automatically analyse the semantic behaviour of the system design. We apply existing tools developed for process algebras based specification languages to test the design and to execute model checking of temporal logic formulae on the design. Sometimes, a complete validation of the design can be done by checking that the set of hazardous states of the control system is never reached.

1. Introduction

In the development of control systems, the requirements analysis for both the mission and the critical issues of the system must be done [10]. Critical issues address what the system should not do and allow to concentrate on the elimination and control of hazardous states. The study of critical issues of the system allows us to derive the constraints necessary to guarantee a safe behaviour of the system (safety constraint) and the strategies to realise it (safety strategy). Validation is the activity that aims to check that the actual behaviour of the developed system respects the safety constraint.

Dataflow is a paradigm for describing systems in which a collection of concurrently executing processes (nodes) communicate asynchronously. The computation proceeds in a data driven manner: a node of the network is ready to execute as soon as the required data are available. The semantics of a dataflow network is given by the set of its traces [9], where a trace represents the sequential order in which communications occur with the environment. Dataflow has been proved to be well suited for the design of control
In this paper we specify dataflow networks by LOTOS (Language of Temporal Ordering Specification) [4] process algebras based formal specification language. Using LOTOS allows us to apply the already developed automatic tools to analyse the behaviour of the system design. In particular the simulator tool can be applied to check if a given trace is a possible trace of the system design [7, 8], moreover we can identify the hazardous states of the control system and we can check if these states are never reached by model checking of temporal logic formulae [6, 5, 8].

2. Background and notations

2.1 Dataflow networks

Given a set $V$, let $V^*$ be the finite sequences on $V$ and $\lambda$ be the empty sequence. Given $\alpha \in V^*$, we denote by $\alpha_i$ the $i$-th element of $\alpha$ and $\#\alpha$ the length of the sequence $\alpha$.

**Definition 1** A dataflow node $p$ is a tuple $<I_p, O_p, S_p, R_p>$ where

- $I_p$ is the set of **input channels**;
- $O_p$ is the set of **output channels** with $I_p \cap O_p = \emptyset$;
- $S_p$ is the set of states, $s_p^0$ is the initial state, $s_p^0 \in S_p$;
- $R_p$ is the set of **firings**. A firing is a tuple $<s, \chi_{in}, s', \chi_{out}>$ where $s, s' \in S_p$, $\chi_{in}$ is a mapping from $I_p$ to $V^*$ and $\chi_{out}$ a mapping from $O_p$ to $V^*$.

A firing $F = <s, \chi_{in}, s', \chi_{out}>$ is **enabled** iff the state of the node is $s$ and the content of each input channel $a \in I_p$ starts with the sequence $\chi_{in}(a)$. When a firing is enabled then it can be **executed**: all the sequences $\chi_{in}(a), a \in I_p$ are consumed and the sequences $\chi_{out}(a)$ are produced on each output channel $a \in O_p$ while the node changes its state from $s$ to $s'$.

Given a node $p$ with $I_p = \{a, b\}$ and $O_p = \{c\}$. We denote a firing $F$ with initial state $s$, final state $s'$ and $\chi_{in}(a) = \alpha$, $\chi_{in}(b) = \beta$ and $\chi_{out}(c) = \gamma$ as: $F = <s, [a \rightarrow \alpha, b \rightarrow \beta], s', [c \rightarrow \gamma]>$. Generally a firing may require data from a subset of the input channels and put data on a subset of the output channels. We introduce the following notations:

- $\forall F = <s, \chi_{in}, s', \chi_{out}> \in R_p,
  \quad \text{I}_p|_F = \{a \mid a \in I_p \land \chi_{in}(a) \neq \lambda\}$ and $\text{O}_p|_F = \{a \mid a \in O_p \land \chi_{out}(a) \neq \lambda\}$

- $\forall s \in S_p, R_p|_s$ is the set of firings of $p$ defined on the state $s$:
  \[ R_p|_s = \{F \in R_p, F = <s, \chi_{in}, s', \chi_{out}>\}. \]

**Definition 2** A dataflow network $N$ consists of a set $P_N$ of nodes, such that in $P_N$ each channel occurs at most once as an input channel and at most once as an output channel. The network is obtained by connecting channels with the same name. We denote by:

- $C_N$ the set of channels of $N$: $C_N = \bigcup_{p \in P_N} (I_p \cup O_p)$;
- $I_N$ the set of **input channels**: $I_N = \bigcup_{p \in P_N} I_p - \bigcup_{p \in P_N} O_p$;
- $O_N$ the set of **output channels**: $O_N = \bigcup_{p \in P_N} O_p - \bigcup_{p \in P_N} I_p$;
- $E_N$ the set of **external channels** of $N$: $E_N = I_N \cup O_N$. 


Figure 1: Dataflow Model of the Train Set Example

\[ I_{CNT_i} = \{ es_i, ok_{(i\in\mathbb{N})} \}, \quad O_{CNT_i} = \{ ls_i, res_{(i\in\mathbb{N})}, rel_{(i\in\mathbb{N})} \}, \]
\[ S_{CNT_i} = \{ s, s' \}. \]

The set of firings is: \( R_{CNT_i} = \{ F_1, F_2 \} \) where
\[ F_1 = s, [es_i \rightarrow 1], s, [rel_{(i\in\mathbb{N})} \rightarrow 1], res_{(i\in\mathbb{N})} \rightarrow 1 >, \]
\[ F_2 = s', [ok_{(i\in\mathbb{N})} \rightarrow 1], s, [ls_i \rightarrow 1 >. \]

Figure 2: Dataflow node \( CNT_i \)

We distinguish between external channels and channels which connect two nodes. A communication event occurs when a datum is inserted into an input channel or a datum is removed from an output channel: we denote it by \( (a, v) \), where \( a \in E_N \) and \( v \in V \) is the value exchanged on \( a \).

If channels are unbounded, an input communication event is always enabled, while an output communication event \( (a, v) \) is enabled iff \( v \) is the first element in the channel \( a \).

Let \( Event \) be the set of the events of \( N: Event = \{ (a, v) \mid a \in E_N, v \in V \} \cup \bigcup_{p \in P_N} R_p \).

The network evolves from the initial state in which all the channels are empty and each node is in its initial state by executing events. The semantics of a dataflow net is given by the set of its traces [9], where a trace denoted by \( \theta \), is a finite or infinite sequence of communication events \( (c_1, v_1), (c_2, v_2), \ldots \), with \( c_i \in E_N, v_i \in V \).

An example of data flow net is reported in Figure 1 where
\[ P_N = \bigcup_{i=0}^{\infty} \{ CNT_i, RES_i, SECT_i \}. \]
The network is closed, since \( E_N = \emptyset \).

The dataflow net models a train set example. We suppose a circuit of six section and two trains \( A \) and \( B \) which move unidirectionally on the circuit. The system is composed of two parts: the plant and the controller communicating by sensor and actuator signals.
When a train enters section $i$, the $SECT_i$ sends an $es$ signal to inform the controller that a train entered the section. Only after receiving the $ls$ signal the train can leave the section and move to the next section. At the same time a signal $sn$ is sent from $SECT_i$ to $SECT_{i+1}$ thus simulating the movement of the train ($\oplus$ and $\ominus$ denote the modulo-6 addition and substraction).

The first part of the controller (module $CNT_i$) does the reservation and release of different sections. Module $CNT_i$, which is assigned to $SECT_i$, releases section $i \ominus 2$ (signal $rel_{i(\ominus2)}$) and tries to reserve section $i \oplus 1$ (signal $res_{i(\oplus1)}$). If the reservation is successful it sends the $ls_i$ signal to $SECT_i$.

The second part of the controller (module $RES_i$) behaves like a memory keeping track of the reserved and free sections of the circuit. Receiving a $rel_i$ signal it releases the section, receiving a $res_i$ signal it reserves the section. Of course if a given section is reserved for a train it can not be reserved for another one. The dataflow definition of $CNT_i$ components is reported in figure 2. The reader can refer to [1] for the complete specification of the network.

2.2 LOTOS

LOTOS [4] is process algebras based specification language. The essential component of a LOTOS specification is the “process definition”, where a process is an entity able to interact with its environment through gates. The behaviour of a process is specified by a “behaviour expression”.

The (simplified) syntax we use for a process definition is the following, where $Gatelist$ and $Varlist$ are gate and variable formal parameters:

```plaintext
process Id[Gatelist](Varlist) : noexit :=
  behaviour expression
endproc
```

A behaviour expression is formed out of terms obtained by applying the language operators. This language includes:

- the operator to execute actions in sequence $g;B$;
- the boolean guarded command $[c] \rightarrow B$ which says that only if $c$ is verified, the behaviour specified by $B$ is performed;
- the nondeterministic choice among actions $B_1 || B_2$;
- the hiding of actions hide $g_1, \cdots, g_n$ in $B$, where $g, g_i$ are gates that are transformed into the internal action $i$;
- the parallel composition $B_1 || [g_1, \cdots, g_n] B_2$ which means that $B_1$ and $B_2$ are able to execute any actions that either are ready to perform at a gate different from any of the $g_i$, but they must synchronise on actions at the gates $g_i$ (synchronisation on an empty set of gates is denoted by $||$).
The language includes also: the output action denotation \( g!e \) to send the value expression \( e \) at the channel \( g \) and the input action denotation \( g?x : t \) to receive a value via gate \( g \) and assign this value to the variable \( x \) of the same type \( t \) of the expression.

Processes communicate synchronously by value passing: if we have process \( P \) performing \( g!3 \), process \( Q \) performing \( g?x : \text{nat} \) and the two processes synchronise at \( g \), the result is that the value \( 3 \) is passed to \( Q \) in the variable \( x \). Finally, the selection predicate on an input action \( g?x : t[x = 3] \) constrains the action to occur if and only if the value in the corresponding value declaration is equal to \( 3 \).

3. LOTOS specifications of dataflow networks

Since LOTOS includes constructs to define networks of processes that communicate synchronously, the asynchronism between an output action and the corresponding input action in a dataflow network is simulated by means of a parallel process \( CP \) (channel process) which behaves like an unbounded FIFO buffer. Moreover we define a process \( nodep \) which simulates the behaviour of the node \( p \).

Two different transformation have been defined according to the following classes of dataflow networks:

1. networks made of simple nodes: intuitively a node is simple iff for each state \( s \) of the node, there exists \( C_s \subseteq I_p \) such that the set of firings defined on \( s \) require the same values from the channels in \( C_s \) while firings may differ only for the data required over one of the other input channels of the node;

2. networks whose nodes have no restrictions on the set of firings.

The main difference between the two classes of networks is that: for networks of simple nodes, data can be removed from the input channels \( C_s \) as far as they are available and one of the enabled firings is selected only after all the data from the channels in \( C_s \) have been taken. On the contrary, for general networks the testing data availability phase must be realised by a synchronous mechanism between the node processes and the processes associated to the channels. Each channel process executes a synchronisation action with the node process when the number of tokens required over the channel are available.

Note that from a mathematical point of view the two classes of networks are equivalent, since both of them can compute the same class of functions. The class of simple nodes is obviously less expressive than the other, yet sufficient to specify dataflow networks of control systems. In the following we report the transformation for networks of simple nodes.

For each channel \( a \in C_N \), we define two gates: \( a \) and \( a^\# \), which correspond to the input and output gate of the \( CP \) process related to channel \( a \). In the following we denote by:

\[
C_{\text{gates}} = \{ a, a \in C_N \} \cup \{ a^\#, a \in C_N \}; \quad E_{\text{gates}} = \{ a, a \in I_N \} \cup \{ a^\#, a \in O_N \}; \\
\forall p \in P_N, I_{p^\#} = \{ a^\#, a \in I_p \} \text{ and } O_p = \{ a, a \in O_p \}.
\]

An example of transformation is reported in figure 3. Let \( t \) be the type of the values in \( V \), if we assume the queue data type be defined in the specification with Append, Next
and First the standard operations over a queue, the channel process can be specified as follows:

\[
\text{process } CP[\text{inp}, \text{out}](ll : \text{queue}, \text{len} : \text{nat}) : \text{noexit} := \\
\begin{cases}
\text{inp}?:x : t; CP[\text{inp}, \text{out}](\text{Append}(x, ll), \text{len} + 1) & \text{if } \text{len} = 0 \\
\text{out}!\text{First}(ll); CP[\text{inp}, \text{out}](\text{Next}(ll), \text{len} - 1)) & \text{if } \text{len} > 0
\end{cases}
\]

endproc

Let \( S_p = \{s_1, \ldots, s_n\} \), \( C_s = \{a_1, \ldots, a_n\} \) and \( R_p \mid s = \{F_1, \ldots, F_h\} \). By definition, \( \forall a \in C_s \), \( \forall F_i, F_j \in R_p \mid a \) we have that \( F_i \) and \( F_j \) remove the same sequence of values from the channels in \( C_s \). We denote it by \( \chi_{in}(a) \). The definition of the node \( p \) is:

\[
\text{process } nodep[I_p^\#, O_p](\text{st} : \text{state}) : \text{noexit} := \\
\begin{cases}
\text{st} = s_1 & \Rightarrow Bexs_1; \\
\cdots \\
\text{st} = s_n & \Rightarrow Bexs_n
\end{cases}
\]

endproc

where, for each state \( s_i \), \( Bex_{s_i} \) is the sequence of actions to get data from the channels in \( C_s \) in a sequential way followed by the behaviour expression to operate a nondeterministic choice among the firings for which all necessary data have been received and the firings for which the token on the supplementary input channel is available followed by the actions to complete the execution the selected firing (the internal computation represented by the LOTOS internal action \( i \) and the output of the sequences over the output channels) and the instantiation of the process \( nodep \) with the new state after the firing as parameter. If no firings are defined on \( s \), \( Bex_s \) is the behaviour expression \( \text{stop} \).

\[
Bex_s = Bexinp_{a_1}; \cdots; Bexinp_{a_n} (BexExec_{F_1} \\
\cdots \\
(BexExec_{F_h})
\]

where \( Bexinp_a \) is the behaviour expression to get data from input channel \( a \):

\[
Bexinp_a = a^?_1X_1 : t[X_1 = \chi_{in}(a)_1]; \cdots; a^?_nX_{\chi_{in}(a)_n} : t[X_{\chi_{in}(a)} = \chi_{in}(a)_{\chi_{in}(a)}].
\]

If \( F = <s, \chi_{in}, s', \chi_{out}> \) requires values only from \( C_s \) \( (I_p \mid F = C_s) \) and \( O_p \mid F = \{b_1, \ldots, b_n\} \) we have: \( BexExec_{F} = i; Bexout_{b_1}; \cdots; Bexout_{b_m}; nodep[I_p^\#, O_p](s') \)

where \( Bexout_b = b\chi_{out}(b)_1; b\chi_{out}(b)_2; \cdots; b\chi_{out}(b)_{\chi_{out}(b)} \).

If \( I_p \mid F = \{C_s \cup \hat{a}\} \), we have that

\[
BexExec_{F} = \hat{a}; X : t[X = \chi_{in}(\hat{a})]; i; Bexout_{b_1}; \cdots; Bexout_{b_m}; nodep[I_p^\#, O_p](s')
\]

where \( BexExec_{F} \) corresponds to execute the action to get the data item from \( \hat{a} \) followed by the internal action \( i \) followed by the actions to put data on the output channels. The LOTOS process definition of the dataflow node \( CNT_i \) is reported in figure 4.
\textbf{process} nodeCNT_i[es; res_{i}^{[#]}, ok_{i}^{[#]}, rel_{i}^{[#]}, ls_{i}](st : state) : \textbf{noexit} :=

\texttt{([st = s] \rightarrow es;^{[#]}x : nat; i; rel_{i}^{[#]}!1; res_{i}^{[#]}!1;)

\texttt{nodeCNT_i[es; res_{i}^{[#]}, ok_{i}^{[#]}, rel_{i}^{[#]}, ls_{i}](s')

\texttt{([st = s'] \rightarrow ok_{i}^{[#]}? : nat; i; ls_{i}!1; nodeCNT_i[es; res_{i}^{[#]}, ok_{i}^{[#]}, rel_{i}^{[#]}, ls_{i}](s))

endproc

Figure 4: Process corresponding to $CNT_i$

The structure of the LOTOS specification of the network $N$ is:

\textbf{process} net_N[\textit{E gates}_N] : \textbf{noexit} :=

\textbf{hide} (\textit{C gates}_N - \textit{E gates}_N) \textbf{in}

\texttt{(CP[a, a'][(\textit{Create}, 0)] || \ldots \forall c \in C_N \ldots || CP[b, b'][(\textit{Create}, 0)])

\texttt{||(\textit{C gates}_N - \textit{E gates}_N)]

\texttt{(nodep[I_{p}^{[#]}, O_{p}](s_{p}^{0}) || \ldots \forall r \in P_N \ldots || nodeq[I_{q}^{[#]}, O_{q}](s_{q}^{0})

endproc

where \textit{Create} is the operation which creates an empty queue.

Channel processes operate on disjoint gates, they are put in parallel with an empty set of synchronisation gates.

Similarly, node processes operate on disjoint gates, they are put in parallel with an empty set of synchronisation gates. The two behaviour expressions synchronise on the intersection of their observable gates ($\textit{C gates}_N - \textit{E gates}_N$).

Furthermore, we hide the actions at the gates in ($\textit{C gates}_N - \textit{E gates}_N$), making observable only the actions at the external channels of the network. The reader can refer to [2] for the complete definition of the transformations.

Given a dataflow network $N$ and the corresponding LOTOS specification $net_N$, we denote by $\mathcal{F}$ the function which maps each input (output) communication event $e \in \textit{Event}$ in the corresponding LOTOS action in $net_N$. We consider $\mathcal{F}$ naturally extended to sequences of communication events. For both the transformations it has been proved the following theorem.

\textbf{Theorem 1} Let $N$ be a dataflow network and $\Theta$ be the set of traces of $N$ (each trace is a sequence of communication events). If $net_N$ is the LOTOS specification obtained by the transformation, the set $S = \{ \mathcal{F}(\theta), \theta \in \Theta \}$ is equal to the observable behaviour of $net_N$.

4. Safety validation of the design of a control system

Let us consider the dataflow net of Figure 1. Once the dataflow network has been transformed into a LOTOS specification, the LOTOS integrated tool environment LITE [8], allows us to execute different kind of analyses of the specification.
The safety validation of the controller is obtained by dataflow networks semantic analysis. On open networks, for each hazardous state of the system we can derive the traces which could lead to that particular state. If we consider only the right side of the dataflow net of figure 1 which corresponds to the controller, one possibility is to analyse the network specification by testing: we can select a test \( T \) and check if it is successfully executed on it.

The selection of the test must take into account the traces leading to hazardous states.

The test \( T \) may have a trace of actions terminated by a \textit{Success} action. Let \( A = \{a_1, \ldots, a_n\}, A \subseteq (I_N \cup O^+_N) \), we have:

\begin{verbatim}
process T[A, Success]: noexit := a_1; \ldots; a_n; Success; stop endproc
\end{verbatim}

The application of a test to a specification can be represented as the parallel composition of the specification with the test process, synchronising on the union of their gates, except for the \textit{Success} action:

\[
netN[I_N, O^+_N] \parallel [A] T[A, Success].
\]

The test terminates successfully if there exists at least one trace corresponding to the sequence of events.

A refusal test can be applied to determine if a set of actions is rejected after a given trace. Let \( A = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}, A \subseteq (I_N \cup O^+_N) \), we have:

\begin{verbatim}
process T[A, Success]: noexit := a_1; \ldots; a_n; (b_1; stop
\[\cdots\]
\[b_m; stop\]
\[Success; stop\]
endproc
\end{verbatim}

Let \( \{b_1, \ldots, b_m\} \) be the set of actions to be rejected. If any of the actions to be rejected were accepted, the process \( T \) stops without having executed the \textit{Success} action.

Moreover, in control systems design, some assumption can be done on the external environment. The input sequences are in general periodic and repetitive in essence. In this case we can specify a process \( Ev \) which simulate the environment, we can combine the network specification in parallel with \( Ev \) with synchronisation on the union of their gates and then we can simulate the specification. This corresponds to test the behaviour of the dataflow network when reacting to stimuli from the environment as stated by process \( Ev \). If we can specify the behaviour of the plant in a similar way, then the plant can be used as a test case for the validation of the controller. The LOTOS specification of the plant can be composed in parallel with the controller to show whether hazards are reached (see figure 1).
Finally, dataflow network analysis is strongly limited by the fact that the behaviour is dependent on the value of data which are input of the network. If we consider dataflow networks in which the set of values is limited or the only relevant data is the presence or absence of signals, independent of the actual values, the LOTOS specification of the network can be given in basic LOTOS (the subset of LOTOS without data). On basic LOTOS several verification activities are possible by automatic tools [8]: including equivalence checking, graphical display of the automaton corresponding to the system behaviour and model checking of temporal logic formulae [5] describing the expected behavioural properties of the system.

In the following we show how the design of a controller for the contact-free moving of trains over a circular track can be validated by model checking of temporal logic formulae.

If we assume the length of a train less than any section, the hazardous states are the states in which the front of one train is in the same or adjacent section as the front of the other train. Hazards are avoided if the following safety condition always holds: the heads of the trains differ for at least two sections. In the design we used a reservation system. A train reserves always two sections for itself: the section occupied by the head of the train and the previous one. When a train has to move to a new section, it has reserved three sections.

In the example we distinguish the train entering a section: 

- \( asn_i \) (\( bsn_i \)) is the action of 
  train \( A \) (\( B \)) when it is entering section \( SECT_i \) and we derive a specification in basic LOTOS (a subset of LOTOS without data) observational equivalent to the previous one [1]. For each value exchanged at a gate, we define a different gate in the specification. Finally, since LOTOS allows to hide actions at some gates and to make visible actions at other gates by the hiding operator, we make observable the actions corresponding to the movement of the trains over the track (gates \( asn_i \) and \( bsn_i \)).

The LOTOS behavioural analyser AUTO [6] allows us to build the automaton of the specification. The automaton has 18 states and 24 transitions. Moreover, we proved automatically, by using the LOGIC CHECKER [5] over the automaton, the following logic formulae to be true for train \( A \):

- train \( A \) can enter any section: 
  \( A[\text{true}\{\text{true}\}]U\{asn_i \}\text{true}\}; \)
- train \( A \) can only move from section \( i \) to section \( i \oplus 1 \):
  \( AG[\text{true}\{\text{cond}\}]U\{-asn_i \}\text{true}\}, \)
  where \( \text{cond} = ((\neg asn_0) \land (\neg asn_1) \land (\negasn_2) \land (\negasn_3) \land (\negasn_4) \land (\negasn_5)); \)
- for each path such that train \( A \) enters section \( i \), train \( B \) cannot enter section \( i \oplus 1 \) until train \( A \) enters section \( i \oplus 1 \):
  \( AG[\text{true}\{\neg bsn_{i\oplus1}\}U\{bsn_{i\oplus1}\}A[\text{true}\{\neg bsn_{i\oplus1}\}U\{asn_{i\oplus1}\}\text{true}\}]. \)

The same formulae can be proved to be true for train \( B \). From this we have that when train \( A \) is in section \( i \), train \( B \) is never in section \( i \oplus 1, i, i \oplus 1 \). This holds also for train \( B \), thus satisfying the safety condition.
5. Conclusions

In this paper we give emphasis on the possibility of using automatic verification tools for the validation of dataflow based control system designs. We proved on an example that hazardous states are never reached during the execution of the system by validating the process algebras specification corresponding to the dataflow design. The assumptions about the behaviour of the physical environment and the condition that the set of data exchanged is generally limited, allow us to validate the design by proving that temporal logic formulae corresponding to the safety strategy are always satisfied. Testing is instead always applicable also on open data flow networks.

References


