A Probabilistic Verification Framework of SysML Activity Diagrams

Samir Ouchani\textsuperscript{1,2}, Otmane Ait Mohamed\textsuperscript{2}, Mourad Debbabi\textsuperscript{1}

\textsuperscript{1}Computer Security Laboratory (CSL), Concordia University, Montreal, Quebec, Canada H3G 1M8
\textsuperscript{2}Hardware Verification Group (HVG), Concordia University, Montreal, Quebec, Canada H3G 1M8

Email: \{s_oucha,ait,debbabi\}@encs.concordia.ca

Abstract—SysML activity diagrams are OMG/INCOSE standard used for modeling and analyzing probabilistic systems. In this paper, we propose a formal verification framework that is based on PRISM probabilistic symbolic model checker to verify the correctness of these diagrams. To this end, we present an efficient algorithm that transforms a composition of SysML activity diagrams to an equivalent probabilistic automata encoded in PRISM input language. To clarify the quality of our verification framework, we formalize both SysML activity diagrams and PRISM input language. Finally, we demonstrate the effectiveness of our approach by presenting a case study.

Index Terms—SysML Activity Diagram, Probabilistic Automata, PRISM Model-Checker, Probabilistic Verification, PCTL.

I. INTRODUCTION

SysML [17] is a standard modeling language used for system applications. It reuses a subset of UML packages [18] and extends others with specific systems’ engineering features such as probability. Mainly, it covers four perspectives of systems modeling: structure, behavior, requirement, and parametric diagrams. Particularly, SysML activity diagrams are behavioral diagrams used to model system’s behavior at various levels of abstraction [13]. To verify these diagrams, the most popular technique used is model checking [7], [19]. Model checking is a formal automatic verification technique for finite state concurrent systems that checks temporal logic specifications on a given model. In addition to qualitative verification, quantitative verification techniques based on probabilistic model checkers [3] have recently gained popularity. Probabilistic model checking offers the capability of evaluating the probability of satisfying a given property on systems that inherently exhibit probabilistic behavior. After a comparison study between the existing probabilistic model checker, we selected PRISM [16] for our work.

PRISM checks probabilistic specifications over probabilistic models. The specifications can be expressed either by a probabilistic computation tree logic (PCTL) [3], [10] or a continuous stochastic logic. The models can be described using PRISM language as discrete-time Markov chains, continuous-time Markov chains, Markov decision processes (MDPs) or probabilistic timed automata. PRISM supports probabilistic automata (PAs), but for simplicity, PRISM refers to a PA by MDP [10]. For the verification efficiency, the constructed models can be stored as binary decision diagrams (BDDs) and multi-terminal BDDs (MTBDDs). In addition, PRISM has built-in symmetry reduction and implements iterative numerical computations to avoid the state-space explosion problem.

Related Work. In the literature, some works target the verification of UML activity diagrams and others target SysML activity diagrams. Here, we are interested to both of them while only few initiatives focus on SysML behavioral diagrams. R. Eshuis [9] translates an activity diagram to NuSMV code to verify model requirements expressed in LTL. Das et al. [6] present a timing verification of activity diagrams where they rely the diagram edge to a clock. Rafe et al. [21] determine the formal semantics of UML2.0 activity diagrams by Graph Transformation Systems (GTS), but, they didn’t show which verification approach or tool can support CST. Paolo et al. [4] specify and verify an airport case study described by a class diagram and an activity diagram. Federico et al. [1] map an activity diagram to COW process calculus. Raida et al. [8] transforms an UML activity diagram to CSP expressions using ATOM tool. Ermeson et al. [2], [5] map SysML state machine and activity diagram into time Petri nets with energy constraints. Jansen et al. [14], [15] extend an extension of UML statecharts with a randomly varying duration. The model is represented as MDP, and the properties ares expressed in PCTL. Debbabi et al. [7] map a UML activity diagrams into NuSMV input language, and, they use PRISM to verify a single SysML activity diagram. They consider DTMCs as a semantic model in the presence of time and as MDPs when time is absent. It is well-known that DTMCs are fully probabilistic models and MDPs do not support the internal nondeterminism [22]. Compared to the existing works, our contribution is innovative by presenting a new verification approach for a composition of SysML activity diagrams. In addition, it uses the adequate semantic model for the appropriate model checking tool.

Framework. In this paper, we are interested in the formal verification of probabilistic systems modeled as SysML activity diagrams. These diagrams can call and communicate with other diagrams and allow for probabilistic behavior specification. The overview of our proposed framework is depicted in Figure 1. It takes a composition set of SysML activity diagrams and PCTL properties as inputs to produce the satisfiability probability of the PCTL property on diagrams.

Our framework is based on transforming a set of SysML
activity diagrams composed by call behavior and send/receive artifacts to an equivalent probabilistic automata that is encoded PRISM input language. Then, PRISM model checker can verify the PCTL property on the resulted probabilistic automata to produce the satisfiability results. In summary, the main contributions of this paper are: 1) Formalizing SysML activity diagrams, 2) Formalizing PRISM models, 3) Proposing a verification framework, 4) Implementing and applying our approach on a real system.

**Organization.** The remainder of this paper is organized as follows. Section II and Section III describe and formalize SysML activity diagrams and PRISM models, respectively. Our verification framework is detailed in Section IV and Section V describes the experimental results. Finally, Section VI concludes this paper and provides the future works.

II. **SysML Activity Diagrams Formalization**

As illustrated in Figure 2, the main artifacts of a SysML activity diagrams are a graph-based model where activity nodes are connected by activity edges. Nodes have three types: activity invocation, object and control nodes. Activity invocation includes receive and send signals (or objects), actions, and call behaviors. Activity control nodes can be initial, flow final, activity final, decision, merge, fork, and join nodes. Activity edges are of two types: control flow and object flow. Control flow edges are used to show the execution path through the activity diagram, and, object flow edges are used to show the flow of data between activity nodes. Concurrency and synchronization are modeled using forks and joins, whereas, branching is modeled using decision and merge nodes. While a decision node specifies a choice between different possible paths based on the evaluation of a guard condition (and/or a probability distribution), a fork node enables more than one guard; the nondeterminism mechanism is adopted.

We present in Definition 1, the formal definition of SysML activity diagrams. Then, we propose two properties that shape the structure of diagrams.

**Definition 1 (SysML Activity Diagram):** A SysML activity diagram is a tuple $A = (t, fn, \mathcal{N}, \mathcal{E}, Prob)$, where:

- $t$ is the initial node,
- $fn = \{\bigcirc, \otimes\}$ is the set final nodes,
- $\mathcal{N}$ is a finite set of activity nodes,
- $\mathcal{E}$ is a finite set of activity edges,
- $Prob : (\{t\} \cup \mathcal{N}) \times \mathcal{E} \rightarrow Dist(\mathcal{N} \cup \text{fin})$ is a partial probabilistic transition function that assigns for each node a discrete probability distribution $\mu \in Dist(\mathcal{N} \cup \text{fin})$.

**Property 1 (Structure Constraint):** For a SysML activity diagram $A$, let $m$ be the number of edges, and $n$ is the number of nodes. Then: $m > n$. initially, when a SysML activity diagram is invoked, then, its initial node is activated. The activation of any other node depends only on the deactivation of its preceded node and the guard satisfaction of its input edge. In addition, the call behavior action or the decision node can consume its input tokens and invoke its specified behavior. In this case, the invocation supports both synchronous and asynchronous calls. In the asynchronous case, the execution of the invoked behavior proceeds without any further dependency on the execution of the activity containing the invoking artifact. But in the synchronous case, the execution of the calling artifact is blocked until it receives a reply token from the invoked behavior. For the case of decision nodes, when the invoked behavior enables more than one guard; the nondeterminism mechanism is adopted.

We present in Definition 1, the formal definition of SysML activity diagrams. Then, we propose two properties that shape the structure of diagrams.

**Definition 1 (SysML Activity Diagram):** A SysML activity diagram is a tuple $A = (t, fn, \mathcal{N}, \mathcal{E}, Prob)$, where:

- $t$ is the initial node,
- $fn = \{\bigcirc, \otimes\}$ is the set final nodes,
- $\mathcal{N}$ is a finite set of activity nodes,
- $\mathcal{E}$ is a finite set of activity edges,
- $Prob : (\{t\} \cup \mathcal{N}) \times \mathcal{E} \rightarrow Dist(\mathcal{N} \cup \text{fin})$ is a partial probabilistic transition function that assigns for each node a discrete probability distribution $\mu \in Dist(\mathcal{N} \cup \text{fin})$.

**Property 1 (Structure Constraint):** For a SysML activity diagram $A$, let $m$ be the number of edges, and $n$ is the number of nodes. Then: $m > n$. Initially, when a SysML activity diagram is invoked, then, its initial node is activated. The activation of any other node depends only on the deactivation of its preceded node and the guard satisfaction of its input edge. In addition, the call behavior action or the decision node can consume its input tokens and invoke its specified behavior. In this case, the invocation supports both synchronous and asynchronous calls. In the asynchronous case, the execution of the invoked behavior proceeds without any further dependency on the execution of the activity containing the invoking artifact. But in the synchronous case, the execution of the calling artifact is blocked until it receives a reply token from the invoked behavior. For the case of decision nodes, when the invoked behavior enables more than one guard; the nondeterminism mechanism is adopted.

We present in Definition 1, the formal definition of SysML activity diagrams. Then, we propose two properties that shape the structure of diagrams.

**Definition 1 (SysML Activity Diagram):** A SysML activity diagram is a tuple $A = (t, fn, \mathcal{N}, \mathcal{E}, Prob)$, where:

- $t$ is the initial node,
- $fn = \{\bigcirc, \otimes\}$ is the set final nodes,
- $\mathcal{N}$ is a finite set of activity nodes,
- $\mathcal{E}$ is a finite set of activity edges,
- $Prob : (\{t\} \cup \mathcal{N}) \times \mathcal{E} \rightarrow Dist(\mathcal{N} \cup \text{fin})$ is a partial probabilistic transition function that assigns for each node a discrete probability distribution $\mu \in Dist(\mathcal{N} \cup \text{fin})$.

**Property 1 (Structure Constraint):** For a SysML activity diagram $A$, let $m$ be the number of edges, and $n$ is the number of nodes. Then: $m > n$. Initially, when a SysML activity diagram is invoked, then, its initial node is activated. The activation of any other node depends only on the deactivation of its preceded node and the guard satisfaction of its input edge. In addition, the call behavior action or the decision node can consume its input tokens and invoke its specified behavior. In this case, the invocation supports both synchronous and asynchronous calls. In the asynchronous case, the execution of the invoked behavior proceeds without any further dependency on the execution of the activity containing the invoking artifact. But in the synchronous case, the execution of the calling artifact is blocked until it receives a reply token from the invoked behavior. For the case of decision nodes, when the invoked behavior enables more than one guard; the nondeterminism mechanism is adopted.

We present in Definition 1, the formal definition of SysML activity diagrams. Then, we propose two properties that shape the structure of diagrams.
<table>
<thead>
<tr>
<th>Activity Artifacts</th>
<th>Formal Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Initial node" /></td>
<td>$I: \cdot \rightarrow \mathcal{N}$</td>
<td>Initial node is activated when a diagram is invoked.</td>
</tr>
<tr>
<td><img src="image2" alt="Activity final node" /></td>
<td>$I: \otimes$</td>
<td>Activity final node stops the execution of the whole diagram.</td>
</tr>
<tr>
<td><img src="image3" alt="Receive node" /></td>
<td>$a \xrightarrow{} \mathcal{N}$</td>
<td>Receive node is used to receive a signal or an object.</td>
</tr>
<tr>
<td><img src="image4" alt="Call behavior node" /></td>
<td>$a \uparrow \mathcal{A} \rightarrow \mathcal{N}$</td>
<td>Call behavior node invokes a new behavior related to an action.</td>
</tr>
<tr>
<td><img src="image5" alt="Merge node" /></td>
<td>$\alpha \cap \mathcal{N}$</td>
<td>Merge node specifies the continuation.</td>
</tr>
<tr>
<td><img src="image6" alt="Decision node" /></td>
<td>$\Delta(x_1,x_2) \rightarrow \mathcal{N}$</td>
<td>Decision node with a call behavior $\mathcal{A}$, a convex distribution ${p, 1-p}$, and guarded edges ${g, \neg g}$.</td>
</tr>
<tr>
<td><img src="image7" alt="Flow final node" /></td>
<td>$\mu \rightarrow \mathcal{N}$</td>
<td>Flow final node kills only the related token.</td>
</tr>
<tr>
<td><img src="image8" alt="Action node" /></td>
<td>$a \rightarrow \mathcal{N}$</td>
<td>Action node defines an atomic action.</td>
</tr>
<tr>
<td><img src="image9" alt="Send node" /></td>
<td>$a \rightarrow \mathcal{N}$</td>
<td>Send node is used to send a signal or an object.</td>
</tr>
<tr>
<td><img src="image10" alt="Join node" /></td>
<td>$\alpha \cap \mathcal{N}$</td>
<td>Join node presents the synchronization.</td>
</tr>
</tbody>
</table>

**TABLE 1**

**Formal Notation of SysML Activity Diagram Artifacts.**

**Property 2 (Token Constraint):** In a SysML activity diagram $\mathcal{A}$, let $m$ represents the number of edges, and $k$ is the number of tokens. Then: $k < m$.

**III. PRISM Formalization**

In this section, we formalize PRISM by focusing more on probabilistic automata that considered as the appropriate semantics model for SysML activity diagrams [20].

Generally, a probabilistic system “$\mathcal{S}$” described as a PRISM program “$\mathcal{P}$” comprises a set of $n$ modules ($n > 0$), the state of each one is defined by an evaluation of a countable set of finite-ranging local variables. The global state of the system is the evaluation of the local variables ($V_L$) and the global ones ($V_G$) denoted by $V = V_G \cup V_L$. The behavior of each module is defined by a set of commands.

A module describes the main changes of $\mathcal{P}$ behaviors. It takes the following form: $[\alpha] \ g \rightarrow p_1 : u_1 + \cdots + p_m : u_m$, or, $[\alpha] \ g \rightarrow u$, which mean, for the action “$\alpha$” if the guard “$g$” is satisfied, then, an update “$u_i$” is enabled with a probability “$p_i$”. The guard “$g$” is a predicate consists of boolean variables and propositional logic operators. The update “$u_i$” is an evaluation of variables expressed as a conjunction of assignments: $(v_1 = val_1 \land \cdots \land v_k = val_k)$ where $v_i$ are local variables and $val_i$ are values evaluated via expressions denoted by “eval” that require type consistency (eval : $V \rightarrow \mathbb{N} \cup \{true, false\}$).

The formal definition of a command is given in Definition 2.

**Definition 2 (PRISM Command):** A PRISM command “$c$” is a tuple $c = (\alpha, g, u)$, where:

- $\alpha$: is an action label,
- $g$: is a logic predicate over $V$,
- $u = \{(p_i, u_i)\} \exists m > 1, i < m, p_i > 0, \sum_{i=1}^{m} p_i = 1$ and $u_i = \{(v, eval(V)) : v \in V_L\} \cup \{(v, eval(V)) : v \in V_L\}$.

A module that describes the behavior of a sub-part of a system can be considered as a set of commands. The variables of each module are declared and initialized locally as follows “$T \ v init \ val;”$ which means the variable $v$ of type $T$ is initialized by the value $val$. A module is formally defined in Definition 3.

**Definition 3 (PRISM Module):** A PRISM module “$M$” is a tuple $M = (V_L, I, C)$, where:

- $V_L$ is a finite set of local variables of the module $M$,
- $I$ is the initial values of $V_L$,
- $C = \{c_i : 0 \leq i \leq k\}$ is a finite set of commands that defines the behavior of $M$.

Generally, PRISM modules are combined using five Communicating Sequential Processes (CSP) [12] operators are:

1) Synchronization: It is a parallel composition of modules. For two modules $M_1$ and $M_2$, their synchronization is denoted by $M_1 || M_2$ and they can synchronize only on actions appearing in both $M_1$ and $M_2$.

2) Interleaving: It is an asynchronous parallel composition of modules that are fully interleaved without synchronization. The module $M_1$ interleaves with $M_2$ is denoted by $M_1 || M_2$.

3) Parallel Interface: It is a restricted parallel composition of modules that synchronize only on shared actions. For the set of shared actions $\{\alpha, \beta, \cdots\}$ between modules $M_1$ and $M_2$, we denote the parallel interface by: $M_1 || (\alpha, \beta, \cdots) || M_2$. 


4) Hiding: This operation permits to hide actions in a module. We denote by $M/(\alpha, \beta, \cdots)$ to hide actions $\alpha$, $\beta$, $\cdots$ in the module $M$.

5) Renaming: This operator facilitates rewriting the behavior of a module by renaming its actions. We denote by $M\{\alpha_1 \leftarrow \alpha_2, \beta_1 \leftarrow \beta_2, \cdots\}$ to rename actions $\alpha_1$ by $\alpha_2$, $\beta_1$ by $\beta_2$, $\cdots$ in the module $M$.

As a result, Definition 4 stipulates formally a system containing $n$ modules and combined by a CSP expression.

**Definition 4 (PRISM System):** A PRISM system is a tuple $\mathcal{P} = (V, I_G, M, \text{sys})$, where:

- $V = V_G \cup \bigcup_{i=1}^{l} V_{\alpha_i}$ is a finite set of the union of global and local variables,
- $I_G$ is the initial values of $V_G$,
- $M$ is a countable set of modules,
- $\text{sys}$ is a CSP algebraic expression that defines the combination between modules in $M$.

IV. THE VERIFICATION APPROACH

In this section, we express in PRISM a SysML activity diagram with call behaviors. To encode the diagram $\mathcal{A} = \mathcal{A}_1 \uparrow \alpha, \mathcal{A}_2$ into PRISM code, we rely to the MDP formalism that refers to PA$^2$ which coincides with SysML structure defined in Definition 1. We define each SysML transition $s \xrightarrow{l} \mu(s)$ in PRISM as one command. Mainly, a probabilistic command takes the following form: $[l] g \rightarrow u_1 : u_1 + \cdots + u_m : u_m$, which means, for the action “$l$” if the guard “$g$” is true, then, an update “$u$” is enabled with a probability “$p_l$”. In the case of a Dirac transition, a command is written simply by: $[l] g \rightarrow u$. The guard “$g$” is a predicate with a conjunction form consisting of the evaluation of the atomic propositions related to the state $s$. The update $u_i$ describes the evaluation of the atomic propositions related to the next state $s_i$ of $s$ such that $s \xrightarrow{l_i} p_i, s_i$.

A given diagram is considered as a PRISM module which is a set of commands. The variables of each module are locals of type boolean initialized to false except for the initial node that is initialized to true. It marks the first token produced by the initial node. Listing 1 describes the diagram composition $\mathcal{A}_1 \uparrow \alpha, \mathcal{A}_2$ in PRISM as follows, where $\mathcal{A}_1$ and $\mathcal{A}_2$ have $n$ and $m$ nodes, respectively. For a diagram $\mathcal{A}_i$, the node $j$ is referenced by the boolean proposition $l_j$ and by $l_{N_j}$ for its successor node. The variables of the module $\mathcal{A}_1$ (line 3-29) are initialized (line 5-11). Its probabilistic decision is presented in line 22 and its Dirac transition is given by line 25. The call behavior transitions (line 18 and 19) synchronize with the initial (line 39) and the final (line 50) transitions of $\mathcal{A}_2$ on the shared actions $l_{t_1}$ and $l_{t_2}$, respectively. The final transitions of $\mathcal{A}_1$ and $\mathcal{A}_2$ (line 28 and 50) express the final node. It initializes all local variables to false which returns $\mathcal{A}$ to its static form.

V. IMPLEMENTATION AND EXPERIMENTAL RESULTS

In this section, we apply our verification framework on an online shopping system case study [11]. The related SysML activity diagrams are modeled on Topcased$^2$ then mapped into Prism code via our Java implementation. In the purpose of providing experimental results demonstrating the efficiency and the validity of our approach, we verify four system functional requirements.

A. Online Shopping System

The online shopping system aims at providing services for purchasing online items. Figure 3 illustrates the corresponding

---

SysML activity diagram. It contains four call-behavior actions, which are: “Browse Catalogue”, “Make Order”, “Process Order” and “Shipment”. And, each call-behavior action is represented by its proper diagram. As example, Figure 4 expands the call behavior action “Process Order”. Listing 2 refers to the generated PRISM fragment code related to the studied system.

```
const int K;
mdp
// The main module of Online Shipping System
module main
initial: bool init true; F1: bool init false;
M1: bool init false; M2: bool init false;
M3: bool init false; M4: bool init false;
J1: bool init false; ReceiveOrder: bool init false;
ProcessOrder: bool init false; SendReport: bool init false;
ReceiveReport: bool init false; ConfirmOrder: bool init false;
D1: bool init false; D2: bool init false; D3: bool init false;
F2: bool init false; i1: bool init false;
F1 = false; F2 = false; F3 = false; F4 = false;
M1' = F1'; M2' = F2'; M3' = F3'; M4' = F4';
J1' = i1';

// The module of making Order
module MakeOrder
InitialMake: bool init false; ReceiveDeliveryReq: bool init false;
CheckInventory: bool init false; Davail: bool init false;
itemavail: bool init false; nitemavail: bool init false;
DispOutItem: bool init false; ReserveOrder: bool init false;
D1: bool init false; D2: bool init false; D3: bool init false;
F1: bool init false; F2: bool init false; F3: bool init false;

// The module of Processing Order
module ProcessOrder
initialprocess: bool init false; CheckAcccount: bool init false;
GetAccount: bool init false; CreateAccount: bool init false;
AuthorizeCard: bool init false; CardOk: bool init false;
NotCardOk: bool init false; D4: bool init false; CreateDelivery: bool init false;
EndProcessing: bool init false; m: [0..K];

// The module of Browse Catalogue
module BrowseCatalogue
InitialCatalogue: bool init false; RequestCatalogue: bool init false;
DisplayCatalogue: bool init false; SelectItem: bool init false;
DisplayPrice: bool init false; EndCatalogue: bool init false;
```

Due to the space limitation, we exclude the call behavior diagrams.

---

**Fig. 3.** The Online Shopping System SysML Activity Diagram.

**Fig. 4.** SysML Activity Diagram of Process Order Behavior.
In order to prove the correctness of the online shopping system, we propose to verify four functional requirements. They are expressed in PCTL as follows where $n \in [0..K]$ and $m$ represent the order and the shipment numbers, respectively.

1) For each order, what is the minimum probability value to make a delivery?

PCTL: $P_{min} = \lceil n \leq K \rceil \ (\text{Delivery})$.

2) After browsing the catalogue, what is the minimum probability value to ship a selected item?

PCTL: $P_{min} = \lceil \text{SelectItem} \land m = n \land m \leq K \rceil \Rightarrow F(\text{Delivery}) \Rightarrow F(\text{Shipment})$.

3) For a given customer, what is the maximum probability value to make a new order after confirming his bill?

PCTL: $P_{max} = \lceil \text{CardOk} \Rightarrow F(\text{MakeOrder}) \rceil$.

4) For each order, what is the maximum probability value to enter a wrong code?

PCTL: $P_{max} = \lceil n = m \Rightarrow F(\text{ConfBill}) \rceil$.

The verification results of the above four properties are done using a 17 CPU 9206@2.67GHz with 12.0GB of RAM. For different values of the maximum number of orders ‘K’, Property 1 converges to 0.9977, Property 2 decrementing quickly to a steady-state value of 0.0420, Property 3 and Property 4 converge to 0.9622 and to 0.9977, respectively.

VI. CONCLUSION

In this paper, we presented a formal verification framework to improve the requirement checking of systems modeled by using SysML activity diagrams. To verify these diagrams, we devised an algorithm that transforms SysML activity diagrams into the input language of the probabilistic model checker PRISM. We formalized SysML activity diagrams and PRISM language that help to explain easily our verification framework. Finally, we demonstrated the effectiveness of our approach by applying it on a case study representing an online shopping system. The proposed framework could form the platform of any future work based on the verification of SysML and UML behavioral diagrams. The presented work can be extended in the following three directions. First, we intend to formalize SysML and UML state machines and sequence diagrams then extend our framework to support them. Second, we target to develop a model-based templates to facilitate expressing temporal logic formulae of requirements for the studied diagrams. Finally, we want prove the soundness of the proposed approach, and validate it on different case studies and real system models.

REFERENCES


Listing 2. The generated PRISM Code for SOS.

```
(InitialCatalogue = true) & (m3 = 0);

endmodule

module shipment

InitialShipment = bool init false; ReceiveDelivery = bool init false; UpdateInventory = bool init false; ChargeCredit = bool init false; UpdateDelivery = bool init false; EmailConfirmation = bool init false; EndShipment = bool init false; n4 : [0..1] init 1;

endmodule

[ShipmentS] Shipment & (n4 = 1) -> (InitialShipment = true) & (n4 = 0);

...)