

Combining SysML V2 and BIP to Model and Verify CPS Interactions

Adel Khelifati¹^a, Ahmed Hammad²^b and Malika Boukala-Ioualalen¹^c

¹*Faculty of Computer Science, USTHB university, Algiers, Algeria*

²*FEMTO-ST Institute, UMR CNRS 6174, Besançon, France*

{*akhelifati, mioualalen*}@usthb.dz, *ahmed.hammad@femto-st.fr*

Keywords: SysML v2, BIP, Interaction Modeling

Abstract: Cyber-physical systems (CPS) require precise interaction modeling and rigorous verification to guarantee reliability and correctness, particularly in safety-critical systems, where interaction errors such as deadlocks may lead to critical failures. Although SysML v2 provides expressive modeling capabilities, it lacks explicit execution semantics for structured interactions. To address this limitation, we propose a structured subset of SysML v2 to specify interactions at the structural level. These interactions are then mapped to the Behavior, Interaction, Priority (BIP) framework, which defines their execution semantics and enables formal analysis. Specifically, we introduce *Rendez-vous* and *Broadcast* connectors to enforce synchronization and one-to-many communication, respectively, ensuring that interactions are explicitly represented and amenable to formal analysis. BIP provides precise execution semantics, facilitating rigorous verification and streamlining the process by eliminating the need for external verification models. We validate our approach through a case study on swarm drone coordination, demonstrating structured execution, the ability to detect and resolve critical deadlocks, and the correctness and robustness of interactions.

1 Introduction

Cyber-physical systems (CPS) integrate computational and physical processes, requiring precise modeling and rigorous verification to ensure reliability and correctness. These systems consist of multiple interacting components that must coordinate execution through synchronous and asynchronous interactions. Failures in coordination can lead to deadlocks or unexpected behaviors, especially in safety-critical applications such as autonomous vehicles, aerospace, and industrial automation. Ensuring interaction correctness demands formal verification techniques that guarantee safety, liveness, and deadlock freedom (Graja et al., 2020).


The *Systems Modeling Language (SysML)* (Object Management Group (OMG), 2012) provides a structured framework for CPS modeling across different abstraction levels. The recent evolution of SysML into *SysML v2* (Object Management Group (OMG), 2024) introduces significant enhancements in terms of precision, expressiveness, modularity, and the flexibility of its graphical and textual representations, aiming


to address the growing complexity of modern CPS design (Friedenthal, 2023). However, SysML v2 lacks the execution semantics necessary to specify and analyze structured interactions formally. In particular, it does not natively support synchronization and communication mechanisms, which are essential for modeling the coordinated execution of CPS components.


Among the formal languages specialized in interaction verification, the Behavior, Interaction, Priority (BIP) framework (Basu et al., 2006) provides a strong semantic foundation for modeling, composing, and analyzing component interactions. However, BIP is not designed as a system-level modeling language and lacks support for high-level features such as multi-view modeling, requirement traceability, standardized graphical notation, and MBSE tool integration.

Conversely, SysML v2 excels in these areas, offering a rich modeling environment for system structure, requirements, and multi-view architecture, but it does not define execution semantics for interactions.

To bridge this gap, we propose a combined approach that leverages both strengths. Our method uses a dedicated subset of constructs to define the interactions structurally within SysML v2. It transforms the resulting model into BIP, where the execution seman-

^a <https://orcid.org/0009-0006-4522-8123>

^b <https://orcid.org/0000-0003-3739-1650>

^c <https://orcid.org/0000-0002-8713-4997>

tics of these interactions are formally defined. This enables the generation of executable models, facilitating rigorous verification while maintaining compatibility with MBSE practices.

In summary, this paper makes the following contributions:

- We define a structured subset of SysML v2 constructs, including explicit interaction mechanisms, to precisely model synchronous and asynchronous interactions in CPS.
- We introduce a systematic transformation process to map SysML v2 models to the Behavior, Interaction, Priority (BIP) framework, ensuring precise execution semantics and enabling verification without intermediate steps.
- We validate the effectiveness and applicability of our proposed approach through a detailed case study involving swarm drone coordination, demonstrating the detection and resolution of execution issues such as potential deadlocks.

The remainder of this paper is structured as follows. Section 2 reviews related work and highlights the limitations of existing methods. Section 3 presents the necessary background on SysML v2 and BIP. Section 4 describes our structured interaction modeling approach. Section 5 elaborates on the systematic transformation process from SysML v2 to BIP, highlighting its verification capabilities. Section 6 illustrates our approach through a case study on swarm drone coordination. Finally, Section 7 summarizes our contributions and outlines future research directions.

2 Related Work

Ensuring the correctness of the SysML model has been a long-standing challenge. Given that SysML does not provide execution semantics, several studies have investigated integrating SysML with formal verification tools to enable rigorous system analysis. The work of (Molnár et al., 2024) explores the feasibility of linking SysML v2 models with verification techniques. However, while these approaches improve the validation of system properties, they lack support for explicit execution semantics of interactions, particularly mechanisms such as multi-component synchronization and prioritized communication.

One significant effort in integrating SysML with BIP was the work of (Nussbaumer and Kieliger, 2017), which developed a bidirectional transformation between SysML v1 and BIP for visualization and

editing. This method allowed BIP models to be represented within SysML using custom UML profiles and stereotypes. However, this work primarily focused on structural mappings rather than execution semantics, meaning that key interaction mechanisms such as broadcast or rendez-vous were not explicitly modeled within SysML. Additionally, this approach was based on SysML v1, which lacks the enhanced expressiveness and formal semantics introduced in SysML v2.

Although transformation efforts aim to enable formal execution semantics, other research has focused on verifying the structural consistency of interactions in SysML models. The work of (Khelifati et al.,) verifies the static semantic of SysML interactions within Internal Block Diagrams (IBDs). This approach ensures that SysML models conform to well-defined structural constraints, preventing issues such as misconnected ports or inconsistent interfaces. However, static verification alone does not analyze how interactions dynamically evolve during execution. Our approach bridges this gap by mapping SysML v2 models to a formally executable framework like BIP.

Another line of research has explored constraint-based verification to specify and analyze interactions in SysML. The work of (Tannoury et al., 2022a) introduces SysReo, which combines SysML with Reo and OCL to define and verify system interactions using constraint automata. An extension of this approach is presented by (Tannoury et al., 2022b), where SysML/MARTE is used to model real-time constraints, making it particularly suitable for automotive CPS. In contrast to Reo, BIP offers a more comprehensive formal framework that supports compositional modeling and direct execution semantics, eliminating the need for automata-based translations and external formal encodings.

Beyond SysML-based approaches, research has also explored the relationship between BIP and Reo as coordination models for system interactions. The work of (Dokter et al., 2015) establishes a formal mapping from BIP (without priority) to Reo, demonstrating how constraint-based verification techniques can be applied to BIP models. This transformation allows for reasoning about BIP interactions within Reo’s formal framework. However, this approach does not leverage system-level modeling languages such as SysML.

In contrast to these approaches, we propose a method that defines a structured subset of SysML v2 constructs to model structured interactions explicitly. By mapping these models into BIP, execution semantics are introduced, enabling both formal execution and verification. This integration ensures a seamless transition from system-level modeling to executable

analysis without requiring additional transformation steps or external model-checking frameworks.

3 Background

This section introduces the fundamental elements of SysML v2 and BIP relevant to our approach. We focus only on the constructs required to define structured interactions and facilitate their transformation into a formal verification framework.

3.1 SysML v2: Essential Constructs for Interaction Modeling

SysML v2 provides a structured framework for modeling Cyber-Physical Systems, offering improved modularity, richer semantics, and textual representations. It defines key constructs for specifying system components, their interactions, and behaviors, including:

Port Definition (port def) Define interaction points between system components, specifying data flow and communication constraints. Listing 1 defines a SysML v2 port as an interaction point between system components.

Listing 1: Example of a Port Definition in SysML v2

```
port def CommandPort {
  in receivedCommand : String;}
```

Part Definition (part def) Represent a system component, encapsulating ports and internal behavior. Listing 2 represents a reusable component type that exposes interaction ports.

Listing 2: Example of a Part Definition in SysML v2

```
part def Controller {
  port commandIn : CommandPort;}
```

State Definition (state def) Capture the different operational modes of a component. They include a set of states and transitions that describe how the component evolves over time, as illustrated in Listing 3.

Listing 3: Example of a State Definition in SysML v2

```
state def ControllerStates {
  state Idle; state Active;
  transition Idle_to_Active
  first Idle}
```

```
accept receivedCommand via commandIn
then Active;}
```

Connections (connection def) Define interactions between components. Listing 4 presents an example of a connection def in SysML v2

Listing 4: Example of a Connection in SysML v2

```
connection def Synchronization {
  end controller : CommandPort;
  end actuator : CommandPort;
  flow controller.receivedCommand
  to actuator.receivedCommand;}
```

3.2 BIP: A Framework for Interaction and Formal Verification

The *Behavior, Interaction, Priority (BIP)* framework provides a rigorous component-based approach to model and verify CPS. It introduces three main layers: Behavior, Interaction, and Priority, each serving a distinct role in defining system execution semantics.

Behavior (Atomic Components) The Behavior layer encapsulates the state-based dynamics of the components using atomic components, or atoms. Each atom defines internal states and transitions between them. Listing 5 illustrates an example of a BIP atomic component that models a simple Controller with two states, Idle and Active, and a transition triggered by receiving a command.

Listing 5: Example of an Atomic Component in BIP

```
atom type Controller() {
  export port CommandPort_t commandIn;
  place Idle, Active;
  initial to Idle;
  on commandIn from Idle to Active do {
    printf("Controller_activated.\n");}}
```

Interaction (Connectors) Connectors define synchronization and communication between multiple components. In BIP, connectors specify how components interact by enabling or enforcing simultaneous execution of multiple ports. Listing 6 shows a Rendez-vous connector, which ensures that two components (c and a) are executed synchronously

Listing 6: Example of a Rendez-vous Connector in BIP

```
connector type Synchronization(
  CommandPort_t c, CommandPort_t a){
  define c a;}
```

In contrast, a Broadcast connector allows one component to communicate with multiple receivers without requiring them to be available simultaneously. In BIP, an apostrophe (') after a port name in the 'define' statement specifies that the port is a trigger, meaning that its activation initiates the interaction. Listing 7 illustrates an example of a Broadcast connector.

Listing 7: Example of a Broadcast Connector in BIP

```
connector type Broadcast (
  BroadcastPort_t sender,
  CommandPort_t rec1, CommandPort_t
  rec2, CommandPort_t rec3) {
  define sender' rec1 rec2 rec3;
  on sender rec1 rec2 rec3 down {
    rec1.commandSignal=sender.
      commandSignal;
    rec2.commandSignal=sender.
      commandSignal;
    rec3.commandSignal=sender.
      commandSignal;}
  on sender receiver1 down {
    rec1.commandSignal=sender.
      commandSignal;}}
```

In this example, the apostrophe (') after 'sender' in the 'define' statement indicates that 'sender' is the trigger, meaning that its activation initiates the interaction. The 'down' statement ensures that when 'sender' is activated, its command signal is propagated to all receivers, allowing them to receive the message simultaneously.

Priority The Priority layer allows specifying execution order between interactions when multiple are possible, ensuring deterministic system behavior. However, this paper does not focus on priority mechanisms.

The next section presents our approach for defining structured interactions in SysML v2 and mapping them to BIP.

4 Proposed Approach

SysML v2 models the structural architecture of Cyber-Physical Systems (CPS) using components, ports, and their connections. Interactions are specified via connection definitions, with semantics inferred from structure. However, it lacks execution semantics, limiting its ability to analyze dynamic behaviors like synchronization or asynchronous communication.

To address this, we define a subset of SysML v2 constructs—port, part, state, and connection

definitions—for modeling structured interactions. These models are then transformed into BIP, where connectors define formal execution semantics. This enables verification while preserving SysML modeling conventions.

We focus on two interaction types: *Rendez-vous* for strict synchronization, and *Broadcast* for one-to-many communication.

Rendez-vous interactions model strict synchronization between multiple components, requiring all participants to reach a common execution point before proceeding. In our approach, this type of interaction is first specified structurally in SysML v2 using a connection definition that links multiple ports, as shown in Listing 8.

Listing 8: Rendez-vous Interaction in SysML v2

```
connection def RendezVous {
  end p1 : SyncPort;
  end p2 : SyncPort;
  end p3 : SyncPort;}
```

This structural specification is then mapped to a BIP connector that enforces the same synchronization semantics at execution time. The equivalent BIP definition is shown in Listing 9, where all involved ports must participate for the interaction to be executed.

Listing 9: Rendez-vous Connector in BIP

```
connector type RendezVous(SyncPort_t
  p1, SyncPort_t p2, SyncPort_t p3)
{
  define p1 p2 p3; }
```

Broadcast interactions represent asynchronous one-to-many communication, where a sender transmits information to multiple receivers, not all of which are required to participate simultaneously. In our approach, such interactions are first modeled structurally in SysML v2 using a connection definition with optional constraints on participant readiness. A basic example is shown in Listing 10.

Listing 10: Broadcast Interaction in SysML v2

```
connection def Broadcast {
  end sender : BroadcastPort;
  end receiver1 : CommandPort;
  end receiver2 : CommandPort;
  end receiver3 : CommandPort;
  constraint {sender.isReady}
  flow sender.Signal to receiver1.
    Signal;
  flow sender.Signal to receiver2.
    Signal;
  flow sender.Signal to receiver3.
    Signal;}
```

This specification is mapped into a BIP connector

where the sender acts as a trigger, initiating the interaction. The BIP equivalent is presented in Listing 11, where all receivers participate when available.

Listing 11: Broadcast Connector in BIP

```
connector type Broadcast (
  BroadcastPort_t sender,
  CommandPort_t receiver1,
  CommandPort_t receiver2,
  CommandPort_t receiver3) {
  define sender' receiver1 receiver2
    receiver3;
  on sender receiver1 receiver2
    receiver3 down {
    receiver1.Signal = sender.
      Signal;
    receiver2.Signal = sender.
      Signal;
    receiver3.Signal = sender.
      Signal;}}
```

In more complex scenarios, broadcast interactions in SysML v2 may include multiple constraints, each specifying a valid subset of receivers based on readiness conditions. Listing 12 illustrates this case.

Listing 12: Broadcast Interaction with multiple Constraints in SysML v2

```
connection def Broadcast {
  end sender : BroadcastPort;
  end receiver1 : CommandPort;
  end receiver2 : CommandPort;
  constraint {sender.isReady and
    receiver1.isReady and
    receiver2.isReady}
  flow sender.Signal to receiver1.
    Signal;
  flow sender.Signal to receiver2.
    Signal;
  constraint {sender.isReady and
    receiver1.isReady}
  flow sender.Signal to receiver1.
    Signal;
  constraint {sender.isReady and
    receiver2.isReady}
  flow sender.Signal to receiver2.
    Signal;
}
```

These alternative conditions are translated into BIP as conditional interaction patterns using **on** statements, ensuring structured execution based on dynamic availability, as shown in Listing 13.

Listing 13: Broadcast Connector with Multiple Constraints in BIP

```
connector type Broadcast (
  BroadcastPort_t sender,
  CommandPort_t receiver1,
  CommandPort_t receiver2) {
  define sender' receiver1 receiver2;
```

```
  on sender receiver1 receiver2 down {
    receiver1.Signal = sender.Signal;
    receiver2.Signal = sender.Signal;}
  on sender receiver1 down {
    receiver1.Signal = sender.Signal;}
  on sender receiver2 down {
    receiver2.Signal = sender.Signal
    ;}}
```

This mapping from structural constraints in SysML v2 to conditional interactions in BIP preserves the intended communication flexibility while enabling formal execution semantics. The next section elaborates the transformation workflow that supports this mapping.

5 Model Transformation from SysML v2 to BIP

We map structured SysML v2 constructs to their BIP equivalents to enable formal verification. This transformation connects system-level modeling to executable semantics while preserving the structure and behavior of interactions. It supports both synchronous and asynchronous patterns and provides a foundation for rigorous analysis.

The transformation consists of three stages: extracting SysML v2 elements (ports, parts, states, connections), applying rule-based mappings, and generating an executable BIP model. Algorithm 1 summarizes the structural, behavioral, and interaction mappings.

Ports are translated into BIP port types, while components with internal states become BIP atoms composed of places and transitions. If multiple transitions share the same source and target states, they are merged using conditional guards to preserve behavioral distinctions.

Connections are mapped to BIP connectors, with the trigger port identified by an apostrophe ('). Constraints are converted into **on** conditions, and flow statements into **down** blocks to manage data propagation and enforce synchronization semantics.

Though currently manual, the transformation process lays a foundation for automation, thanks to the structured and semantically precise nature of SysML v2. The SysML v2 API enables systematic extraction of model elements (parts, ports, states), allowing transformation rules to be applied programmatically. Integrated with BIP toolchains, this would support the automatic generation of executable BIP models.

Such automation enhances scalability and reduces modeling effort while preserving correctness and traceability. It also strengthens the link between high-

Algorithm 1 Transformation from SysML v2 to BIP

Require: SysML v2 model M_{SysML} **Ensure:** Equivalent BIP model M_{BIP}

- 1: Parse M_{SysML} to extract essential elements:
 - 2: Identify all port def, part def, state def, and connection def
 - 3: Extract transitions from state def
 - 4: Identify constraints and their associated flow statements
 - 5: Initialize an empty BIP model M_{BIP}
 - 6: **for** each port def in M_{SysML} **do**
 - 7: Map to a BIP port type with corresponding attributes
 - 8: Add the mapped BIP port type to M_{BIP}
 - 9: **end for**
 - 10: **for** each part def containing a state def **do**
 - 11: Create a BIP atom type corresponding to the SysML part
 - 12: **for** each state def in the part **do**
 - 13: **for** each state in the state def **do**
 - 14: Create a BIP place representing the state
 - 15: **end for**
 - 16: Convert transitions into BIP transitions
 - 17: **end for**
 - 18: Add the BIP atom type to M_{BIP}
 - 19: **end for**
 - 20: **for** each connection def in M_{SysML} **do**
 - 21: Create a BIP connector type with the corresponding ports
 - 22: Identify the port that appears in all constraints and designate it as the *trigger port* (apostrophe ('))
 - 23: **for** each constraint in the connection **do**
 - 24: Extract the constraint and translate it into a BIP on condition
 - 25: Identify the corresponding flow statements
 - 26: Group the flow statements under the corresponding on condition in the down section
 - 27: **end for**
 - 28: Add each constructed BIP connector type to M_{BIP}
 - 29: **end for**
 - 30: **Transform the System Composition**
 - 31: Identify the top-level part that contains other parts
 - 32: Map it to a BIP compound type
 - 33: Add each BIP compound type to M_{BIP}
 - 34: **for** each contained part instance **do**
 - 35: Add the corresponding BIP atom type to M_{BIP}
 - 36: **end for**
 - 37: **for** each connection def inside the system composition **do**
 - 38: Add the corresponding BIP connector type to M_{BIP}
 - 39: **end for**
 - 40: **return** M_{BIP}
-

level SysML v2 modeling and the formal execution and verification capabilities of the BIP framework.

6 Case Study: Swarm Drone Coordination with SysML v2 and BIP

Swarm drone coordination is an essential challenge in autonomous flight systems, where multiple drones must execute synchronized maneuvers while maintaining strict communication and interaction constraints. This case study focuses on the formal modeling and verification of a swarm drone system using SysML v2 and BIP, ensuring correct coordination and synchronization of drones.

The system consists of a command station that issues flight commands and a swarm of three drones that receive and execute the commands. Two interaction mechanisms govern communication: a broadcast mechanism, where the command station sends commands to all drones simultaneously, and a rendezvous synchronization, where all drones synchronize before resetting to their initial state.

6.1 SysML v2 Model

The swarm drone coordination system is modeled in SysML v2, capturing the structure, behavior, and interactions between the command station and the drones. The model includes port definitions for command transmission and synchronization, part definitions representing system components, state machines defining operational states and transitions, connection definitions establishing interactions, and a system composition assembling all components into the Drone Swarm system. The complete textual specification of this SysML v2 model is presented in the Listing 14.

Listing 14: Complete SysML v2 Model for Swarm Drone Coordination

```
package DroneSwarm {
  private import ScalarValues::*;
  port def CommandPort {
    in attribute commandSignal :
      String ;
    attribute isReady = true ;
  }
  port def BroadcastPort {
    out attribute commandSignal :
      String ;
    attribute isReady = true ;
  }
  port def RendezVousPort { }
  part def Drone {
    port commandIn : CommandPort;
    port rendezVous : RendezVousPort;
    attribute commandSignal : String ;
    state def DroneStates {
      entry; then Idle;
      state Idle; state Ready;
```

```

state Flying; state Formation;
transition 'Idle-Ready'
  first Idle accept signal :
    String via commandIn
  then Ready {commandSignal = "
    Ready";}
transition 'Ready-Flying'
  first Ready
  accept signal : String via
    commandIn
  then Flying {commandSignal = "
    InAir";}
transition 'Flying-Formation'
  first Flying
  accept signal : String via
    commandIn
  then Formation {commandSignal
    = "InFormation";}
transition 'Formation-Idle'
  first Formation
  accept signal : String via
    rendezVous
  then Idle; } }
part def StationDeCommande {
  port stationCommand :
    BroadcastPort;
  state def StationStates {
    attribute phase : Integer = 0;
    attribute compteur : Integer =
      0;
    attribute maxCycles : Integer =
      10;
    state Active; state Stop;
    entry; then Active;
    transition 'Active-Start'
      first Active
      accept Signal : String via
        stationCommand
      if phase == 0 and compteur <
        maxCycles
      then Active {
        phase = 1;
        compteur = compteur + 1;}
    transition 'Active-Takeoff'
      first Active
      accept Signal : String via
        stationCommand
      if phase == 1 and compteur <
        maxCycles
      then Active {
        phase = 2;
        compteur = compteur + 1;}
    transition 'Active-Formation'
      first Active
      accept Signal : String via
        stationCommand
      if phase == 2 and compteur <
        maxCycles
      then Active {
        phase = 0;
        compteur = compteur + 1; }
    transition 'Active-Stop'

```

```

first Active
  if compteur >= maxCycles
  then Stop; }}
connection def Broadcast {
  end sender : BroadcastPort;
  end receiver1 : CommandPort;
  end receiver2 : CommandPort;
  end receiver3 : CommandPort;
  constraint {sender.isReady and
    receiver1.isReady and
    receiver2.isReady and
    receiver3.isReady}
  flow sender.commandSignal to
    receiver1.commandSignal;
  flow sender.commandSignal to
    receiver2.commandSignal;
  flow sender.commandSignal to
    receiver3.commandSignal;
  constraint {sender.isReady and
    receiver1.isReady}
  flow sender.commandSignal to
    receiver1.commandSignal; }
connection def RendezVous {
  end d1 : RendezVousPort;
  end d2 : RendezVousPort;
  end d3 : RendezVousPort; }
part DroneSwarm {
  part station : StationDeCommande;
  part d1 : Drone;
  part d2 : Drone;
  part d3 : Drone;
  connection leaderCommand :
    Broadcast connect (station.
      stationCommand, d1.commandIn,
      d2.commandIn, d3.commandIn);
  connection rendezvousSync :
    RendezVous connect (d1.
      rendezVous, d2.rendezVous, d3.
      rendezVous); } }

```

Figure 1 provides the complete assembly of the swarm drone system components.

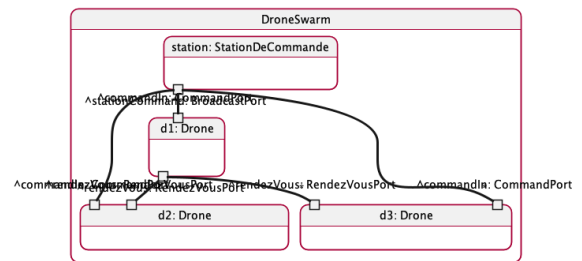


Figure 1: Swarm System Composition in SysML v2

6.2 BIP Model

The BIP model for swarm drone coordination was generated following the algorithm proposed in Section 5. This transformation systematically maps

SysML v2 constructs into their BIP equivalents, ensuring structural and behavioral consistency while preserving interaction semantics. To enhance the analysis of execution processes, additional trace messages were incorporated into the BIP model. These messages allow us to observe the evolution of system states. The complete BIP model representation of the system is shown in Listing 15.

Listing 15: Complete BIP Model for Swarm Drone Coordination

```
@cpp(include="stdio.h")
package Dronell
  extern function printf(string, int)
  port type CommandPort_t(string
    commandSignal)
  port type BroadcastPort_t(string
    commandSignal)
  port type RendezVousPort_t()
  atom type Drone(int id)
    data string commandSignal
    export port CommandPort_t
      commandIn(commandSignal)
    export port RendezVousPort_t
      rendezVous()
  place Idle, Ready, Flying,
    Formation
  initial to Idle do {
    commandSignal = "";
  }
  on commandIn from Idle to Ready do
  {
    commandSignal = "Ready";
    printf("Drone_%d_is_Ready\n", id
    );
  }
  on commandIn from Ready to Flying
  do {
    commandSignal = "InAir";
    printf("Drone_%d_is_Flying\n",
    id);
  }
  on commandIn from Flying to
    Formation do {
    commandSignal = "InFormation";
    printf("Drone_%d_is_in_Formation
    \n", id);
  }
  on rendezVous from Formation to
    Idle do {
    printf("Drone_%d_resets_from_
    Formation_to_Idle\n", id);
  }
end
atom type StationDeCommande()
  data string commandSignal
  data int phase
  data int compteur
  data int maxCycles
  export port BroadcastPort_t
    broadcastCommand(commandSignal
    )
  place Active, Stop
  initial to Active do {
    phase = 0; compteur = 0;
    maxCycles = 4;
```

```
    commandSignal = "";
  }
  on broadcastCommand from Active to
    Active provided (compteur <
    maxCycles) do {
    if (phase == 0) then
      commandSignal = "Start"; phase
      = 1;
    else if (phase == 1) then
      commandSignal = "Takeoff";
      phase = 2;
    else if (phase == 2) then
      commandSignal = "Formation
      ";
      phase = 0;
    fi fi fi
    compteur = compteur + 1;
    printf("Station:_Sending_%s_
    Command\n", commandSignal);
  }
  internal from Active to Stop
    provided (compteur >=
    maxCycles) do {
    printf("Station:_Max_cycles_
    reached._Stopping.\n", 0);
  }
end
connector type Broadcast(
  BroadcastPort_t sender,
  CommandPort_t receiver1,
  CommandPort_t receiver2,
  CommandPort_t receiver3)
  define sender' receiver1 receiver2
    receiver3
  on sender receiver1 receiver2
    receiver3 down { receiver1.
    commandSignal = sender.
    commandSignal;
    receiver2.commandSignal = sender
    .commandSignal;
    receiver3.commandSignal = sender
    .commandSignal;
  }
  on sender receiver1 down {
    receiver1.commandSignal = sender.
    commandSignal;
  }
end
connector type RendezVous(
  RendezVousPort_t d1,
  RendezVousPort_t d2,
  RendezVousPort_t d3)
  define d1 d2 d3
end
compound type DroneSwarm()
  component StationDeCommande station
    ()
  component Drone d1(1)
  component Drone d2(2)
  component Drone d3(3)
  connector Broadcast leaderCommand(
    station.broadcastCommand, d1.
    commandIn, d2.commandIn, d3.
    commandIn)
  connector RendezVous rendezvousSync
    (d1.rendezVous, d2.rendezVous,
```



```

        d3.rendezVous)
    end
end

```

6.3 Execution and Results

The BIP model was executed to validate the correct operation of the swarm drone coordination system. The execution trace, illustrated in Figure 2, demonstrates interactions between the command station and the drones, confirming structured command dissemination and synchronized execution. Initially, the command station successfully broadcasts a Start command, causing all drones to transition from Idle to Ready. Subsequent commands systematically move them through the Flying and Formation states, as defined by the initial SysML v2 specification (Listing 14). When the drones reach the Formation state, they synchronize through the rendez-vous connector, ensuring coordinated execution before resetting to Idle. The broadcast connector facilitates simultaneous dissemination of commands, thus maintaining synchronization and consistent transitions among drones.

Despite these structured transitions, the initial execution results indicated that after completing a predefined number of cycles, the system entered an unintended deadlock state. Specifically, the deadlock emerged in state #5, where the station reached its stopping condition. Due to the absence of additional valid transitions, the system reached a state in which no further interactions were possible, resulting in a deadlock.

To address and illustrate the potential resolution of this deadlock, we modified the SysML model of the command station by removing the stopping condition, thus enabling continuous cyclic operations. The revised StationDeCommande SysML v2 specification is presented in Listing 16. The BIP model generated from this modified SysML v2 specification is correspondingly presented in Listing 17.

Listing 16: Modified Command Station Definition in SysML v2 (without stopping condition)

```

part def StationDeCommande {
    port stationCommand:BroadcastPort;
    state def StationStates {
        attribute phase : Integer = 0;
        state Active; entry; then Active;
        transition 'Active-Start'
            first Active
            accept Signal : String via
                stationCommand
            then Active { phase = 1; }
        transition 'Active-Takeoff'
            first Active

```

```

Station: Sending ♦ Command
Drone 1 is Ready
Drone 2 is Ready
Drone 3 is Ready
[BIP ENGINE]: state #1: 1 interaction:
[BIP ENGINE]: [0] ROOT.leaderCommand: static
commandSignal=Ready;)
[BIP ENGINE]: -> choose [0] ROOT.leaderCommand
mmandIn(commandSignal=Ready;)
Station: Sending (x Command
Drone 1 is Flying
Drone 2 is Flying
Drone 3 is Flying
[BIP ENGINE]: state #2: 1 interaction:
[BIP ENGINE]: [0] ROOT.leaderCommand: static
n(commandSignal=InAir;)
[BIP ENGINE]: -> choose [0] ROOT.leaderCommand
commandIn(commandSignal=InAir;)
Station: Sending (t Command
Drone 1 is in Formation
Drone 2 is in Formation
Drone 3 is in Formation
[BIP ENGINE]: state #3: 2 interactions:
[BIP ENGINE]: [0] ROOT.leaderCommand: static
[BIP ENGINE]: [1] ROOT.rendezvousSync: d1.re
[BIP ENGINE]: -> choose [1] ROOT.rendezvousSyn
Drone 1 resets from Formation to Idle
Drone 2 resets from Formation to Idle
Drone 3 resets from Formation to Idle
[BIP ENGINE]: state #4: 1 interaction:
[BIP ENGINE]: [0] ROOT.leaderCommand: static
;) d3.commandIn(commandSignal=InFormation;)
[BIP ENGINE]: -> choose [0] ROOT.leaderCommand
ormation;) d3.commandIn(commandSignal=InFormat
Station: Sending ♦ Command
Station: Max cycles reached. Stopping.
Drone 1 is Ready
Drone 2 is Ready
Drone 3 is Ready
[BIP ENGINE]: state #5: deadlock!

```

Figure 2: Execution trace of the BIP model showing drone state transitions.

```

accept Signal : String via
    stationCommand
then Active { phase = 2; }
transition 'Active-Formation'
    first Active
    accept Signal : String via
        stationCommand
    then Active {phase = 0;}}}

```

The resulting BIP model, presented in Listing 17, eliminates the stopping condition of the command station and ensures continuous operation, thus removing the conditions that previously led to the deadlock.

Listing 17: Modified BIP Model for Continuous Execution

```

atom type StationDeCommande()
    data string commandSignal
    data int phase
    export port BroadcastPort_t
        broadcastCommand(commandSignal)
    place Active
    initial to Active do { phase = 0;
        commandSignal = ""; }

```

```

on broadcastCommand from Active to
Active do {
  if (phase == 0) then
    commandSignal = "Start";
    phase = 1;
  else if (phase == 1) then
    commandSignal = "Takeoff";
    phase = 2;
  else if (phase == 2) then
    commandSignal = "Formation";
    phase = 0; fi fi fi
  printf("Station: Sending %s Command\n", commandSignal);
end

```

After executing this modified model, the deadlock situation no longer arises. Instead, the system continuously cycles through the predefined states (Idle, Ready, Flying, and Formation) without interruption, demonstrating the correctness and robustness of the structured interactions modeled. This modification highlights the flexibility and effectiveness of our structured approach in diagnosing and resolving interaction-related issues such as deadlocks.

7 Conclusion

This paper presented an approach for modeling structured interactions in Cyber-Physical Systems (CPS) by defining a structured subset of SysML v2. This subset introduces *Rendez-vous* and *Broadcast* connectors to explicitly capture synchronous and asynchronous interactions, enabling their transformation into the Behavior, Interaction, Priority (BIP) framework for formal execution and analysis.

The transformation process systematically maps SysML v2 models into BIP while preserving structural integrity and interaction semantics. A case study on swarm drone coordination validated the approach, demonstrating its ability to model multi-agent interactions and verify synchronization properties through BIP-based execution. The results confirmed that the method supports structured execution, enables the detection and resolution of deadlocks, and ensures interaction correctness.

While effective, the current approach has certain limitations. The transformation does not yet support all BIP constructs, such as hierarchical priorities and composite connectors, and remains a manual process that requires user intervention. Future work will extend the transformation to support these advanced constructs, integrate time-aware interactions, and automate the mapping through model-to-model transformations. Furthermore, incorporating an incremental verification strategy using tools like D-Finder

2 (Bensalem et al., 2011) could enable early detection of inconsistencies. Enhancing automation, scalability, and coverage will further consolidate the integration of SysML v2 and BIP for robust CPS modeling and formal verification.

REFERENCES

- Basu, A., Bozga, M., and Sifakis, J. (2006). Modeling heterogeneous real-time components in bip. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12. Ieee.
- Bensalem, S., Griesmayer, A., Legay, A., Nguyen, T.-H., Sifakis, J., and Yan, R. (2011). D-finder 2: Towards efficient correctness of incremental design. In *Nasa Formal Methods Symposium*, pages 453–458. Springer.
- Dokter, K., Jongmans, S.-S., Arbab, F., and Bludze, S. (2015). Relating bip and reo. *arXiv preprint arXiv:1508.04848*.
- Friedenthal, S. (2023). Future directions for mbse with sysml v2. In *MODELSWARD*, pages 5–9.
- Graja, I., Kallel, S., Guermouche, N., Cheikhrouhou, S., and Hadj Kacem, A. (2020). A comprehensive survey on modeling of cyber-physical systems. *Concurrency and Computation: Practice and Experience*, 32(15):e4850.
- Khelifati, A., Boukala-Ioualalen, M., and Hammad, A. Construction of consistent sysml models applied to the cps. *ACM Journal on Emerging Technologies in Computing Systems*.
- Molnár, V., Graics, B., Vörös, A., Tonetta, S., Cristoforetti, L., Kimberly, G., Dyer, P., Giammarco, K., Koethe, M., Hester, J., et al. (2024). Towards the formal verification of sysml v2 models. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, pages 1086–1095.
- Nussbaumer, C. J. M. and Kieliger, L. (2017). Bidirectional transformation between bip and sysml for visualisation and editing.
- Object Management Group (OMG) (2012). OMG Systems Modeling Language SysML. Technical report.
- Object Management Group (OMG) (2024). Systems Modeling Language (SysML) v2 Beta 2 Specification: Language. <https://www.omg.org/spec/SysML/2.0/Beta2/Language/PDF>. Accessed Mars 2025.
- Tannoury, P., Chouali, S., and Hammad, A. (2022a). An incremental model-based design methodology to develop cps with sysml/ocl/reo. In *Journées du GDR GPL, Jun 2022, Vannes*.
- Tannoury, P., Chouali, S., and Hammad, A. (2022b). Model driven approach to design an automotive cps with sys-reo language. In *Proceedings of the 20th ACM International Symposium on Mobility Management and Wireless Access*, pages 97–104.