A Formal Definition of SysML Block Definition Diagrams Based on a JSON Schema Representation

April 3, 2025

1 Introduction

The Systems Modeling Language (SysML) provides a standardized graphical language for systems engineering applications. The Block Definition Diagram (BDD) is a cornerstone of SysML, used to define system structure, including blocks, interfaces, value types, and their relationships. To enable rigorous analysis and automated processing, a formal, unambiguous definition of the BDD is essential.

This document provides such a formalization, based directly and strictly on a specific JSON schema designed to represent SysML diagrams (the schema corresponding to `BlockDefinitionsDiagram.json` provided separately). We define a BDD as a collection of sets, where each set contains elements of a specific SysML type (e.g., Blocks, Attributes, Associations). Each element is represented as a tuple mirroring its properties defined in the JSON schema. We focus primarily on elements central to the BDD, while acknowledging the provided schema may encompass elements from other diagram types for broader model representation. Finally, we establish fundamental consistency constraints that any valid BDD instance, conforming to this formalization, must satisfy. Standard mathematical notation, including calligraphic letters for sets, is employed for clarity and adherence to academic conventions.

2 Formal Definitions of SysML Elements

This section provides a formal definition of SysML elements that are structured for JSON representation. These definitions establish a rigorous foundation for LLM-generated diagrams and ensure consistency across diagram types. To ensure precise communication between LLMs and modeling tools, we established a rigorous formalization of SysML diagram elements using set theory and schema-based validation. This formalization serves multiple purposes: 1. It provides a canonical JSON representation of SysML Block Definition Diagrams 2. It enables formal verification of LLM outputs against a well-defined schema 3. It creates a foundation for extending our approach to other SysML diagram types 4. It supports interoperability between different modeling tools through a standardized exchange format The following definitions comprehensively describe each element type within our SysML JSON schema, including its intrinsic properties and relationships with other elements.

2.1 Universe of Discourse

Definition 2.1 (Element Universe)**.** *The set E_J contains all diagram elements defined in the JSON instance J. Each element $e \in E_J$ possesses a unique identifier $e.id \in String$ and a type discriminator $e.type \in String$. E_J is partitioned into subsets based on the* `type` *property, e.g., B_{JSON} (Blocks), P_{JSON} (Ports), $Attr_{JSON}$ (Attributes), etc.*

2.2 Classifier Elements

Definition 2.2 (Package). $Pkg_{JSON} = \{pkg \in E_J \mid pkg.type = \text{"Package"} \wedge pkg \text{ conforms to } \mathcal{S}_{Package}\}$
Properties:

- $pkg.id \in String$ *(Unique Identifier)*
- $pkg.type = \text{"Package"}$
- $pkg.name \in String$. *Defines function* $name_{Pkg} : Pkg_{JSON} \to String$.

Relationships:

- $pkg.elementImports \subseteq String$: *Set of IDs. Defines relation* $ImportsElement \subseteq Pkg_{JSON} \times E_J$, *where* $(pkg, e) \in ImportsElement \iff e.id \in pkg.elementImports$.
- $pkg.packageImports \subseteq String$: *Set of IDs. Defines relation* $ImportsPackage \subseteq Pkg_{JSON} \times Pkg_{JSON}$, *where* $(pkg_1, pkg_2) \in ImportsPackage \iff pkg_2.id \in pkg_1.packageImports$.

Required: `id`, `type`, `name`.

Definition 2.3 (Block). $B_{JSON} = \{b \in E_J \mid b.type = \text{"Block"} \wedge b \text{ conforms to } \mathcal{S}_{Block}\}$
Properties:

- $b.id \in String$
- $b.type = \text{"Block"}$
- $b.name \in String$. *Defines function* $name_B : B_{JSON} \to String$.
- $b.isAbstract \in Boolean$
- $b.isActive \in Boolean$
- $b.isEncapsulated \in Boolean$

Relationships (via ID arrays):

- $b.attributeIds \subseteq String$: *Defines* $HasAttribute \subseteq B_{JSON} \times Attr_{JSON}$, *where* $(b, a) \in HasAttribute \iff a.id \in b.attributeIds$.
- $b.operationIds \subseteq String$: *Defines* $HasOperation \subseteq B_{JSON} \times Op_{JSON}$ *(assuming* Op_{JSON} *for Operations).*
- $b.constraintIds \subseteq String$: *Defines* $HasConstraintRef \subseteq B_{JSON} \times Cstr_{JSON}$ *(references Constraint elements).*
- $b.partIds \subseteq String$: *Defines* $HasPart \subseteq B_{JSON} \times Part_{JSON}$.
- $b.referenceIds \subseteq String$: *Defines* $HasReferenceProperty \subseteq B_{JSON} \times Attr_{JSON}$ *(references Attributes acting as reference properties).*
- $b.valueIds \subseteq String$: *Defines* $HasValueProperty \subseteq B_{JSON} \times Attr_{JSON}$ *(references Attributes acting as value properties).*

Required: `id`, `type`, `name`.

Definition 2.4 (InterfaceBlock). $IB_{JSON} = \{ib \in E_J \mid ib.type = \text{"InterfaceBlock"} \wedge ib \text{ conforms to } \mathcal{S}_{InterfaceBloc}$
Properties: $ib.id$, $ib.type = \text{"InterfaceBlock"}$, $ib.name \in String$ $(name_{IB})$.
Relationships: `attributeIds`, `operationIds`, `constraintIds` *(as for Block).* `signalIds` $\subseteq String$: *Defines* $HasSignal \subseteq IB_{JSON} \times Sig_{JSON}$.
Required: `id`, `type`, `name`.

Definition 2.5 (FlowSpecification). $FS_{JSON} = \{fs \in E_J \mid fs.type = \text{"FlowSpecification"} \wedge fs \text{ conforms to } \mathcal{S}_{FlowSpecification}\}$

Properties: $fs.id$, $fs.type = \text{"FlowSpecification"}$, $fs.name \in String$ ($name_{FS}$), $fs.flowType \in String$, $fs.dataType \in String$, $fs.unit \in String$.

Relationships: `attributeIds` (defines $HasAttribute \subseteq FS_{JSON} \times Attr_{JSON}$).

Required: `id`, `type`, `name`.

Definition 2.6 (ConstraintBlock). $CB_{JSON} = \{cb \in E_J \mid cb.type = \text{"ConstraintBlock"} \wedge cb \text{ conforms to } \mathcal{S}_{ConstraintBlock}\}$

Properties: $cb.id$, $cb.type = \text{"ConstraintBlock"}$, $cb.name \in String$ ($name_{CB}$), $cb.expression \in String$.

Relationships: `parameterIds` $\subseteq String$: Defines $HasParameter \subseteq CB_{JSON} \times Param_{JSON}$. `attributeIds`.

Required: `id`, `type`, `name`, `expression`.

Definition 2.7 (Domain). $Dom_{JSON} = \{dom \in E_J \mid dom.type = \text{"Domain"} \wedge dom \text{ conforms to } \mathcal{S}_{Domain}\}$

Properties: $dom.id$, $dom.type = \text{"Domain"}$, $dom.name \in String$ ($name_{Dom}$).

Relationships: `attributeIds`.

Required: `id`, `type`, `name`.

Definition 2.8 (ValueType). $VT_{JSON} = \{vt \in E_J \mid vt.type = \text{"ValueType"} \wedge vt \text{ conforms to } \mathcal{S}_{ValueType}\}$

Properties: $vt.id$, $vt.type = \text{"ValueType"}$, $vt.name \in String$ ($name_{VT}$).

Relationships: `quantityKindId` $\in String$: Defines $HasQuantityKind \subseteq VT_{JSON} \times QK_{JSON}$. `unitId` $\in String$: Defines $HasUnit \subseteq VT_{JSON} \times Unit_{JSON}$. `attributeIds`.

Required: `id`, `type`, `name`.

Definition 2.9 (QuantityKind). $QK_{JSON} = \{qk \in E_J \mid qk.type = \text{"QuantityKind"} \wedge qk \text{ conforms to } \mathcal{S}_{QuantityK}$

Properties: $qk.id$, $qk.type = \text{"QuantityKind"}$, $qk.name \in String$ ($name_{QK}$), $qk.symbol \in String$, $qk.description \in String$, $qk.definitionURI \in String$.

Relationships: `attributeIds`.

Required: `id`, `type`, `name`.

Definition 2.10 (Unit). $Unit_{JSON} = \{u \in E_J \mid u.type = \text{"Unit"} \wedge u \text{ conforms to } \mathcal{S}_{Unit}\}$

Properties: $u.id$, $u.type = \text{"Unit"}$, $u.name \in String$ ($name_{Unit}$), $u.symbol \in String$, $u.description \in String$, $u.definitionURI \in String$.

Relationships: `quantityKindIds` $\subseteq String$: Defines $AssociatedWithQK \subseteq Unit_{JSON} \times QK_{JSON}$. `attributeIds`.

Required: `id`, `type`, `name`.

Definition 2.11 (Enumeration). $Enum_{JSON} = \{en \in E_J \mid en.type = \text{"Enumeration"} \wedge en \text{ conforms to } \mathcal{S}_{Enumeration}\}$

Properties: $en.id$, $en.type = \text{"Enumeration"}$, $en.name \in String$ ($name_{Enum}$). $en.literals$: An array of objects, $Lit_{en} = \{l \mid l \in en.literals\}$. Each $l \in Lit_{en}$ has $l.id \in String$ and $l.name \in String$.

Relationships: `attributeIds`.

Required: `id`, `type`, `name`, `literals`.

Definition 2.12 (Signal). $Sig_{JSON} = \{sig \in E_J \mid sig.type = \text{"Signal"} \wedge sig \text{ conforms to } \mathcal{S}_{Signal}\}$

Properties: $sig.id$, $sig.type = \text{"Signal"}$, $sig.name \in String$ ($name_{Sig}$).

Relationships: `parameterIds` $\subseteq String$: Defines $HasParameter \subseteq Sig_{JSON} \times Param_{JSON}$. `attributeIds`.

Required: `id`, `type`, `name`.

Definition 2.13 (Interface). $Iface_{JSON} = \{iface \in E_J \mid iface.type = \text{"Interface"} \wedge iface \text{ conforms to } \mathcal{S}_{Interfac}$

Properties: $iface.id$, $iface.type = \text{"Interface"}$, $iface.name \in String$ ($name_{Iface}$).

Relationships: `operationIds`, `signalIds`, `attributeIds`.

Required: `id`, `type`, `name`.

2.3 Instance and Property Elements

Definition 2.14 (Instance). $Inst_{JSON} = \{inst \in E_J \mid inst.type = \text{"Instance"} \wedge inst \text{ conforms to } \mathcal{S}_{Instance}\}$

Properties: inst.id, inst.type = "Instance", inst.name ∈ String ($name_{Inst}$).

Relationships: `classifierId` *∈ String: Defines* $IsInstanceOf \subseteq Inst_{JSON} \times (B_{JSON} \cup IB_{JSON} \cup \dots)$. `slotIds` *⊆ String: Defines* $HasSlot \subseteq Inst_{JSON} \times Slot_{JSON}$.

Required: `id`, `type`, `name`, `classifierId`.

Definition 2.15 (Port). $P_{JSON} = \{p \in E_J \mid p.type \in \{\text{"Port"}, \text{"ProxyPort"}, \text{"FullPort"}, \text{"FlowPort"}\} \wedge p \text{ conforms to } \mathcal{S}_{p.type}\}$

Common Properties: p.id, p.type ∈ {"Port", "ProxyPort", "FullPort", "FlowPort"}, p.name ∈ String ($name_P$), p.isConjugated ∈ Boolean.

Common Relationships: `blockId` *∈ String: Defines* $OwnedByBlock \subseteq P_{JSON} \times B_{JSON}$. `attributeIds`.

Proxy/Full Specific Relationships: `providedInterfaceIds` *⊆ String: Defines* $ProvidesInterface \subseteq P_{JSON} \times (Iface_{JSON} \cup IB_{JSON})$. `requiredInterfaceIds` *⊆ String: Defines* $RequiresInterface \subseteq P_{JSON} \times (Iface_{JSON} \cup IB_{JSON})$.

Flow Specific Relationship: `flowSpecificationId` *∈ String: Defines* $TypedByFlowSpec \subseteq P_{JSON} \times FS_{JSON}$.

Required: `id`, `type`, `name`, `blockId`. *Additionally,* `flowSpecificationId` *for FlowPort.*

Definition 2.16 (Part). $Part_{JSON} = \{pt \in E_J \mid pt.type = \text{"Part"} \wedge pt \text{ conforms to } \mathcal{S}_{Part}\}$

Properties: pt.id, pt.type = "Part", pt.name ∈ String ($name_{Part}$).

Relationship: `of` *∈ String: Defines* $TypedByBlock \subseteq Part_{JSON} \times B_{JSON}$. *(Interpreted as the type of the part).*

Required: `id`, `type`, `name`, `of`.

Definition 2.17 (Attribute/Property). $Attr_{JSON} = \{a \in E_J \mid a.type = \text{"Property"} \wedge a \text{ conforms to } \mathcal{S}_{Attribute}\}$

Properties: a.id, a.type = "Property", a.name ∈ String ($name_{Attr}$), a.propertyType ∈ {"String", "Integer", ...}, a.value ∈ String, a.visibility ∈ {"public", ...}, a.aggregation ∈ {"none", ...}, a.lowerValue ∈ String, a.upperValue ∈ String, a.defaultValue ∈ String, a.isReadOnly ∈ Boolean, a.isStatic ∈ Boolean, a.isDerived ∈ Boolean, a.isDerivedUnion ∈ Boolean, a.isOrdered ∈ Boolean, a.isUnique ∈ Boolean, a.isID ∈ Boolean.

Required: `id`, `type`, `name`, `propertyType`.

Definition 2.18 (Parameter). $Param_{JSON} = \{param \in E_J \mid param.type = \text{"Parameter"} \wedge param \text{ conforms to } \mathcal{S}_{Parameter}\}$

Properties: param.id, param.type = "Parameter", param.name ∈ String ($name_{Param}$), param.parameterType ∈ String, param.defaultValue ∈ String.

Required: `id`, `type`, `name`, `parameterType`.

2.4 Relationship Elements

Definition 2.19 (InterfaceRealization). $IR_{JSON} = \{ir \in E_J \mid ir.type = \text{"InterfaceRealization"} \wedge ir \text{ conforms to } \mathcal{S}_{InterfaceRealization}\}$

Properties: ir.id, ir.type = "InterfaceRealization", ir.name ∈ String ($name_{IR}$).

Relationship Defined: $RealizesInterface \subseteq (B_{JSON} \cup IB_{JSON}) \times (Iface_{JSON} \cup IB_{JSON})$, *where* $(c_1, c_2) \in RealizesInterface \iff \exists ir \in IR_{JSON}(ir.sourceId = c_1.id \wedge ir.targetId = c_2.id)$.

Required: `id`, `type`, `name`, `sourceId`, `targetId`.

Definition 2.20 (Link). $Link_{JSON} = \{lnk \in E_J \mid lnk.type = \text{"Link"} \wedge lnk \text{ conforms to } \mathcal{S}_{Link}\}$

Properties: lnk.id, lnk.type = "Link", lnk.name ∈ String ($name_{Link}$).

Relationship Defined: $ConnectsInstances \subseteq Link_{JSON} \times Inst_{JSON} \times Inst_{JSON}$, *where* $(lnk, i_1, i_2) \in ConnectsInstances \iff (lnk.instance1Id = i_1.id \wedge lnk.instance2Id = i_2.id)$.

Relationship: `associationId` $\in$ *String: Defines* $InstantiatesAssociation \subseteq Link_{JSON} \times Assoc_{JSON}$.

Required: `id`, `type`, `name`, `instance1Id`, `instance2Id`.

Definition 2.21 (AssociationBlock). $AB_{JSON} = \{ab \in E_J \mid ab.type \in \{$*"AssociationBlock"*, *"AssociationBlock* *ab conforms to* $\mathcal{S}_{ab.type}\}$

Properties: ab.id, ab.type $\in \{$*"AssociationBlock"*, *"AssociationBlockWithOwnedEnds"*$\}$, *ab.name* $\in$ *String* ($name_{AB}$).

Relationship Defined: $ConnectsEnds \subseteq AB_{JSON} \times B_{JSON} \times B_{JSON}$, *where* $(ab, b_1, b_2) \in ConnectsEnds \iff (ab.end1Id = b_1.id \wedge ab.end2Id = b_2.id)$.

Relationships: `attributeIds`, `operationIds`, `constraintIds`.

Required: `id`, `type`, `name`, `end1Id`, `end2Id`.

Definition 2.22 (Association). $Assoc_{JSON} = \{r \in E_J \mid r.type \in \{$*"Association"*, *"DirectedAssociation"*, *"Aggr* r *conforms to* $\mathcal{S}_{r.type}\}$

Properties: r.id, r.type $\in \{$*"Association"*, *"DirectedAssociation"*, *"Aggregation"*, *"DirectedAggregation"*, *"Co* $r.name \in String$ ($name_{Assoc}$), $r.sourceMultiplicity \in String$, $r.targetMultiplicity \in String$.

Relationship Defined: $Associates \subseteq Assoc_{JSON} \times Cls_{JSON} \times Cls_{JSON}$ *(where* Cls_{JSON} *is the union of all classifier sets), where* $(r, c_1, c_2) \in Associates \iff (r.sourceId = c_1.id \wedge r.targetId = c_2.id)$. *Subtypes add semantics (directionality, aggregation kind).*

Relationships: `memberEndIds` $\subseteq$ *String: Defines* $HasMemberEnd \subseteq Assoc_{JSON} \times Attr_{JSON}$. `ownedEndIds` $\subseteq$ *String: Defines* $HasOwnedEnd \subseteq Assoc_{JSON} \times Attr_{JSON}$.

Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.23 (Generalization). $Gen_{JSON} = \{g \in E_J \mid g.type =$ *"Generalization"*$\wedge g$ *conforms to* $\mathcal{S}_{Generalizat}$

Properties: g.id, g.type = *"Generalization"*, $g.name \in String$ ($name_{Gen}$), $g.isSubstitutable \in$ *Boolean.*

Relationship Defined: $IsSubtypeOf \subseteq Cls_{JSON} \times Cls_{JSON}$, *where* $(c_1, c_2) \in IsSubtypeOf \iff \exists g \in Gen_{JSON}(g.sourceId = c_1.id \wedge g.targetId = c_2.id)$.

Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.24 (Usage). $Usage_{JSON} = \{u \in E_J \mid u.type =$ *"Usage"* $\wedge\ u$ *conforms to* $\mathcal{S}_{Usage}\}$

Properties: u.id, u.type = *"Usage"*, $u.name \in String$ ($name_{Usage}$).

Relationship Defined: $Uses \subseteq Cls_{JSON} \times Cls_{JSON}$, *where* $(c_1, c_2) \in Uses \iff \exists u \in Usage_{JSON}(u.sourceId = c_1.id \wedge u.targetId = c_2.id)$.

Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.25 (ItemFlow). $Flow_{JSON} = \{f \in E_J \mid f.type =$ *"ItemFlow"*$\wedge f$ *conforms to* $\mathcal{S}_{ItemFlow}\}$

Properties: f.id, f.type = *"ItemFlow"*, $f.name \in String$ ($name_{Flow}$).

Relationship Defined: $FlowsBetween \subseteq Flow_{JSON} \times E_J \times E_J$, *where* $(f, e_1, e_2) \in FlowsBetween \iff (f.sourceId = e_1.id \wedge f.targetId = e_2.id)$.

Relationships: `flowSpecificationId` $\in$ *String: Defines* $SpecifiesFlowItem \subseteq Flow_{JSON} \times FS_{JSON}$. `itemPropertyIds` $\subseteq$ *String.*

Required: `id`, `type`, `sourceId`, `targetId`, `flowSpecificationId`.

Definition 2.26 (Connector). $Conn_{JSON} = \{cn \in E_J \mid cn.type =$ *"Connector"*$\wedge cn$ *conforms to* $\mathcal{S}_{Connector}\}$

Properties: cn.id, cn.type = *"Connector"*, $cn.name \in String$ ($name_{Conn}$), $cn.kind \in \{$*"assembly"*, *"delegatio*

Relationship Defined: $ConnectsInternally \subseteq Conn_{JSON} \times (P_{JSON} \cup Part_{JSON}) \times (P_{JSON} \cup Part_{JSON})$, *where* $(cn, e_1, e_2) \in ConnectsInternally \iff (cn.source = e_1.id \wedge cn.target = e_2.id)$.

Required: `id`, `type`, `source`, `target`.

Definition 2.27 (Dependency). $Dep_{JSON} = \{dep \in E_J \mid dep.type = \text{"Dependency"} \wedge dep \text{ conforms to } \mathcal{S}_{Dependen}$
Properties: dep.id, dep.type = "Dependency", dep.name ∈ String ($name_{Dep}$), dep.sourceMultiplicity ∈ String, dep.targetMultiplicity ∈ String.
Relationship Defined: DependsOn ⊆ $E_J \times E_J$, where $(e_1, e_2) \in DependsOn \iff \exists dep \in Dep_{JSON}(dep.sourceId = e_1.id \wedge dep.targetId = e_2.id)$.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.28 (Realization). $Real_{JSON} = \{real \in E_J \mid real.type = \text{"Realization"} \wedge real \text{ conforms to } \mathcal{S}_{Realizat}$
Properties: real.id, real.type = "Realization", real.name ∈ String ($name_{Real}$), real.sourceMultiplicity ∈ String, real.targetMultiplicity ∈ String.
Relationship Defined: Realizes ⊆ $E_J \times E_J$, where $(e_1, e_2) \in Realizes \iff \exists real \in Real_{JSON}(real.sourceId = e_1.id \wedge real.targetId = e_2.id)$.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.29 (Abstraction). $Abs_{JSON} = \{abs \in E_J \mid abs.type = \text{"Abstraction"} \wedge abs \text{ conforms to } \mathcal{S}_{Abstraction}$
Properties: abs.id, abs.type = "Abstraction", abs.name ∈ String ($name_{Abs}$), abs.sourceMultiplicity ∈ String, abs.targetMultiplicity ∈ String.
Relationship Defined: Abstracts ⊆ $E_J \times E_J$, where $(e_1, e_2) \in Abstracts \iff \exists abs \in Abs_{JSON}(abs.sourceId = e_1.id \wedge abs.targetId = e_2.id)$.
Required: `id`, `type`, `sourceId`, `targetId`.

2.5 Other Diagram Elements

Definition 2.30 (Comment). $Cmt_{JSON} = \{cmt \in E_J \mid cmt.type = \text{"Comment"} \wedge cmt \text{ conforms to } \mathcal{S}_{Comment}\}$
Properties: cmt.id, cmt.type = "Comment", cmt.body ∈ String.
Relationship: `annotatedElementId` *∈ String: Defines AnnotatesElement ⊆ $Cmt_{JSON} \times E_J$.*
Required: `id`, `type`, `body`, `annotatedElementId`.

Definition 2.31 (InternalBlock). $IntBlk_{JSON} = \{iblk \in E_J \mid iblk.type = \text{"InternalBlock"} \wedge iblk \text{ conforms to } \mathcal{S}_{InternalBlock}\}$
Properties: iblk.id, iblk.type = "InternalBlock", iblk.name ∈ String ($name_{IntBlk}$).
Relationships: `attributeIds`, `partIds`.
Relationship: `of` *∈ String: Defines RepresentsBlockType ⊆ $IntBlk_{JSON} \times B_{JSON}$.*
Required: `id`, `type`, `name`, `of`.

Definition 2.32 (Requirement). $Req_{JSON} = \{req \in E_J \mid req.type = \text{"Requirement"} \wedge req \text{ conforms to } \mathcal{S}_{Requirem}$
Properties: req.id, req.type = "Requirement", req.name ∈ String ($name_{Req}$), req.text ∈ String, req.reqId ∈ String.
Relationship: `attributeIds`.
Required: `id`, `type`, `name`, `text`.

Definition 2.33 (Actor). $Act_{JSON} = \{act \in E_J \mid act.type = \text{"Actor"} \wedge act \text{ conforms to } \mathcal{S}_{Actor}\}$
Properties: act.id, act.type = "Actor", act.name ∈ String ($name_{Act}$).
Relationship: `attributeIds`.
Required: `id`, `type`, `name`.

Definition 2.34 (UseCase). $UC_{JSON} = \{uc \in E_J \mid uc.type = \text{"UseCase"} \wedge uc \text{ conforms to } \mathcal{S}_{UseCase}\}$
Properties: uc.id, uc.type = "UseCase", uc.name ∈ String ($name_{UC}$).
Relationship: `attributeIds`.
Required: `id`, `type`, `name`.

Definition 2.35 (Activity). $Acty_{JSON} = \{acty \in E_J \mid acty.type = \text{"Activity"} \wedge acty \text{ conforms to } \mathcal{S}_{Activity}\}$
Properties: acty.id, acty.type = "Activity", acty.name ∈ String ($name_{Acty}$).
Relationship: `attributeIds`.
Required: `id`, `type`, `name`.

Definition 2.36 (State). $State_{JSON} = \{st \in E_J \mid st.type = \text{"State"} \wedge st \text{ conforms to } \mathcal{S}_{State}\}$
Properties: st.id, st.type = "State", st.name ∈ String ($name_{State}$).
Relationship: `attributeIds`.
Required: `id`, `type`, `name`.

Definition 2.37 (Lifeline). $LL_{JSON} = \{ll \in E_J \mid ll.type = \text{"Lifeline"} \wedge ll \text{ conforms to } \mathcal{S}_{Lifeline}\}$
Properties: ll.id, ll.type = "Lifeline", ll.name ∈ String ($name_{LL}$).
Relationship: `represents` *∈ String: Defines* $RepresentsInstance \subseteq LL_{JSON} \times E_J$.
Required: `id`, `type`, `name`.

Definition 2.38 (Message). $Msg_{JSON} = \{msg \in E_J \mid msg.type = \text{"Message"} \wedge msg \text{ conforms to } \mathcal{S}_{Message}\}$
Properties: msg.id, msg.type = "Message", msg.name ∈ String ($name_{Msg}$), msg.messageType ∈ String, msg.signature ∈ String.
Relationships: `source`, `target` *define* $SentBetweenLifelines \subseteq Msg_{JSON} \times LL_{JSON} \times LL_{JSON}$.
Required: `id`, `type`, `name`, `source`, `target`.

Definition 2.39 (Include). $Incl_{JSON} = \{incl \in E_J \mid incl.type = \text{"Include"} \wedge incl \text{ conforms to } \mathcal{S}_{Include}\}$
Properties: incl.id, incl.type = "Include", incl.name ∈ String ($name_{Incl}$).
Relationship defined: $IncludesUseCase \subseteq UC_{JSON} \times UC_{JSON}$ *via* `sourceId`, `targetId`.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.40 (Extend). $Ext_{JSON} = \{ext \in E_J \mid ext.type = \text{"Extend"} \wedge ext \text{ conforms to } \mathcal{S}_{Extend}\}$
Properties: ext.id, ext.type = "Extend", ext.name ∈ String ($name_{Ext}$).
Relationship defined: $ExtendsUseCase \subseteq UC_{JSON} \times UC_{JSON}$ *via* `sourceId`, `targetId`.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.41 (ControlFlow). $CF_{JSON} = \{cf \in E_J \mid cf.type = \text{"ControlFlow"} \wedge cf \text{ conforms to } \mathcal{S}_{ControlFlow}\}$
Properties: cf.id, cf.type = "ControlFlow", cf.name ∈ String ($name_{CF}$), cf.guard ∈ String.
Relationship defined: $ControlFlowsTo \subseteq ActNode_{JSON} \times ActNode_{JSON}$ *via* `sourceId`, `targetId`.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.42 (ObjectFlow). $OF_{JSON} = \{of \in E_J \mid of.type = \text{"ObjectFlow"} \wedge of \text{ conforms to } \mathcal{S}_{ObjectFlow}\}$
Properties: of.id, of.type = "ObjectFlow", of.name ∈ String ($name_{OF}$), of.guard ∈ String.
Relationship defined: $ObjectFlowsTo \subseteq ActNode_{JSON} \times ActNode_{JSON}$ *via* `sourceId`, `targetId`.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.43 (Transition). $Trans_{JSON} = \{trans \in E_J \mid trans.type = \text{"Transition"} \wedge trans \text{ conforms to } \mathcal{S}_{Transition}\}$
Properties: trans.id, trans.type = "Transition", trans.name ∈ String ($name_{Trans}$), trans.trigger ∈ String, trans.guard ∈ String, trans.effect ∈ String.
Relationship defined: $TransitionsTo \subseteq State_{JSON} \times State_{JSON}$ *via* `sourceId`, `targetId`.
Required: `id`, `type`, `sourceId`, `targetId`.

Definition 2.44 (Constraint). $Cstr_{JSON} = \{cstr \in E_J \mid cstr.type = \text{"Constraint"} \wedge cstr \text{ conforms to } \mathcal{S}_{Constrain}$
Properties: cstr.id, cstr.type = "Constraint", cstr.name ∈ String ($name_{Cstr}$), cstr.specification ∈ String.
Relationship: `constrainedElementIds` $\subseteq$ *String: Defines* $ConstrainsElement \subseteq Cstr_{JSON} \times E_J$.
Required: `id`, `type`, `name`, `specification`.

3 Conclusion

This document has presented a formal definition of SysML Block Definition Diagrams, meticulously derived from a specific JSON schema representation. By employing a set-theoretic approach and defining elements as tuples mirroring the schema's properties, we establish a precise and unambiguous foundation. The focus remains on core BDD elements, ensuring relevance and clarity. The use of standard calligraphic notation ($\mathcal{P}, \mathcal{B}, \dots$) for sets aligns with common academic practice. The outlined consistency constraints (Identifier Uniqueness, Referential Integrity, Type Consistency, Ownership, Port Context, Association Ends, Generalization, Attribute Typing) are fundamental for ensuring the well-formedness of BDD models conforming to this definition. This formalization serves as a critical step towards enabling reliable automated validation, analysis, and manipulation of SysML BDD models in rigorous engineering workflows. Further work could involve defining more complex semantic constraints (e.g., multiplicity validation, constraint language semantics) and operational semantics based on this structural foundation.