Discrete Dynamics Ontology

- A State is an Initial State if it begins a parent loop or a parent sequence. With this definition, a simple model can have multiple initial states.

  SWRL:
  State(?s) ^ initializesSequence(?s,?seq) ^ isParentSetElement(?seq,"true"^^xsd:boolean) - > InitialState(?s)
  Equivalent OWL Expressions:
  State and (initializesSequence some Sequence and (isParentSetElement value true)) SubClassOf: InitialState
- Unlike Initial State, a State is a Final State if it ends a parent loop or a parent sequence. Again, with this definition, a simple model can have multiple final states.

  SWRL:
  State(?s) ^ finalizesSequence(?s,?seq) ^ isParentSetElement(?seq,"true"^^xsd:boolean) - > FinalState(?s)
  Equivalent OWL Expressions:
  State and (finalizesSequence some Sequence and (isParentSetElement value true)) SubClassOf: FinalState
- A state is a Current State if it is active. It means, if a State has the data property Is *Active = True*, then, it is a Current State. In concurrent environments, there can be multiple current states.

  SWRL:
  State(?s) ^ isActive(?s,"true"^^xsd:boolean) - > CurrentState(?s)
  Equivalent OWL Expressions:
  State and (isActive value true) SubClassOf: CurrentState

- A State is a Next State if it is part of the range of the Has Next State property of the Current State. In concurrent environments, there can be multiple next states.

  SWRL:
  CurrentState(?s1) ^ State(?s2) ^ hasNextState(?s1, ?s2) -> NextState(?s2)
  Equivalent OWL Expressions:
  State and (hasPreviousState some CurrentState) SubClassOf: NextState

- A State is a Previous State if it is part of the range of the Has Previous State property of the Current State. In concurrent environments, there can be multiple previous states.

  SWRL:
  CurrentState(?s1) ^ State(?s2) ^ hasPreviousState(?s1, ?s2) -> PreviousState(?s2)
  Equivalent OWL Expressions:
  State and (hasNextState some CurrentState) SubClassOf: PreviousState

- A State is a Concurrent State if they are at least two current states activated. All the active states (at the same time) become members of the Concurrent State class.

  SWRL:
  CurrentState(?s1) ^ CurrentState(?s2) ^ differentFrom(?s1, ?s2) -> ConcurrentState(?s1) ^ ConcurrentState(?s2)

- A State is a Resource-Shared State if more than one transition can be activated depending on its resource and each transition is connected to other additional state. The Resource-Shared State can fire only one transition at the same time.

  SWRL:
  State(?s1) ^ State(?s2) ^ State(?s3) ^ Transition(?t1) ^ Transition(?t2) ^ differentFrom(?s1,?s2) ^ differentFrom(?s1,?s3) ^ differentFrom(?s2,?s3) ^ differentFrom(?t1,?t2) ^ isConnectedToTransition(?s1, ?t1) ^ isConnectedToTransition(?s1, ?t2) ^ isConnectedToTransition(?s2, ?t1) ^ isConnectedToTransition(?s3, ?t2) -> Resource-SharedState(?s1)

- A State is a Synchronous State if more than one state is connected to the same transition. All the states that comply this characteristic become members of the Synchronous State class.

SWRL:
State(?s1) ^ State(?s2) ^ differentFrom(?s1,?s2) ^ Transition(?t) ^ isConnectedToTransition(?s1,?t) ^ isConnectedToTransition(?s2,?t) -> SynchronousState(?s2) ^ SynchronousState(?s1)

- A Transition is a One-to-Many Transition if it is followed by more than one state and preceded by exactly one state. There can be multiple One-to-Many Transitions.

  SWRL:
  Transition(?t) ^ State(?s1) ^ State(?s2) ^ State(?s3) ^ differentFrom(?s1,?s2) ^ differentFrom(?s2,?s3) ^ hasNextState(?s1,?s2) ^ hasNextState(?s1,?s3) ^ isConnectedToTransition(?s1,?t) ^ isConnectedToState(?t,?s2) ^ isConnectedToState(?t,?s3) -> OneToManyTransition(?t)

- A Transition is a Many-to-Many Transition if it is followed and preceded by more than one state. There can be multiple Many-to-Many Transitions.

  SWRL:
  Transition(?t) ^ State(?s1) ^ State(?s2) ^ State(?s3) ^ State(?s4) ^ differentFrom(?s1,?s2) ^ differentFrom(?s3,?s4) ^ differentFrom(?s2,?s3) ^ hasNextState(?s1,?s3) ^ hasNextState(?s1,?s4) ^ hasNextState(?s2,?s3) ^ hasNextState(?s2,?s4) ^ isConnectedToTransition(?s1,?t) ^ isConnectedToTransition(?s2,?t) ^ isConnectedToState(?t,?s3) ^ isConnectedToState(?t,?s4) -> ManyToManyTransition(?t)

- A Transition is a One-to-One Transition if it is followed and preceded by exactly one state. There can be multiple One-to-One Transitions.

  SWRL:
  Transition(?t) ^ State(?s1) ^ State(?s2) ^ differentFrom(?s1,?s2) ^ hasNextState(?s1,?s2) ^ isConnectedToTransition(?s1,?t) ^ isConnectedToState(?t,?s2) -> OneToOneTransition(?t)

- A Transition is a Many-to-One Transition if it is followed by exactly one state and preceded by more than one state. There can be multiple Many-to-One Transitions.

  SWRL:
  Transition(?t) ^ State(?s1) ^ State(?s2) ^ State(?s3) ^ differentFrom(?s1,?s2) ^ differentFrom(?s2,?s3) ^ hasNextState(?s1,?s3) ^ hasNextState(?s2,?s3) ^ isConnectedToTransition(?s1,?t) ^ isConnectedToTransition(?s2,?t) ^ isConnectedToState(?t,?s3) -> ManyToOneTransition(?t)

- If a variable Acts Over an object, then, it is an Output.

  SWRL:
  Variable(?v) ^ Object(?e) ^ actsOver(?v,?e) -> Output(?v)
  Equivalent OWL Expressions:
  Variable and (actsOver some Object) SubClassOf: Output
- If a variable Reads About an object, then, it is an Input.

  SWRL:
  Variable(?v) ^ Object(?e) ^ monitors(?v,?e) -> Input(?v)
  Equivalent OWL Expressions:
  Variable and (monitors some Object) SubClassOf: Input

- If a variable has the data type equal to Boolean, then, it is a Digital Variable.

  SWRL:
  Variable(?v) ^ hasDataType(?v,"boolean"^^xsd:string) -> DigitalVariable(?v)
  Equivalent OWL Expressions:
  Variable and (hasDataType value "boolean") SubClassOf: DigitalVariable

- If a variable has the data type equal to number (Integer, Long, Float, or Double), then, it is an Analog Variable. Of course, it fits to the other numeric data types and can be implemented, but just 4 datatypes will be used for this exercise.

SWRL:

- *Variable(?v) ^ hasDataType(?v,"integer"^^xsd:string) -> AnalogVariable(?v)*
- *Variable(?v) ^ hasDataType(?v,"long"^^xsd:string) -> AnalogVariable(?v)*
- *Variable(?v) ^ hasDataType(?v,"float"^^xsd:string) -> AnalogVariable(?v)*
- *Variable(?v) ^ hasDataType(?v,"double"^^xsd:string) -> AnalogVariable(?v)*

Equivalent OWL Expressions:
Variable and ((hasDataType value "integer") or (hasDataType value "long") or (hasDataType value "float") or (hasDataType value "double")) SubClassOf: AnalogVariable

- If a variable has the data type equal to string or any type of array, then, it is a Data Variable. The datatype "array" is defined as any type of array for simplicity reasons.

SWRL:

- *Variable(?v) ^ hasDataType(?v,"string"^^xsd:string) -> DataVariable(?v)*
- *Variable(?v) ^ hasDataType(?v,"array"^^xsd:string) -> DataVariable(?v)*

Equivalent OWL Expressions:
Variable and ((hasDataType value "string") or (hasDataType value "array")) SubClassOf: DataVariable

- A Function is a State Function if it is associated to a state.

SWRL:
Function(?f) ^ State(?s) ^ hasStateFunction(?s,?f) -> StateFunction(?f)

- A Function is a Transition Function if it is associated to a transition.

SWRL:
Function(?f) ^ Transition(?t) ^ hasTransitionFunction(?t,?f) -> TransitionFunction(?f)

- A Function is a Request Function if it calls a service.

SWRL:
Function(?f) ^ callsAService(?f,?s) -> RequestFunction(?f)
Equivalent OWL Expressions:
Function and (callsAService some Service) SubClassOf: RequestFunction

- If a Transition Function has a return value of True, the associated transition sets its "Is Enabled" property to True.

SWRL:
Transition(?t) ^ hasTransitionFunction(?t,?f) ^ TransitionFunction(?f) ^ hasReturnValue(?f,?v) ^ swrlb:equal(?v,"true"^^xsd:boolean)-> isEnabled(?t,"true"^^xsd:boolean)

- A Sequence is a Parent if it is initialized by an Initial State and finalized by a Final State.

SWRL:
Sequence(?seq) ^ InitialState(?is) ^ initializesSequence(?is, ?seq) ^ finalizesSequence(?fs, ?seq) ^ FinalState(?fs) ^ differentFrom(?is, ?fs) -> isParentSetElement(?seq, true)

Minified Standards Ontology (MSTO)

- The Standards have a data property referred to what agents it concerns to, according to on which kind of agent the standard can take effect. If an Agent has the characteristic of “Software Resource” and the Standard concerns to “Software Resouce”, then, the Standard standardizes such Agent.

  SWRL:
  - *Standard(?s) ^ Thing(?t) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"Thing") -> standardizes(?s,?t)*
  - *Standard(?s) ^ Device(?d) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"Device") -> standardizes(?s,?d)*
  - *Standard(?s) ^ SoftwareResource(?sr) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"SoftwareResource") -> standardizes(?s,?sr)*

- A Standard is context standard of a domain as long as the interest domain is normalized by the standard.

  SWRL:
  Domain(?d) ^ Process(?p) ^ belongsToSubdomain(?p, ?d) ^ isStandardizedBy(?d, ?s) ^ Standard(?s) -> isContextStandard(?s, true)

- The embedding capability of the standards is defined by the properties “Has Software Content”, “Has Official Resource”, and “Has DBpedia Resource” (the last two from the STO); if the standard has none of these properties, then, the standard has an embedding capability of “None”. If the standard has one of the three properties, it has an embedding capability of “Low”. If the standard has two of the three properties, it has an embedding capability of “Medium”. At last, if it has all the properties, then, it has an embedding capability of “High”.

  SWRL:
  - *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:equal(?sc, "") ^ swrlb:equal(?dbr, "") ^ swrlb:equal(?ores, "") -> hasEmbeddingCapability(?s, "None")*
  - *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:notEqual(?sc, "") ^ swrlb:notEqual(?dbr, "") ^ swrlb:notEqual(?ores, "") -> hasEmbeddingCapability(?s, "High")*
  - *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:notEqual(?sc, "") ^ swrlb:equal(?dbr, "") ^ swrlb:equal(?ores, "") -> hasEmbeddingCapability(?s, "Low")*
  - *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:equal(?sc, "") ^ swrlb:notEqual(?dbr, "") ^ swrlb:equal(?ores, "") -> hasEmbeddingCapability(?s, "Low")*
  - *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:equal(?sc, "") ^ swrlb:equal(?dbr, "") ^ swrlb:notEqual(?ores, "") -> hasEmbeddingCapability(?s, "Low")*
  - *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:notEqual(?sc, "") ^ swrlb:notEqual(?dbr, "") ^ swrlb:equal(?ores, "") -> hasEmbeddingCapability(?s, "Medium")*

- *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:notEqual(?sc, "") ^ swrlb:equal(?dbr, "") ^ swrlb:notEqual(?ores, "") -> hasEmbeddingCapability(?s, "Medium")*
- *Standard(?s) ^ hasSoftwareContent(?s,?sc) ^ hasDBpediaResource(?s,?dbr) ^ hasOfficialResource(?s,?ores) ^ swrlb:equal(?sc, "") ^ swrlb:notEqual(?dbr, "") ^ swrlb:notEqual(?ores, "") -> hasEmbeddingCapability(?s, "Medium")*

- An Agent can take different roles depending on its functionality and characteristics. This rule allows to identify agents which have more than one role, and likewise, which agents have several roles in the system. If an agent has a characteristic "Thing", it becomes an instance of the Thing class and similarly to the other classes.

  SWRL:
  - *Agent (?x) ^ hasFeature (?x, "Thing") -> Thing(?x)*
  - *Agent (?x) ^ hasFeature (?x, "Device") -> Device(?x)*
  - *Agent (?x) ^ hasFeature (?x, "Robot") -> Robot(?x)*
  - *Agent (?x) ^ hasFeature (?x, "SoftwareResource") -> SoftwareResource(?x)*

  Equivalent OWL Expressions:
  - *Agent and (hasFeature value "Thing") SubClassOf: Thing*
  - *Agent and (hasFeature value "Device") SubClassOf: Device*
  - *Agent and (hasFeature value "Robot") SubClassOf: Robot*
  - *Agent and (hasFeature value "SoftwareResource") SubClassOf: SoftwareResource*

Automation I4.0 Core Ontology

- An Agent can take different roles depending on its functionality and characteristics. This rule allows to identify agents which have more than one role, and likewise, which agents have several roles in the system. If an agent has a characteristic "Thing", it becomes an instance of the Thing class and similarly to the other classes.

  SWRL:
  - *Agent (?x) ^ hasFeature (?x, "Thing") -> Thing(?x)*
  - *Agent (?x) ^ hasFeature (?x, "Device") -> Device(?x)*
  - *Agent (?x) ^ hasFeature (?x, "Robot") -> Robot(?x)*
  - *Agent (?x) ^ hasFeature (?x, "SoftwareResource") -> SoftwareResource(?x)*

  Equivalent OWL Expressions:
  - *Agent and (hasFeature value "Thing") SubClassOf: Thing*
  - *Agent and (hasFeature value "Device") SubClassOf: Device*
  - *Agent and (hasFeature value "Robot") SubClassOf: Robot*
  - *Agent and (hasFeature value "SoftwareResource") SubClassOf: SoftwareResource*

- The agents have a data property referred to the architecture level (taking as reference the ISA S95/IEC 62264 standard architecture), and this value can define the kind of agent themselves. The value 1 associates the agents as Things, the value 2 associates them as Devices, and values from 2 to 5 (included) associates them as Software Resources.

  SWRL:
  - *Agent (?x) ^ belongsToArchitectureLevel(?x, ?y) ^ swrlb:equal(?y, 1) -> Thing(?x)*
  - *Agent (?x) ^ belongsToArchitectureLevel(?x, ?y) ^ swrlb:equal(?y, 2) -> Device(?x)*
  - *Agent(?x) ^ belongsToArchitectureLevel(?x, ?y) ^ swrlb:greaterThanOrEqual(?y, 3) ^ swrlb:lessThanOrEqual(?y, 5) -> SoftwareResource(?x)*

- The agents have a property referred to interoperability degree. If an agent supports some interoperable protocols, in this case, OPC UA and TCP-based protocols such as MQTT or HTTP, the agent has a "high" interoperability degree. If it only supports one of those assertions, it has a "medium" interoperability degree. If the agent does not support any of the assertions, it is provided with a "low" interoperability degree. Other combinations of desired supported protocols can be used instead; for instance, the FIPA ACL Protocol for agent communication.

SWRL:

- *Agent(?a) ^supportsProtocol(?a, ?p1) ^ swrlb:equal(?p1, "TCP") ^ supportsProtocol(?a, ?p2) ^ swrlb:equal(?p2, "OPC UA") ^ differentFrom(?p1, ?p2) -> hasInteroperabilityDegree(?a, "High")*
- *Agent(?a) ^ supportsProtocol(?a, ?p1) ^ swrlb:equal(?p1, "TCP") ^ supportsProtocol(?a, ?p2) ^ swrlb:notEqual(?p2, "OPC UA") ^ differentFrom(?p1, ?p2) -> hasInteroperabilityDegree(?a, "Medium")*
- *Agent(?a) ^ supportsProtocol(?a, ?p1) ^ swrlb:notEqual(?p1, "TCP") ^ supportsProtocol(?a, ?p2) ^ swrlb:equal(?p2, "OPC UA") ^ differentFrom(?p1, ?p2) -> hasInteroperabilityDegree(?a, "Medium")*
- *Agent(?a) ^ supportsProtocol(?a, ?p1) ^ swrlb:notEqual(?p1, "TCP") ^ supportsProtocol(?a, ?p2) ^ swrlb:notEqual(?p2, "OPC UA") ^ differentFrom(?p1, ?p2) -> hasInteroperabilityDegree(?a, "Low")*

- The Standards have a data property referred to what agents it concerns to, according to on which kind of agent the standard can take effect. If an Agent has the characteristic of "Software Resource" and the Standard concerns to "Software Resouce", then, the Standard standardizes such Agent.

  SWRL:

  - *Standard(?s) ^ Thing(?t) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"Thing") -> standardizes(?s,?t)*
  - *Standard(?s) ^ Device(?d) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"Device") -> standardizes(?s,?d)*
  - *Standard(?s) ^ SoftwareResource(?sr) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"SoftwareResource") -> standardizes(?s,?sr)*

- Each identifier must be unique for agents and services. If two agents have the same identifier, then, they are the same agent with two different roles. If two services have the same identifier, the model considers both as the same service, but unlike agents, two services with the same identifier in one system could throw errors in a real environment.

  SWRL:

  - *Agent(?x) ^ hasIdentifier(?x,?z) ^ Agent(?y) ^ hasIdentifier(?y,?z) -> sameAs(?x,?y)*
  - *Service(?x) ^ hasIdentifier(?x,?z) ^ Service(?y) ^ hasIdentifier(?y,?z) -> sameAs(?x,?y)*

- A data sent by any actor has a recipient and a sender, identifying who is the generator and the receiver of the data or information in the system.

  SWRL:

  - *Data(?d) ^ Actor(?a) ^ hasTransmitter(?d, ?ai) ^hasIdentifier(?a, ?ai) -> generates(?a, ?d)*
  - *Data(?d) ^ Actor(?a) ^ hasReceiver(?d, ?ai) ^ hasIdentifier(?a, ?ai) -> receives(?a, ?d)*