# 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 fire 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)*

* 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(?s2,?s3) ^ hasNextState(?s1,?s2) ^ hasNextState(?s1,?s3) ^ isConnectedToState(?t,?s1) ^ 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) ^ hasNextState(?s1,?s3) ^ hasNextState(?s1,?s4) ^ hasNextState(?s2,?s3) ^ hasNextState(?s2,?s4) ^ isConnectedToState(?t,?s1) ^ isConnectedToState(?t,?s2) ^ 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) ^ hasNextState(?s1,?s2) ^ isConnectedToState(?t,?s1) ^ 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) ^ hasNextState(?s1,?s3) ^ hasNextState(?s2,?s3) ^ isConnectedToState(?t,?s1) ^ isConnectedToState(?t,?s2) ^ isConnectedToState(?t,?s3) -> ManyToOneTransition(?t)*

* An Arc is a Unidirectional Arc if it has either a source point or a sink point but not both on the elements that it connects.

**SWRL:**

*Arc(?a) ^ hasSourcePoint(?a,?st) ^ hasSinkPoint(?a,?ts) ^ differentFrom(?st,?ts) -> UnidirectionalArc(?a)*

**Equivalent OWL Expressions:**

*Arc and not(BidirectionalArc) SubClassOf: UnidirectionalArc*

* An Arc is a Bidirectional Arc if it has both a source point and a sink point each on the state and the transition that it connects.

**SWRL:**

*Arc(?a) ^ State(?s) ^ Transition(?t) ^ hasSourcePoint(?a,?t) ^ hasSinkPoint(?a,?t) ^ hasSourcePoint(?a,?s) ^ hasSinkPoint(?a,?s) -> BidirectionalArc(?a)*

**Equivalent OWL Expressions:**

*Arc and not(UnidirectionalArc) SubClassOf: BidirectionalArc*

* A State has a Next State if it is source point of the arc of the transition between two states, and the transition is source point of an arc of the Previous State. The same rule applies with the has Previous State property considering the sink points. A State has a Previous State if it is sink point of the arc of the transition between to states, and the transition is sink point of an arc of the Next State.

**SWRL:**

*Transition(?t) ^ hasSinkPoint(?a1, ?t) ^ hasSourcePoint(?a1, ?s1) ^ State(?s1) ^ State(?s2) ^ hasSinkPoint(?a2, ?s2) ^ Arc(?a1) ^ hasSourcePoint(?a2, ?t) ^ Arc(?a2) ^ differentFrom(?s1, ?s2) ^ differentFrom(?a1, ?a2) -> hasNextState(?s1,?s2) ^ hasPreviousState(?s2,?s1)*

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

**SWRL:**

*Variable(?v) ^ Entity(?e) ^ actsOver(?v,?e) -> Output(?v)*

**Equivalent OWL Expressions:**

*Variable and (actsOver some Entity) SubClassOf: Output*

* If a variable Reads About an object, then, it is an Input.

**SWRL:**

*Variable(?v) ^ Entity(?e) ^ readsAbout(?v,?e) -> Input(?v)*

**Equivalent OWL Expressions:**

*Variable and (readsAbout some Entity) 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*

* If a state has a resource and the resource has a value of true, then, the state is active. Otherwise, the state is not active.

**SWRL:**

* + *State(?s) ^ hasResource(?s,?r) ^ hasValue(?r,"true"^^xsd:boolean) -> isActive(?s,"true"^^xsd:boolean)*
  + *State(?s) ^ hasResource(?s,?r) ^ hasValue(?r,"false"^^xsd:boolean) -> isActive(?s,"false"^^xsd:boolean)*
* An Entity is an Actor if it has processing capabilities (can execute tasks or functions).

**SWRL:**

*Entity(?e) ^ Function(?f) ^ executes(?e,?f) -> Actor(?e)*

**Equivalent OWL Expressions:**

*Entity and (executes some Function) SubClassOf: Actor*

* 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 Multivariable State Function if it is a State Function and has a number of parameters greater than one.

**SWRL:**

*StateFunction(?sf) ^ hasNumberOfParameters(?sf,?np) ^ swrlb:greaterThan(?np,1) -> MultivariableStateFunction(?sf)*

**Equivalent OWL Expressions:**

*StateFunction and (hasNumberOfParameters min 2) SubClassOf: MultivariableStateFunction*

* A Function is a Multivariable Transition Function if it is a Transition Function and has a number of parameters greater than one.

**SWRL:**

*TransitionFunction(?tf) ^ hasNumberOfParameters(?tf,?np) ^ swrlb:greaterThan(?np,1) -> MultivariableTransitionFunction(?tf)*

**Equivalent OWL Expressions:**

*TransitionFunction and (hasNumberOfParameters min 2) SubClassOf: MultivariableTransitionFunction*

* A Function is an As a Service Function if it runs a service.

**SWRL:**

* + *Function(?f) ^ runsAService(?f,?s) -> As-a-ServiceFunction(?f)*
  + *Actor(?a) ^ executes(?a,?f) ^ As-a-ServiceFunction(?f) -> executesAServiceFunction(?a,?f)*

**Equivalent OWL Expressions:**

*Function and (runsAService some Service) SubClassOf: As-a-ServiceFunction*

* 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 Sequence if it belongs to no sequences. A Sequence is also a Parent Sequence if it is initialized by an Initial State and finalized by a Final State. Otherwise, the Sequence is a Child Sequence, and thus, a Subsequence.

**SWRL:**

* *Sequence(?s1) ^ Sequence(?s2) ^ belongsToSequence(?s1,?s2) -> Subsequence(?s1) ^ isParentSetElement(?s1,"false"^^xsd:boolean)*
* *Sequence(?seq) ^ InitialState(?is) ^ FinalState(?fs) ^ initializesSequence(?is,?seq) ^ finalizesSequence(?fs,?seq) -> isParentSetElement (?seq,"true"^^xsd:boolean)*
* *isParentSetElement(?seq,?v) ^ swrlb:equal(?v,"true"^^xsd:boolean) -> Sequence(?seq)*
* *Sequence(?seq) ^ isParentSetElement(?seq,?v) ^ swrlb:equal(?v,"false"^^xsd:boolean) -> Subsequence(?seq)*
* *Subsequence(?subs) -> isParentSetElement(?subs, false)*

**Equivalent OWL Expressions:**

* + *(isParentSetElement value true) SubClassOf: Sequence*
  + *Sequence and (isParentSetElement value false) SubClassOf: Subsequence*
* A Sequence is a Loop if the state that finalizes it has the state that initializes it as next state.

**SWRL:**

* *Sequence(?seq) ^ State(?s1) ^ State(?s2) ^ initializesSequence(?s1,?seq) ^ finalizesSequence(?s2,?seq) ^ hasNextState(?s2,?s1) -> Loop(?seq)*
* *Sequence(?seq) ^ State(?s1) ^ State(?s2) ^ initializesSubsequence(?s1,?seq) ^ finalizesSubsequence(?s2,?seq) ^ hasNextState(?s2,?s1) -> Subloop(?seq)*
* A Loop is a Parent Loop if it is initialized by an Initial State and finalized by a Final State. Otherwise, the Loop is a Child Loop, and thus, a Subloop.

**SWRL:**

* *Loop(?l) ^ InitialState(?s1) ^ FinalState(?s2) ^ initializesLoop(?s1,?l) ^ finalizesLoop(?s2,?l) -> isParentSetElement(?l,"true"^^xsd:boolean)*
* *Loop(?l) ^ isParentSetElement(?l,?v) ^ swrlb:equal(?v, "false"^^xsd:boolean) -> Subloop(?l)*
* *Subloop(?sl) -> isParentSetElement(?sl, false)*

**Equivalent OWL Expressions:**

*Loop and (isParentSetElement value false) SubClassOf: Subloop*

* If any state or transition belongs to a subsequence, then, it belongs to the outer sequence.

**SWRL:**

* *State(?s) ^ belongsToSubsequence(?s,?subs) ^ belongsToSequence(?subs,?seq) -> belongsToSequence(?s,?seq)*
* *Transition(?t) ^ belongsToSubsequence(?t,?subs) ^ belongsToSequence(?subs,?seq) -> belongsToSequence(?t,?seq)*

# Minified Standards Ontology (MSTO)

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

**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) ^ Standard(?s) ^ isNormalizedBy(?s,?d) -> isContextStandard(?s, "true"^^xsd:boolean)*
* *Domain(?d) ^ Standard(?s) ^ hasDomain(?s, ?d) -> isContextStandard(?s, "true"^^xsd:boolean)*
* *Standard(?s) ^ isContextStandard(?s,?v) ^ swrlb:notEqual(?v, "true"^^xsd:boolean) -> isContextStandard(?s, "false"^^xsd:boolean)*
* 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 Actor can take different roles depending on its functionality and characteristics. This rule allows to identify actors which have more than one role, and likewise, which actors have several roles in the system. If an actor has a characteristic “Thing”, it becomes an instance of the Thing class and similarly to the other classes.

**SWRL:**

* *Actor(?x) ^ hasCharacteristic(?x, "Thing") -> Thing(?x)*
* *Actor(?x) ^ hasCharacteristic(?x, "Device") -> Device(?x)*
* *Actor(?x) ^ hasCharacteristic(?x, "SoftwareResource") -> SoftwareResource(?x)*
* *Actor(?x) ^ hasCharacteristic(?x, "Person") -> Person(?x)*

**Equivalent OWL Expressions:**

* *Actor and (hasCharacteristic value "Thing") SubClassOf: Thing*
* *Actor and (hasCharacteristic value "Device") SubClassOf: Device*
* *Actor and (hasCharacteristic value "SoftwareResource") SubClassOf: SoftwareResource*
* *Actor and (hasCharacteristic value "Person") SubClassOf: Person*

# Automation I4.0 Core Ontology

* An Actor can take different roles depending on its functionality and characteristics. This rule allows to identify actors which have more than one role, and likewise, which actors have several tasks in the system. If an actor has a characteristic “Thing”, it becomes an instance of the Thing class and similarly to the other classes.

**SWRL:**

* + *Actor(?x) ^ hasCharacteristic(?x, "Thing") -> Thing(?x)*
  + *Actor(?x) ^ hasCharacteristic(?x, "Device") -> Device(?x)*
  + *Actor(?x) ^ hasCharacteristic(?x, "SoftwareResource") -> SoftwareResource(?x)*
  + *Actor(?x) ^ hasCharacteristic(?x, "Person") -> Person(?x)*

**Equivalent OWL Expressions:**

* *Actor and (hasCharacteristic value "Thing") SubClassOf: Thing*
* *Actor and (hasCharacteristic value "Device") SubClassOf: Device*
* *Actor and (hasCharacteristic value "SoftwareResource") SubClassOf: SoftwareResource*
* *Actor and (hasCharacteristic value "Person") SubClassOf: Person*
* The actors 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 actor themselves. The value 1 associates the actors as Things, the value 2 associates them as Devices, and values from 2 to 5 (included) associates them as Software Resources.

**SWRL:**

* *Actor(?x) ^ belongsToArchitectureLevel(?x, ?y) ^ swrlb:equal(?y, 1) -> Thing(?x)*
* *Actor(?x) ^ belongsToArchitectureLevel(?x, ?y) ^ swrlb:equal(?y, 2) -> Device(?x)*
* *Actor(?x) ^ belongsToArchitectureLevel(?x, ?y) ^ swrlb:greaterThanOrEqual(?y, 3) ^ swrlb:lessThanOrEqual(?y, 5) -> SoftwareResource(?x)*
* Actors can communicate one another regardless of the architecture level as long as they have a common semantics and understand the same language and protocol.

**SWRL:**

*Actor(?x) ^ hasIdentifier(?x,?id1) ^ Actor(?y) ^ hasIdentifier(?y,?id2) ^ Language(?z) ^ Protocol(?p) ^ understands(?x, ?z) ^ understands(?y, ?z) ^ understands(?x, ?p) ^ understands(?y, ?p) ^ Information(?i) ^ hasTransmitter(?i, ?id1) ^ hasReceiver(?i, ?id2) -> interacts(?x, ?y)*

* The actors have a property referred to interoperability degree; it is assigned according to the interoperability that the actor supports. If an actor has non-interoperability compliant languages and non-interoperability compliant protocols, thus the actors has a “low” interoperability degree. If a language or protocol is interoperability compliant, the actor has a “medium” interoperability degree. And if a language and a protocol are interoperability compliant, the actor has a “high” interoperability degree.

**SWRL:**

* *Actor(?x) ^ Language(?l) ^ Protocol(?p) ^ understands(?x,?l) ^ understands(?x,?p) ^ compliesSemanticInteroperability(?l, ?smio) ^ compliesSyntacticInteroperability(?p, ?stio) -> hasInteroperabilityDegree(?x, "High")*
* *Actor(?x) ^ Language(?l) ^ Protocol(?p) ^ understands(?x, ?l) ^ understands(?x, ?p) ^ compliesSemanticInteroperability(?l, ?smio) ^ (compliesSyntacticInteroperability = 0)(?p) -> hasInteroperabilityDegree(?x, "Medium")*
* *Actor(?x) ^ Language(?l) ^ Protocol(?p) ^ understands(?x, ?l) ^ understands(?x, ?p) ^ (compliesSemanticInteroperability = 0)(?l) ^ compliesSyntacticInteroperability(?p, ?stio)-> hasInteroperabilityDegree(?x, "Medium")*
* *Actor(?x) ^ Language(?l) ^ Protocol(?p) ^ understands(?x, ?l) ^ understands(?x, ?p) ^ (compliesSemanticInteroperability = 0)(?l) ^ (compliesSyntacticInteroperability = 0)(?p) -> hasInteroperabilityDegree(?x, "Low")*

**ISSUE:**

The “High” rule worked, but the “Low” and “Medium” (2 rules) not. The “High” property worked because it can be declared without negations using SWRL common syntax, but the “Low” and “Medium” rules require a negation which is not possible to declare using SWRL common syntax; as stated in [87], SWRL does not support negation as a failure, which is necessary to correctly define these rules. A similar way to do it is provided in the Protégé support using a combination of SWRL syntax and OWL Expressions. They in [87] explain how to do it, changing, for example, the structure:

*Person(?p) ^ ¬ hasCar(?p, ?c) -> CarlessPerson(?p),*

For:

*Person(?p) ^ (hasCar = 0)(?p) -> CarlessPerson(?p),*

in the Protégé Rules tab. Even in the syntax of the version 5.2 was not possible to do. Protégé seems not to support negation as a failure yet (tested in version 5.5 beta and version 5.2).

* The Standards have a data property referred to what actors are concerned by, making the standardizes property inheritance to focus on the interest actors of the standard. This property depends on what standards are chosen in the application of the model.

**SWRL:**

* *Standard(?s) ^ Thing(?t) ^ standardizes(?s,?r) ^ defines(?r,?a) ^ isComposedBy(?a,?t) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"Thing") -> standardizes(?s,?t)*
* *Standard(?s) ^ Device(?d) ^ standardizes(?s,?r) ^ defines(?r,?a) ^ isComposedBy(?a,?d) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"Device") -> standardizes(?s,?d)*
* *Standard(?s) ^ SoftwareResource(?sr) ^ standardizes(?s,?r) ^ defines(?r,?a) ^ isComposedBy(?a,?sr) ^ concernsTo(?s,?cont) ^ swrlb:contains(?cont,"SoftwareResource") -> standardizes(?s,?sr)*
* If a language is defined by the representation as a formal language for the system and at least one actor can understand the language, the language complies semantic interoperability and inherits the formalization of the system’ semantics. Similarly occurs in the protocols, if the representation defines a protocol as formal, and one actor can understand it, the protocol complies syntactic interoperability and inherits the formalization of the system’ semantics.

**SWRL:**

* *Language(?l) ^ Representation(?r) ^ Semantics(?s) ^ defines(?r,?l) ^ SemanticInteroperability(?sio) ^ Actor(?x) ^ understands(?x, ?l) -> compliesSemanticInteroperability(?l, ?sio) ^ formalizes(?s,?l)*
* *Protocol(?p) ^ Representation(?r) ^ Semantics(?s) ^ defines(?r,?p) ^ SyntacticInteroperability(?sio) ^ Actor(?x) ^ understands(?x, ?p) -> compliesSyntacticInteroperability(?p, ?sio) ^ formalizes(?s,?p)*
* Each identifier must be unique for actors and services. If two actors have the same identifier, then, they are the same actor with two different roles. If two services have the same identifier, the model considers both as the same service, but unlike o actors, two services with the same identifier in one system could throw errors in a real environment.

**SWRL:**

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