An executable axiomatization of the REA^2 ontology

Abstract

The formalization of the REA^2 ontology presented in this paper offers a minimal set of operationalized semantics for a single white-box model relevant to all business stakeholders independent of their role or involvement in economic activities.

This paper's theoretical innovations are the use of MERODE to model increment and decrement semantics as fundamental stand-alone concepts that simultaneously affect economic resources, event, agents and the semantics of the stock-flow, participation and ownership associations and the formalization of the REA axioms as executable finite state machines. MERODE's possibilities for model execution through fast prototyping allowed validation through the modelling of an archetypical exchange scenario.

Both innovations contribute to the reliability of a generic semantic model for finance and logistics in both the traditional as well as the sharing economy, thus promoting traceability and accountability in value networks and supply chains supported by both centralized and decentralized ledger technologies.

Keywords

Resource, Event, Agent, Ontology, Axiomatization, Blockchain, MERODE, Model-Driven Engineering, Finite State Machines, Existence Dependency, Accountability, Traceability, Transparency

I. INTRODUCTION

Context

Blockchain is believed – by some – to be the silver bullet that will solve almost any issue in the world, where others – because of its current technological limitations – merely see it as a fad (Stratopoulos & Wang, 2018). The blockchain technology is particularly known for its applications in finance (e.g. Bitcoin), although it's impact on the non-financial economy (e.g. logistics) is also growing rapidly (e.g. Beef Ledger[1], Provenance[2]). Where blockchain already has an impact in both finance and logistics, the full power of the technology has, to the best of our knowledge, not yet been realized in both domains simultaneously.

What is needed is an approach that can link financial and logistical processes to each other, to promote transparency and fair trade, and fight money laundering and counterfeit (Allisson, 2019) and the funding of terrorism, in both centralized (e.g. blockchain) and decentralized (e.g. cloud) systems – independent of technological fads.

---

[1] https://beefledger.io

[2] https://www.provenance.org

To achieve this objective, this paper presents an executable formalization of the REA^2 ontology that should be applicable in both centralized and decentralized systems and has the power to support their semantic interoperability.

Introduction to REA and REA^2

The Resource-Event-Agent (REA) ontology (McCarthy, 1982) has been shown to possess the ability to unite financial and non-financial value streams and to serve as a conceptual foundation for blockchain implementations (Gal & McCarthy, 2018). REA has a conceptualization for describing an organization's metabolism (i.e. the value chain (McCarthy, 2003)) from the inside, and a conceptualization for describing the eco-system (e.g. supply chain, value network) that supports the organization from the outside. The former conceptualization is called the dependent or trading-partner view and was first documented by W.E. McCarthy in his 1982 paper. The latter conceptualization is called the independent (and sometimes helicopter) view. It represents the perspective of any third party in the business ecosystem not taking part in an economic exchange (e.g., investors, the government, the market, a random observer) and was first documented by Haugen (2007), together with the ISO/IEC (ISO/IEC, 2007).

The REA^2 ontology (Laurier, Kiehn, & Polovina, 2018) unites these two views demonstrating by means of an operationalized mapping that the three perspectives on an exchange (i.e. buyer, seller and third party) are information equivalent despite being semantically different.

The operationalized formalization of the REA^2 ontology presented in this paper should offer both centralized and decentralized information systems a minimal set of operationalized semantics to agree on a single white-box model of an economic exchange in a format that is relevant to all stakeholders independent of their involvement in the economic activities. The REA^2 ontology (Laurier, Kiehn, & Polovina, 2018), also offers the possibility to conceal information that gives trading-partners a competitive advantage even though it is available in a publishable format. This opting-out approach applied to transparency has two advantages: (1) the publisher can differentiate access to the data (e.g., the customs office has full access to the supply and production history, the national bank has full access to all monetary flows, where other stakeholders have no or limited access), (2) in case of emergency the relevant data can be published effortlessly and without delay (e.g., when food safety is at stake and recall of contaminated food is necessary, all relevant information is already available in publishable format and can instantly be shared with the general public, for example in a blockchain by sharing the key for deciphering the formerly private information).

Structure of this paper

Section II introduces the methodology of this project, discussing the Model-Driven Engineering (MDE) approach taken to transform the REA ontology into executable code, and the MERODE (i.e. Model-driven, Existence-dependency Relation, Object-oriented DEvelopment) approach taken to check

model coherence. Section III then shows the Platform Independent Models (PIMs). The first subsection introduces the existence dependency graph, which is a static conceptual model, of the REA^2 ontology. The third subsection formalizes the REA axioms as behavioral conceptual models, and the second subsection shows the methodology that checks the coherence of the latter and the former. Section IV presents an example scenario, and the paper concludes with section VI preceded by a discussion on the paper's limitations in section V.

II. METHODOLOGY

Model-driven engineering

This paper aims at operationalizing a truthful implementation of the REA ontology, taking an MDE (Schmidt, 2006) approach in which the REA ontology – and its formalization – serves as a Computation Independent Model (CIM). According to the MDE-approach a CIM is elaborated in a Platform Independent Model (PIM), which is suitable for a particular implementation technology domain, while abstracting from the implementation details. The PIM is finally transformed to code via an intermediary transformation to a Platform Specific Model (PSM), which covers the implementation details in the technology of choice. In this paper, the CIM is the REA^2 ontology as formalized by Laurier et al. (2018) and the PIM is

a static conceptual model (i.e. Existence Dependency Graph[3] (EDG)) complemented with a set of behavioral conceptual models (i.e. Finite State Machines (FSMs)). This paper considers these static and behavioral models to be a single PIM as the MERODE methodology explained in section 3.2 guarantees the coherence of the EDG and FSMs. The output of the MERLIN[4] CASE (i.e. Computer-Aided Software Engineering) tool, which is used for building the MERODE compliant PIM in this project, is used to generate directly executable Java code by means of the companion code-generator. The resulting Java application is subsequently used to validate the run-time behavior of the models presented below. In MERLIN and the MERODE code generator, the PSM is tacit. Figure 1 summarizes the above. Where the MERODE eco-system is currently limited to generating Java code, a webservice and an Ethereum smart contract generator (i.e. B-MERODE) are being developed (Amaral de Sousa, Burnay, & Snoeck, 2020; Schreynen, 2016).

---

[3] which is a formal subset of class diagrams relevant for the MERODE methodology.

[4] http://merode.econ.kuleuven.be/Tools.html

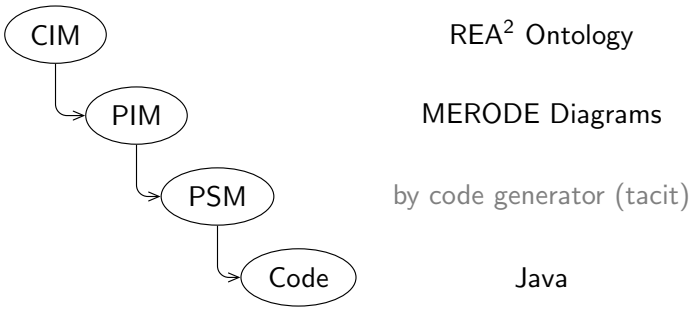

Figure 1: Project approach: Model-Driven Engineering

MERODE

The MERODE methodology (Snoeck, 2014) relies on the concept of existence dependency for its static conceptual model (i.e. the EDG).

"The existence-dependency relation is a partial ordering on objects and object types which is defined as follows: Let P and Q be object types. P is existence dependent on Q if and only if the life of each occurrence p of type P is embedded in the life of one single and always the same occurrence q of type Q. p is called the dependent object (P is the dependent object type) and is existence dependent on q, called the master object (Q is the master object type). " (Snoeck, 2014, pp. 83-84)

"A more informal way of defining existence dependency is as follows: If each object of an object type P always is associated with minimum one, maximum one and always the same occurrence of object type Q, then P is existence dependent on Q. In terms of life cycles, existence dependency means that the life of the existence- dependent object cannot

start before the life of its master. Similarly, the life of an existence-dependent object ends at the latest at the same time that the life of its master ends." (Snoeck, 2014, pp. 83-84)

For example, "a hotel customer and a room object will have to be created before a reservation object by that customer for that room can be created, and reversely, due to referential integrity rules, the reservation object will have to be terminated before life of the customer or the room object can be ended." *(Snoeck, 2014, p. 83)*

Next to existence dependency, MERODE relies on event propagation to check the coherence of EDG and FSMs. These propagated MERODE events are software events that differ considerably in granularity and semantics from the REA events (i.e. economic events and business events) discussed below.

The event propagation rule implies that *"if P is existence dependent on Q, then Q participates in each event in which P participates. In other words, each event marked for the dependent P must also be marked for the master Q. This can be explained as follows. When an existence-dependent object is involved in an event, its master objects are automatically involved in this event as well. For example, if a copy book is created, the corresponding book title is (implicitly) involved as well, as we now count one more copy for this title. Similarly, a state change of a book loan, e.g. because of the return of the copy, automatically implies*

a state change of the related book copy and library member: the copy is back on shelf and the member has one copy less in loan. (Snoeck, 2014, p. 115)

III. PLATFORM INDEPENDENT MODELS

Existence dependency graph for the REA² ontology

Figure 2 portrays the EDG for the REA ontology using the UML class diagram notation. It shows `EconomicResource`, `EconomicEvent` and `EconomicAgent` as the top-level object types on which the others depend. Economic resources have been defined as things that are scarce, have utility and are under the control of an agent. Economic events have been defined as phenomena that affect the value of scarce means (e.g. production, exchange, consumption). Economic agents have been defined as persons and organizations that participate in or are responsible for economic events (McCarthy, 1982).

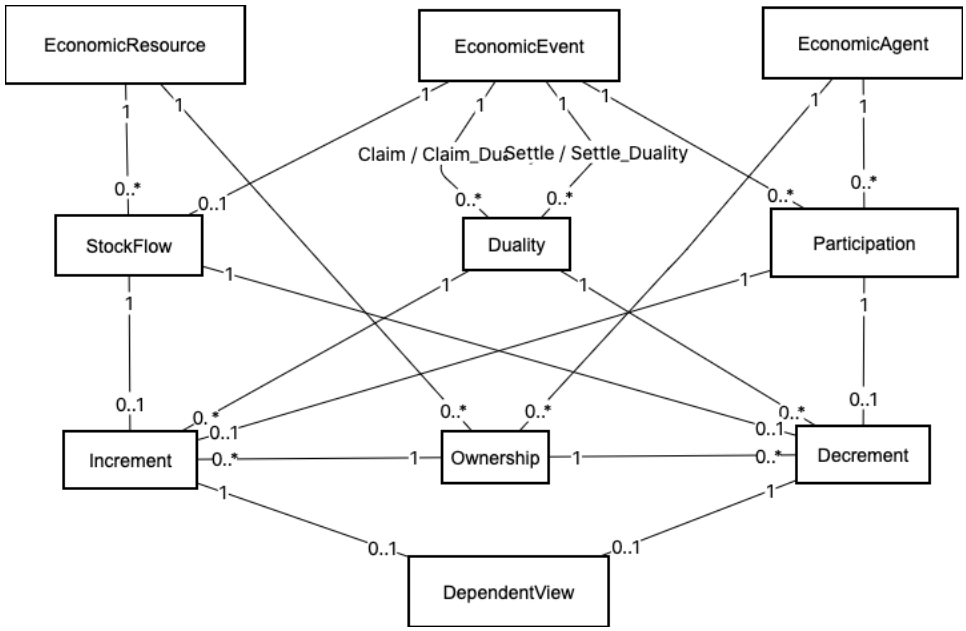

Figure 2: The REA[2] existence dependency graph

Figure 2 also shows that `StockFlow` objects[5] are existence dependent[6] on both `EconomicResource` and `EconomicEvent` objects for their existence.

---

[5] In the EDG, the stock-flow, ownership, duality and participation associations of the REA ontology are modeled as MERODE object types that are existence dependent on the `EconomicResource, EconomicEvent` and `EconomicAgent` object types.

[6] For reasons of readability all existence dependency relationship types in Figure 2 have been modeled as regular UML associations, even though these do not express the exact meaning of existence dependency. The modelling tool allows for the use a specific MERODE notation for a more accurate representation.

Stock-flow are reified associations between economic resources and the economic events that affect their value. The `StockFlow` object type results from applying the contract rule in MERODE.

The MERODE contract rule states that: *"When two object types share two or more event types, these common event types should be owned or shared by one or more common existence-dependent object types."* (Snoeck, 2014, p. 120) *"A contract should however always have a duration that is delimited by at least one creating event type and one ending event type. An additional object type cannot be the result of only one common event type."* (Snoeck, 2014, p. 122)

`Stock Flow` objects therefore act as association-class objects that connect `EconomicResources` to `EconomicEvents`. According to the MERODE principles, `Stock Flow` objects will therefore act as MERODE contracts between the `EconomicResource` and the `EconomicEvent` they connect.

In MERODE, whenever two object types are jointly involved in MERODE event types, the creation of such association or MERODE contract object type is mandatory (see Snoeck 2014, 120). From a behavioral perspective, the MERODE contract object `StockFlow` will ensure behavioral consistency of manipulations to the `EconomicResource` and the `EconomicEvent` by defining for the

common event types a set of scenarios that are acceptable for both the `EconomicResource` and the `EconomicEvent` objects.

See Snoeck 2014, page 139, below Fig. 6.13

As a result, when an object type has more than one master object type, the set of scenarios accepted by the dependent object type is in the intersection of the sets of scenarios accepted by all its master object types. Notice that not all scenarios in the intersection are necessarily relevant. The dependent object type can put additional constraints such that we obtain a subset of the intersection of the sets of scenarios of the master object types. In this sense, the intersection acts as a contract area and the dependent object type acts as the agreed contract between the master object types: for the common event types, it defines the set of scenarios that will be accepted by all master object types.

Similarly, the `Participation` object type acts as a MERODE contract between the `EconomicEvent` and `EconomicAgent` object type, while the `Duality` object type acts as MERODE contract between two distinct instances of the `EconomicEvent` object type, and the `Ownership` object type as MERODE contract between the `EconomicResource` and `EconomicAgent` object type. Participation associations relate economic agents to the economic events they participate in or are responsible for, duality associations relate claim inducing and

claim settling[7] economic events to each other, and ownership associations relate economic agents to the economic resources over which they will eventually have, have or have had economic control (Geerts & O'Leary, 2014).

As economic resources, events and agents can be involved in many stock-flow, duality, ownership or participation associations simultaneously, or none at all, the existence dependency relationship types between them have all been modelled with a zero-to-many multiplicity on the dependent side[8], except for the `EconomicEvent-StockFlow` existence dependency relationship type to which a zero-to-one multiplicity has been assigned. This means that each `EconomicEvent` object relates to exactly one `EconomicResource` object. When, for example, multiple resources need to be delivered, the resources will need to be packaged into a single resource by means of a conversion or multiple delivery events will be required. This zero-to-one multiplicity is necessary to enforce the REA axioms in opposing[9] views by inhibiting the creation of redundant views.

---

[7] Hence the `claim` and `settle` roles of the `EconomicEvent` object type in Figure 2.

[8] As a dependent object always depends on a single master for its entire life, the multiplicity on the master-side is always 1..1.

[9] The view of a buyer opposes that of a seller taking part in the same economic exchange as what is taken by one is given by the other. For example, when a cookie is exchanged for cash, the

Where the upper part (i.e. the part described above) of Figure 2 is aligned with the vast majority of the conceptual models representing the REA ontology, the lower part innovates by portraying the `Increment` and `Decrement` object type as fundamental independent concepts that have the potential to simultaneously affect the semantics of the entire upper part through event propagation resulting from existence dependency.

To add the dependent perspective of the buyer and seller to the third-party perspective that has been modeled with the MERODE object types discussed above, without losing the semantics of the third-party perspective, increment and decrement semantics, which inherently adhere to the perspective of the trading-partner that experiences the increment (e.g. as an economic resource inflow) or decrement (e.g. as an economic resource outflow), need to be modeled. This must be done independently of the `EconomicEvent` object type as the semantics of the increment and decrement might convey information (e.g. the value of the increment or decrement) that the trading-partners are not willing to share with each other or a third-party (e.g. a cost price). The third-party perspective on an economic exchange is that of any entity (e.g. person, authority) that does not take part in the exchange. For example, when a cookie is exchanged for cash, a third party sees a cookie transfer from the seller to the buyer and a money transfer from

---

buyer gives money and takes the cookie in return, where the seller gives the cookie and takes the money in return.

the buyer to the seller. Figure 2 shows the `Increment` object type as a stand-alone concept that has the potential to simultaneously connect a `StockFlow` object as an "inflow" of value to a `Participation` object as a "receiving" participation, whereas the `Decrement` object type has the potential to simultaneously connect a `StockFlow` object as an "outflow" of value to a `Participation` object as a "providing" participation. The multiplicity of the `StockFlow-Increment` and `StockFlow-Decrement` existence dependency relationship types is limited to zero-to-one, as a `StockFlow` object is either[10] an inflow or an outflow in a trading-partner view, and trading-partner views are limited to two opposing views in REA^2 (Laurier, Kiehn, & Polovina, 2018). Similarly, the multiplicity of the `Participation-Increment` and `Participation-Decrement` existence dependency relationship types is limited to zero-to-one, as a `Participation` object is either[11] receiving or providing in a trading-partner view. The multiplicity of the `Ownership-Increment` and `Ownership-Decrement` existence dependency relationship types has been modelled as zero-to-many, as an agent can lose and regain[12] ownership of a resource multiple times during his/her lifespan and that of the

[10] This mutual exclusiveness is expressed as a multiple propagation constraint. An overview of all multiple propagation constraints can be found in Appendix C.

[11] This multiple exclusiveness is expressed as a finite state machine shown in Figure 6.

[12] These semantics are expressed in Figure 3, which models the stock-flow axiom.

resource. For example, you can repurchase the car you sold earlier and sell it again afterwards. The zero-to-many multiplicities of both the `EconomicEvent-Duality` existence dependency relationship types allows increment and decrement (events) to be involved in multiple dualities simultaneously (e.g., a single monthly payment settles the debt for several weekly deliveries).

Finally, the `DependentView` object type accounts for the opposing views that trading-partners have on a `Duality` object thanks to the increment and decrement semantics that are simultaneously assigned to economic events. For example, a debtor sees a payment as a cash disbursement (i.e. decrement), where the creditor sees it as a cash receipt (i.e. increment). Hence, each trading-partner perceives the duality differently. For example, if a delivery is dual to a payment, the delivery is a shipment (i.e. the delivery as a decrement) paired in duality with a cash receipt (i.e. a payment as an increment) to the seller, and a receipt (i.e. a delivery as an increment) paired in duality with a cash disbursement (i.e. the payment as a decrement) to the buyer, who has an opposing view on the exchange (Laurier, Kiehn, & Polovina, 2018). This interpretation of the notion dependent view in REA is congruent with the notion of value interface, which shows the value object (i.e. economic resources) an actor (i.e. economic agent) is willing to exchange in return for other value objects, in e3-value, which is also an ontology that represents the independent view on business transactions (Gordijn, 2002).

As a single dependent view captures increment and decrement semantics, the multiplicities of the existence dependency relationships have been limited to zero-to-one, meaning that `Increment` and `Decrement` objects can only relate to a single `DependentView` object that acts as a MERODE contract between them. As such the semantics of the `Increment` and `Decrement` object types align with the notion of directed (i.e. "in" and "out") value ports that are exclusively assigned to a single value interface in e3value.

Object Event Table for the REA² ontology

The Object-Event Table (OET), which is an essential part of the MERODE methodology, offers a comprehensive overview of MERODE event propagation in an EDG by listing the events that can potentially change the state of an object that is an instance of one of the object types in the EDG. These lists then allow for checking the coherence between the EDG and object life cycles specified as (default) FSMs, as the latter depicts the potential state changes of such an object. In particular, dependent objects being always associated to their master, the master object will always be affected -at least indirectly- by a state change in one of its dependent objects, thus resulting in acquired event participations. In the OET, objects can therefore own (`o`) or acquire (`a`) MERODE events that create (`c`) or end (`e`) an object's life or modify (`m`) its state following MERODE's type of involvement rule (Snoeck, 2014, p. 118), which states:

- *If in the column of an existence-dependent object type a row contains a* `c`*, then on the same row a* `m` *(or* `c`*) must appear in the column of each master object type.*

- *If in the column of an existence-dependent object type a row contains an* `m`*, then on the same row an* `m` *must appear in the column of each master object type.*

- *If in the column of an existence-dependent object type a row contains an* `e`*, then on the same row a* `m` *(or* `e`*) must appear in the column of each master object type.*

Master object types acquire events from their dependent object types through event propagation according to the type of involvement rule cited above. Consequently, all creating, ending and modifying events of an object are acquired by the objects on which it depends. As event propagation prescribes that propagated events have the potential to change the state of the master object to which they propagate, they are all shown as acquired modifiers (i.e. `a/m`) in Table 1. The original events are either owned create-events, which are shown as `o/c`, owned end-events, which are shown as `o/e`, or owned modifiers, which are shown as `o/m`.

	EconomicResource	EconomicEvent	EconomicAgent	StockFlow	Duality	Participation	Increment	Ownership	Decrement	DependentView
cr_EconomicResource	O/C									
end_EconomicResource	O/E									
cr_EconomicEvent		O/C								
end_EconomicEvent		O/E								
cr_EconomicAgent			O/C							
end_EconomicAgent			O/E							
cr_StockFlow	A/M	A/M		O/C						
end_StockFlow	A/M	A/M		O/E						
cr_Duality		A/M A/M			O/C					
end_Duality		A/M A/M			O/E					
cr_Participation		A/M	A/M			O/C				
end_Participation		A/M	A/M			O/E				
cr_Increment	A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M	A/M	A/M	O/C	A/M		
end_Increment	A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M	A/M	A/M	O/E	A/M		
cr_Ownership	A/M		A/M					O/C		
end_Ownership	A/M		A/M					O/E		
cr_Decrement	A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M	A/M	A/M		A/M	O/C	
end_Decrement	A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M	A/M	A/M		A/M	O/E	
cr_DependentView	A/M A/M A/M A/M	A/M A/M A/M A/M A/M A/M A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M A/M	A/M A/M	A/M	A/M A/M	A/M	O/C
end_DependentView	A/M A/M A/M A/M	A/M A/M A/M A/M A/M A/M A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M A/M	A/M A/M	A/M	A/M A/M	A/M	O/E
end_LastDependentView	A/M A/M A/M A/M	A/M A/M A/M A/M A/M A/M A/M A/M	A/M A/M A/M A/M	A/M A/M	A/M A/M	A/M A/M	A/M	A/M A/M	A/M	O/E

Table 1: The Object-Event Table for Figure 2.

For example, Table 1 shows that when a `Duality` object is created (`o/c`), this object creation has the potential to change the state (`a/m`) of the two `EconomicEvent` objects on which it depends directly. A `Decrement` object, on the other hand, has the potential to change the state of the `Participation`, `Ownership`, `StockFlow` and `Duality` object on which it depends directly, and also has the potential to change the state of the `EconomicAgent`, `EconomicEvent` and `EconomicResource` object(s) on which it depends indirectly. Whether or not it actually changes the state of these objects will depend on the FSMs that model the behavior of the system. The following subsection presents such FSMs.

The Finite State Machines for the REA Axioms

This subsection models the REA axioms (Geerts & McCarthy, 2000) as FSMs[13]. When required, for modelling the FSMs that adhere to the EDG in Figure 2 and OET in Table 1, the original REA axioms have been split in lemmas or rephrased to align with the semantics of the FSMs that (partially) represent them.

[13] To the reader familiar with REA, it should be clear that these finite state machines are unrelated to those in (ISO/IEC, 2015)

The Stock-Flow Axiom

For the purpose of modeling the original stock-flow axiom *"At least one inflow event and one outflow event exist for each economic resource; conversely inflow and outflow events must affect identifiable resources."* (Geerts & McCarthy, 2000) has been split in two lemmas. The first lemma, which we will call the *ownership lemma*, stipulates that all economic resources in ownership should have an inflow and most eventually have an outflow. The second lemma, which we will call the *flow lemma*, prescribes that economic events must affect identifiable resources. Appendix B also discusses an additional FSM for assigning inflow and outflow semantics to `StockFlow` objects.

The Finite State Machine for the Ownership lemma. Figure 3 depicts the ownership lemma as an FSM for the `Ownership` object type. It shows that when an `Ownership` object is created (i.e. the `cr_Ownership` event in Table 1), the initial state of the object is `image`. In the `image` state, it can acquire the `inOwnership` state through the creation of an `Increment` object (i.e. the `cr_Increment` event in Table 1) that depends on the `Ownership` object in question. Simultaneously, according to Figure 2 this object depends on the receiving participation of the `EconomicAgent` object on which the `Ownership` object depends, a `Duality` object and a `StockFlow` object in an inflow (i.e. the inflow from the ownership lemma) state as specified by Figure A in Appendix B that affects the `EconomicResource` object the `Ownership` object relates to.

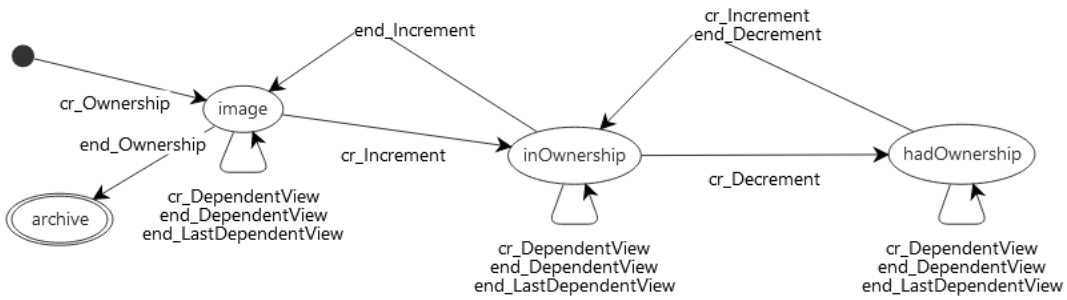

Figure 3: The ownership lemma as an FSM for the `Ownership` object type.

From the `inOwnership` state, an `Ownership` object can also acquire the `hadOwnership` state, through the creation of a `Decrement` object[14] (i.e. the `cr_Decrement` event in Table 1) that depends on the `Ownership` object. From the `hadOwnership` state, an `Ownership` object can regain the `inOwnership` state when an additional `Increment` object is assigned to it. This would represent, for example, the reacquisition of a resource the agent in question sold earlier. In case of error, the `Decrement` object can be deleted (i.e. the `end_Decrement` event in Table 1), which will also result in regaining the `inOwnership` state. The creation or deletion of `DependentView` objects (i.e.

[14] This object will also depend on a `StockFlow` object (i.e. the outflow from the ownership lemma) depending on the same `EconomicResource` object, a `Duality` and a `Participation` object depending on the same `EconomicAgent` object as the `Ownership` object in question.

the `cr_DependentView` event in Table 1 and the `end_DependentView` event in Table 1) does not affect the state of `Ownership` objects. Since the creation of `DependentView` objects is impossible without `Increment` or `Decrement` objects, the `cr_DependentView` event is impossible in the `image` state.

The Finite State Machine for the Flow Lemma. Figure 4 depicts both the flow lemma – on the left – and – on the right-hand side – the duality axiom that will be discussed in the next subsection. The leftmost state of this FSM shows that the initial state of an `EconomicEvent` object is `business_event`. In this initial state, the `EconomicEvent` object can only be associated with economic agents through the creation of `Participation` objects (i.e. the `cr_Participation` event in Table 1). In case of error, these `Participation` objects can also be deleted (i.e. the `end_Participation` event in Table 1).

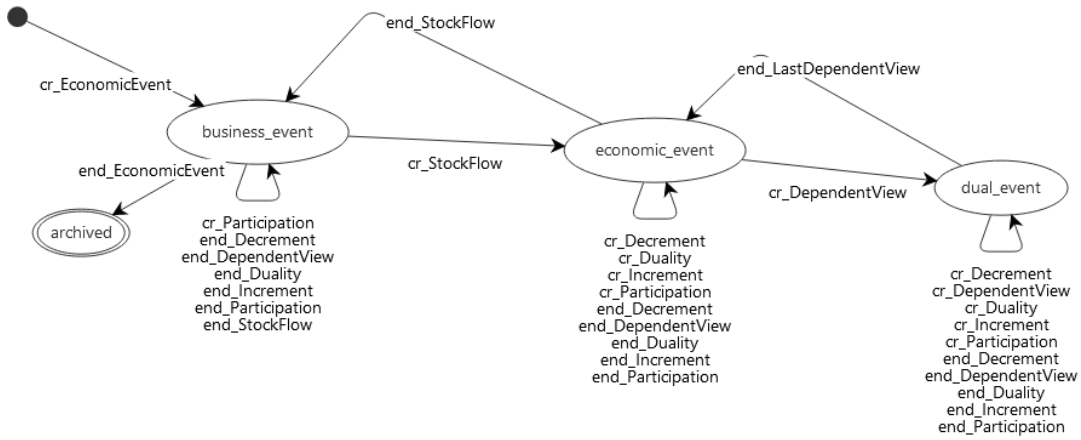

Figure 4: The flow lemma and duality axiom as an FSM for the `EconomicEvent` object type.

Figure 4 also shows that `EconomicEvent` objects only receive the state of `economic_event` when they affect identifiable resources, which is operationalized by the creation of a `StockFlow` object (i.e. the `cr_StockFlow` event in Table 1. Additional `Participation` objects can also be assigned in this state. In case of error, all directly and indirectly dependent objects can also be deleted. The deleting of a `StockFlow` object (i.e. the `end_StockFlow` event in Table 1) will make the `EconomicEvent` object lose its `economic_event` state, as the multiplicities in Figure 2 show that a single `EconomicEvent` object can only relate to a single `EconomicResource` object through a single `StockFlow` object.

The Duality Axiom

The duality axiom stipulates that *"All events effecting an outflow must be eventually paired in duality relationships with events effecting an inflow and vice-versa."* (Geerts & McCarthy, The Ontological Foundation of REA Enterprise Information Systems, 2000)

The right-hand side of Figure 4 shows that only `EconomicEvent` objects in an `economic_event` state (i.e. having a dependent `StockFlow` object) can be paired in duality through the creation of at least one `DependentView` object

(i.e. the cr_DependentView event in Table 1). Since economic events are allowed to be paired in many dualities, additional cr_Duality events are allowed in the dual_event state. Deleting the last DependentView object that indirectly depends on the EconomicEvent object (i.e. the end_lastDependentView event in Table 1) leads to the loss of the dual_event status.

As this paper has the ambition to present models that account for the dependent views of both trading-partners simultaneously, all economic events that represent transfers receive both inflow and outflow semantics. As economic events are not allowed to be dual to themselves, DependentView creating and ending events that propagate through the Increment object should not propagate to the same EconomicEvent object as the DependentView creating and ending events that propagate through the Decrement object. As dependent views are defined by economic agents, the DependentView creating and ending events that propagate through the Increment object should propagate to the same EconomicAgent object as the DependentView creating and ending events that propagate through the Decrement object.

Such constraints can be formulated in MERODE as multiple propagation constraints. These enforce that the masters reached via different paths should be

either equal or different[15]. The former multiple propagation constraint (MPC) could be rephrased as the dual event lemma: *distinct economic events are paired in duality*. The latter multiple propagation constraint could be rephrased as the dependent view lemma: *in each dependent view, one agent should play both a provider and a recipient role in distinct participations*.

The Participation Axiom

The participation axiom instructs that *"Each exchange needs an instance of both the inside and outside subsets."* (Geerts & McCarthy, 2000) The agent playing both the provider and recipient role in the multiple propagation constraint in the last paragraph of the section above could be seen as the inside agent in his own perspective, or the outside agent when taking the opposing perspective of his trading-partner.

The absence of inside and outside semantics in the independent view and the presence of inside and outside semantics in the dependent view, combined with this paper's ambition to account for both simultaneously (i.e. representing the buyer, seller and third-party perspective simultaneously) asks for rephrasing the participation axiom as the *exchange lemma* (i.e. In an exchange, each duality

[15] Appendix C offers an overview of all multiple propagation constraints in the model.

represents two opposing dependent views, defined by a single trading-partner each). The definition of the opposing views mentioned in this lemma requires inside and outside agent semantics to be replaced with provider and recipient semantics, which were introduced by Hruby (2006) and shown to be information equivalent to inside/outside semantics by Laurier and Poels (2014). The FSM for the provider and recipient semantics assigned to `Participation` objects is shown in Figure 6.

Figure 5 shows that the initial state of a `Duality` object is `image`. In this state, increment and decrement semantics can be assigned to it (i.e. the `cr_Increment` and the `cr_Decrement` event in Table 1). When these increment and decrement objects are related by a single `DependentView` object, the according duality is assigned conversion duality semantics in the `conversion` state. In this state, additional increment and decrement objects can be assigned to the duality. When these objects are assigned to a second `DependentView` object, the `Duality` object acquires the `exchange` state in which no further additions are allowed, as the number of dependent views on a duality is limited[16] to two opposing views.

---

[16] The constraint states that each `EconomicEvent` object can only be assigned a single `StockFlow` object, which in turn can be assigned only a single

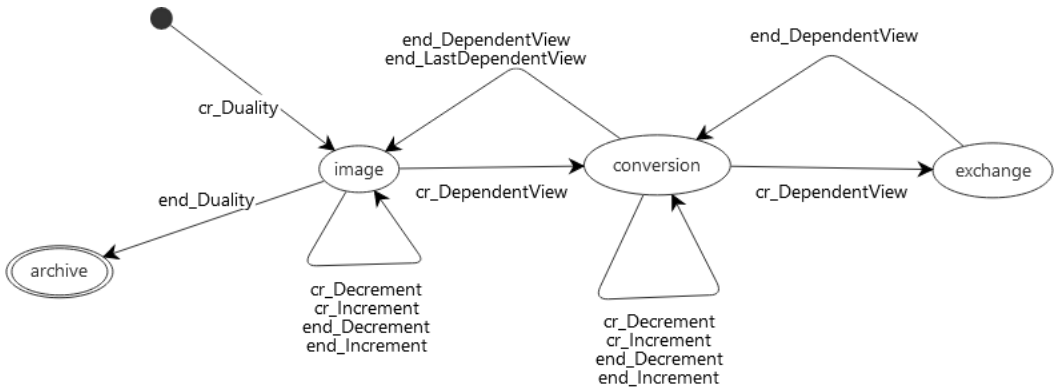

Figure 5: The exchange lemma as an FSM for the `Duality` object type.

`Increment` object, which corresponds to a receiving participation, and a single `Decrement` object, which assigns a 'provider' state to a providing `Participation` object. As each `Increment` and each `Decrement` object can only be used once as masters of a `DependentView` object, while a dependent view is a local view on a duality, and `Duality` objects depend on two distinct `EconomicEvent` objects, where `StockFlow` objects only depend on a single `EconomicEvent` object, the `Increment` and `Decrement` objects depending on a single `StockFlow` – and hence a single `EconomicEvent` – object cannot be dual. Consequently, only two dependent views that are inherently opposing can be created under these constraints.

The Finite State Machine for the Provider and Recipient Participation

Lemma. Both the exchange and dependent view lemma require that the provider and recipient role are mutually exclusive, which is enforced by the FSM for the *provider and recipient participation lemma* that stipulates that *each providing economic agent participates in a decrement event, where each receiving economic agent participates in an increment event*. Figure 6 formalizes the provider and recipient participation lemma as an FSM for the `Participation` object type. It shows a clear dichotomy between the `provider` and `recipient` state. It also shows that the `provider` state is reached by assigning decrement semantics to the event in which the agent participates by creating a `Decrement` object, and that the `recipient` state is reached by assigning increment semantics to the event in which the agent participates by creating an `Increment` object. The initial state of a `Participation` object has been named `undirected` in Figure 6.

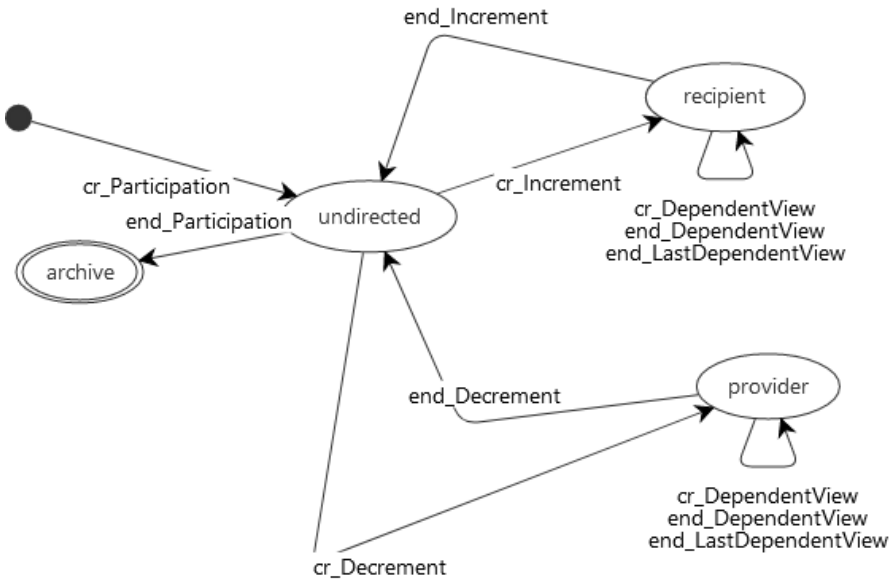

Figure 6: The provider and recipient participation lemma as an FSM for the `Participation` object type.

IV. EXAMPLE SCENARIO: ELMO & (COOKIE) MONSTER'S COOKIE EXCHANGE

The example scenario in Figure 7 is based on W.E. McCarthy's narrative on Elmo and Cookie Monster's (hereinafter abbreviated to Monster) cookie for cash exchange (McCarthy, 2004). This scenario can be implemented in the information system that can be generated from the models presented in this paper using the MERLIN code generator for MERODE. Based on the EDG, OET and FSMs the system is expected to enforce the REA axioms at run-time. This should help users to record data in a format that is complete, coherent and fully REA-compliant. This scenario serves as (1) an example of how a REA-axiom compliant system could be used, (2) a validation of the MERODE PIM by testing whether the system produces the desired behavior, (3) a proof of concept for REA-based systems implemented with the MERODE methodology and tools. The redundant object names have been chosen to facilitate understanding the scenario and the system. In a real-world application of the presented system, they could be replaced by hashes void of REA semantics.

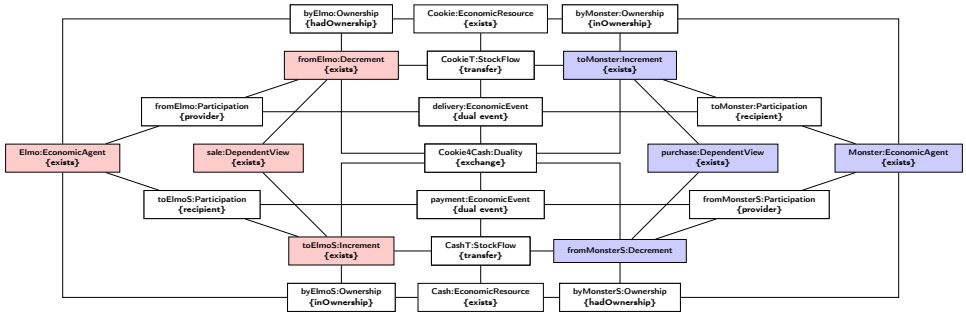

Figure 7: An object diagram for the example scenario: the cookie for cash exchange between Elmo and Monster.

Table 2 shows the state changes of the objects in the scenario that yields Figure 7. It starts with an economic agent (i.e. `Elmo:EconomicAgent`), who has an economic resource (i.e. `Cookie:EconomicResource`) in ownership[17], and another economic agent (i.e. `Monster:EconomicAgent`), who owns an economic resource (i.e. `Cash:EconomicResource`).

[17] How to get to the `inOwnership` state, even when it is not known how Elmo acquired the cookie (or how Monster acquired the cash), is explained in Appendix A.

event	Economic-Resource	Economic-Event	Economic-Agent	Stock-Flow	Ownership	Partici-pation	Duality	Increment	Decrement	Dependent-View
cr_Economic-Event	Cookie: exists Cash:exists	... delivery: business event	Monster: exists Elmo:exists	...	byElmo: inOwnership byMonster$: inOwnership	...	...	...	...	
cr_Economic-Event	Cookie: exists Cash:exists	... delivery: business event payment: business event	Monster: exists Elmo:exists	...	byElmo: inOwnership byMonster$: inOwnership	...	...	...	...	
cr_Stock-Flow	Cookie: exists Cash:exists	... delivery: economic event payment: business event	Monster: exists Elmo:exists	... CookieT: use	byElmo: inOwnership byMonster$: inOwnership	...	...	...	...	
cr_Stock-Flow	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster$: inOwnership	...	...	...	...	
cr_Ownership	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership	...	...	...	...	
cr_Ownership	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	...	...	...	...	
cr_Partici-pation	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: undirected	...	...	...	
cr_Partici-pation	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: undirected toMonster: undirected	...	...	...	
cr_Partici-pation	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: undirected toMonster: undirected fromMonster$: undirected	...	...	...	
cr_Partici-pation	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: undirected toMonster: undirected fromMonster$: undirected toElmo$: undirected	...	...	...	
cr_Duality	Cookie: exists Cash:exists	... delivery: economic event payment: economic event	Monster: exists Elmo:exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: undirected toMonster: undirected fromMonster$: undirected toElmo$: undirected	Cookie4Cash: image	...	...	

Table 2: Example Scenario part 1: The third-party perspective.

Elmo wants to exchange the `Cookie:EconomicResource` for the `Cash:EconomicResource`. Consequently, the exchange between Elmo and

Monster involves both a cookie transfer (i.e. `delivery:EconomicEvent`) and a cash transfer (i.e. `payment:EconomicEvent`). The first line of Table 2 represents the creation[18] of the `delivery:EconomicEvent` object. This object is in a `business_event` state at first. Figure 4 defines the `business_event` state as the initial state for `EconomicEvent` objects. The initial state of the `payment:EconomicEvent` object is also `business_event` as it is not related to a resource when it is created, as shown on the second line of Table 2.

Subsequently, the `delivery:EconomicEvent` is related to the `Cookie:EconomicResource` through the creation of the `CookieT:StockFlow` object to indicate that it is the `Cookie:EconomicResource` that will be delivered. As the `delivery:EconomicEvent` object now affects an identifiable resource, it acquires the `economic_event` state according to the stock-flow axiom, which is shown in Figure 4 and operationalized by the propagation of the `cr_StockFlow` event to the `delivery` object, as is shown on the third line of Table 2. As a consequence of this propagation, the `delivery:EconomicEvent` moves from the `business_event` to the `economic_event` state.

[18] The impact of each object creating event on the objects and their state is shown in red.

Through the creation of the CashT:StockFlow object, which depends on both the payment object and the Cash object, the payment object acquires the economic_event state, as is shown on the fourth line of Table 2. Both the CookieT:StockFlow and CashT:StockFlow object in Table 2 are in the use state, which is the initial state of StockFlow objects as defined by Figure A.

Afterwards, a new Ownership object is created to reflect that Monster:EconomicAgent wants to own the Cookie:EconomicResource. At this point Monster:EconomicAgent only has an image of the cookie he wants to acquire, which is shown as the image state of the byMonster:Ownership object as defined by the FSM in Figure 3 and show on the fifth line of Table 2. As Elmo:EconomicAgent is about to acquire ownership of the Cash:EconomicResource, a new Ownership object (i.e. byElmo$: Ownership) needs to be created. At creation it is also assigned the image state as shown on the sixth line of Table 2.

The next step is modeling that both Elmo:EconomicAgent and Monster:EconomicAgent will participate in the delivery:EconomicEvent. The next two lines of Table 2 show the creation of the fromElmo:Participation dependent on Elmo:EconomicAgent and delivery:EconomicEvent and the toMonster:Participation dependent on Monster:EconomicAgent and delivery:EconomicEvent. Next, the Participation objects for

Elmo:EconomicAgent and Monster:EconomicAgent 's participation in the payment:EconomicEvent are created. The fromMonster$:Participation object depends on the Monster:EconomicAgent object and the payment:EconomicEvent object. The toElmo$:Participation object depends on the Elmo:EconomicAgent object and the payment:EconomicEvent object. Both participations are in an undirected state at first, as defined by the FSM in Figure 6.

Finally, on the last line of Table 2, the delivery:EconomicEvent and payment:EconomicEvent objects are associated though the creation of the Cookie4Cash:Duality object. The initial state of the duality is image, as defined by the FSM in Figure 5.

Where Table 2 describes the third-party perspective on the cookie for cash exchange, Table 3 shows the increment and decrement semantics that depend on the perspective of each trading-partner (i.e. Elmo:EconomicAgent and Monster:EconomicAgent). In Table 3, the trading-partners assign increment and decrement semantics to the delivery:EconomicEvent and the payment:EconomicEvent objects. Elmo:EconomicAgent assigns decrement semantics to the delivery:EconomicEvent object as for him the delivery of Cookie:EconomicResource is a value decrease of a (stock of) scarce and valuable resource(s) (i.e. cookies). Monster:EconomicAgent, on

the other hand, assigns increment semantics to the `delivery:EconomicEvent` object as for him the delivery of `Cookie:EconomicResource` is a value increase of a (stock of) scarce and valuable resource(s). Additionally, `Elmo:EconomicAgent` assigns increment semantics to the `payment:EconomicEvent` object as for him the payment of `Cash:EconomicResource` is a value increase of a (stock of) scarce and valuable resource(s) (i.e. cash). `Monster:EconomicAgent`, on the other hand, assigns decrement semantics to the `payment:EconomicEvent` object as for him the payment of `Cash:EconomicResource` is a value decrease of a (stock of) scarce and valuable resource(s).

event	Economic-Resource	Economic-Event	Economic-Agent	Stock-Flow	Ownership	Partici-pation	Duality	Increment	Decrement	Dependent-View
cr_Duality	Cookie: exists Cash: exists	... delivery: economic event payment: economic event	Monster: exists Elmo: exists	... CookieT: use CashT: use	byElmo: inOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: undirected toMonster: undirected fromMonster$: undirected toElmo$: undirected	Cookie4Cash: image	...	...	
cr_Decrement	Cookie: exists Cash: exists	... delivery: economic event payment: economic event	Monster: exists Elmo: exists	... CookieT: outflow CashT: use	byElmo: hadOwnership byMonster: image byMonster$: inOwnership byElmo$: image	... fromElmo: provider toMonster: undirected fromMonster$: undirected toElmo$: undirected	Cookie4Cash: image	...	... fromElmo: exists	
cr_Increment	Cookie: exists Cash: exists	... delivery: economic event payment: economic event	Monster: exists Elmo: exists	... CookieT: transfer CashT: use	byElmo: hadOwnership byMonster: inOwnership byMonster$: inOwnership byElmo$: image	... fromElmo: provider toMonster: recipient fromMonster$: undirected toElmo$: undirected	Cookie4Cash: image	... toMonster: exists	... fromElmo: exists	
cr_Increment	Cookie: exists Cash: exists	... delivery: economic event payment: economic event	Monster: exists Elmo: exists	... CookieT: transfer CashT: inflow	byElmo: hadOwnership byMonster: inOwnership byMonster$: inOwnership byElmo$: inOwnership	... fromElmo: provider toMonster: recipient fromMonster$: undirected toElmo$: recipient	Cookie4Cash: image	... toMonster: exists toElmo$: exists	... fromElmo: exists	
cr_Decrement	Cookie: exists Cash: exists	... delivery: economic event payment: economic event	Monster: exists Elmo: exists	... CookieT: transfer CashT: transfer	byElmo: hadOwnership byMonster: inOwnership byMonster$: hadOwnership byElmo$: inOwnership	... fromElmo: provider toMonster: recipient fromMonster$: provider toElmo$: recipient	Cookie4Cash: image	... toMonster: exists toElmo$: exists	... fromElmo: exists fromMonster$: exists	
cr_Dependent-View	Cookie: exists Cash: exists	... delivery: dual event payment: dual event	Monster: exists Elmo: exists	... CookieT: transfer CashT: transfer	byElmo: hadOwnership byMonster: inOwnership byMonster$: hadOwnership byElmo$: inOwnership	... fromElmo: provider toMonster: recipient fromMonster$: provider toElmo$: recipient	Cookie4Cash: conversion	... toMonster: exists toElmo$: exists	... fromElmo: exists fromMonster$: exists	sale: exists
cr_Dependent-View	Cookie: exists Cash: exists	... delivery: dual event payment: dual event	Monster: exists Elmo: exists	... CookieT: transfer CashT: transfer	byElmo: hadOwnership byMonster: inOwnership byMonster$: hadOwnership byElmo$: inOwnership	... fromElmo: provider toMonster: recipient fromMonster$: provider toElmo$: recipient	Cookie4Cash: exchange	... toMonster: exists toElmo$: exists	... fromElmo: exists fromMonster$: exists	sale: exists purchase: exists

Table 3: Example Scenario part 2: Trading-partner dependent perspectives.

The first line of Table 3 matches the last line of Table 2, to stress that Table 3 builds on the semantics developed in Table 2. The second line of Table 3 shows the creation of a `fromElmo:Decrement` object, which depends on the `CookieT:StockFlow` object, the `byElmo:Ownership` object, the `Cookie4Cash:Duality` and the `fromElmo:Participation` object as

shown in Figure 7. The cr_Decrement event propagates to the CookieT:StockFlow, byElmo:Ownership, and fromElmo:Participation objects respectively changing their state to outflow, hadOwnership and provider as defined by their FSMs (i.e. Figure A, 3 and 6 respectively). The Cookie4Cash object did not change state as a result of the event propagation, since the FSM in Figure 4 shows that no state changes are expected when a cr_Decrement event propagates to a Duality object. The outflow state of the CookieT:StockFlow object represents the Cookie:EconomicResource object's loss of utility – and hence value – to Elmo:EconomicAgent, the hadOwnership state of the byElmo:Ownership object represents the loss of ownership and the provider state of the fromElmo:Participation object shows that Elmo:EconomicAgent gives the Cookie:EconomicResource away.

The next line of Table 3 shows the creation of the toMonster:Increment object, which depends on the CookieT:StockFlow object, the byMonster:Ownership object, the Cookie4Cash:Duality and the toMonster:Participation object as shown by Figure 7. The cr_Increment event propagates to the CookieT:StockFlow, byMonster:Ownership and toMonster:Participation objects respectively changing their state to transfer, inOwnership and recipient as defined by Figure A, 3 and 6 respectively. The Cookie4Cash:Duality

object did not change state as a result of the event propagation, since the FSM in Figure 4 shows that no state changes are expected when a `cr_Increment` event propagates to a `Duality` object. The `transfer` state of the `CookieT:StockFlow` object represents the `Cookie:EconomicResource` object's loss of utility to `Elmo:EconomicAgent`, which is at the same time a utility – and value – gain to `Monster:EconomicAgent`, the `inOwnership` state of the `byMonster:Ownership` object represents the gain of ownership by `Monster:EconomicAgent` and the `recipient` state of the `toMonster:Participation` object shows that `Monster:EconomicAgent` takes the `Cookie:EconomicResource`.

When `Elmo:EconomicAgent` assigns increment semantics to the `payment:EconomicEvent` object by creating the `toElmo$:Increment` object that depends on the `CashT:StockFlow`, `byElmo$:Ownership`, `toElmo$:Participation` and `Cookie4Cash:Duality` object as show in Figure 7. The fourth line of Table 3 shows the increment semantics propagation to the `CashT:StockFlow`, `byElmo$:Ownership` and `toElmo$:Participation` objects, respectively changing their state to `inflow`, `inOwnership` and `recipient` as defined by the FSMs of Figure A, 3 and 6 respectively. The `Cookie4Cash:Duality` object does not change state as a result of the event propagation, since the FSM in Figure 4 shows that no state

changes are expected when a `cr_Increment` event propagates to a `Duality` object.

At this point in the transfer, the state machines indicate that `Elmo:EconomicAgent` and `Monster:EconomicAgent` share ownership over the `Cash:EconomicResource` object, which can only be a transitory state[19]. When in the fifth line Table 3 `Monster:EconomicAgent` then assigns decrement semantics to the `payment:EconomicEvent` object by creating the `fromMonster$:Decrement` object, which depends on the `CashT:StockFlow`, `byMonster$:Ownership`, `fromMonster$:Participation` and `Cookie4Cash:Duality` object as show in Figure 7, the decrement semantics propagate to the `CashT:StockFlow`, `byElmo$:Ownership` and `fromMonster$:Participation` object respectively changing their state to `transfer`, `hadOwnership` and `provider` as defined by the FSMs of Figure A, 3 and 6 respectively. The `Cookie4Cash:Duality` object does not change state as a result of the event propagation, since Figure 4 shows that no state changes are expected when a `cr_Increment` event propagates to a `Duality` object.

[19] This could be mitigated by requiring that decrements have to be registered before the corresponding increments. Although, this might lead to issues with complex processes.

Where Table 3 now complies with the stock-flow axiom, it still has to satisfy the participation and duality axiom. The second-last line of Table 3 shows the creation of the `sale:DependentView` object. A `DependentView` object can only be created after creating the corresponding `Increment` and `Decrement` objects as its existence depends on them, which is defined by the EDG in Figure 2. This `sale:DependentView` object represents `Elmo:EconomicAgent`'s perspective on the exchange and changes the `Cookie4Cash:Duality` object's state to `conversion`, as defined by the FSM in Figure 5, since it is now perceived by a single trading-partner. As soon as the first trading-partner acknowledges the duality by creating a `DependentView` object, the `EconomicEvent` objects on which this `DependentView` object depends acquire the `dual_event` status, as is defined by the FSM in Figure 4. However, since there are two opposing views on an exchange, a second `DependentView` object needs to be created. The last line of Table 3 shows that such a second `DependentView` instance (i.e. `purchase:DependentView`) changes the state of a `Duality` object (here `Cookie4Cash:Duality`) from the `conversion` to the `exchange` state. Figure 4 shows that the `dual_event` status of an `EconomicEvent` object is only lost after the deletion of the last `DependentView` object that depends indirectly on that `EconomicEvent` object.

V. DISCUSSION AND LIMITATIONS

As the MERLIN CASE-tool does not yet support all aspects of the MERODE theory, some features of the models shown in this paper had to be realized by manually modifying the generated code. This is considered a limitation of the results presented in this paper. More details about the modification made are described in Appendix D.

Where most models in this paper are expected to be straight-forward for the REA community, this paper's interpretation of ownership might merit some discussion. This section discusses the special role of ownership in operationalizing the simultaneous representation of the buyer, seller and third-party perspective and its impact on the representation of economic resources.

In this paper, `Ownership` objects should be seen as the equivalent of the economic resources in the original REA axioms as they were written for the dependent view (i.e. a particular economic agent's trading-partner view). Therefore, the readers familiar with REA's trading-partner view might be inclined to assign the FSM for the `Ownership` object type to the `EconomicResource` object type. However, as the EDG accounts for a theoretically infinite number of trading-partners, each having their own view, `EconomicResource` objects should be seen as independent of a trading-partner's view. Through the use of ownership relationships, this paper distinguishes between those economic resources that are in an economic agent's ownership and those that are not. As a result, the

`EconomicResource` object type in this paper represents view independent economic resources. As a consequence, the stock-flow axiom was decomposed in an ownership lemma and a flow lemma. The ownership lemma applies the parts of the original axiom relevant for economic resources in a dependent view to ownership associations in an independent view, as the dependent view only considers the economic resources that are relevant (i.e. those that are in its ownership) for the economic agent defining the view.

An additional limitation relates to this paper's focus on exchanges. Although this paper has the ambition to present a fully-fledged REA model, the paper only reports on the model's ability to document exchanges. Consequently, future research should demonstrate how a more elaborate version of the PIM presented in this paper can be used to document all possible configurations of conversions and incorporate the law of conservation of matter and mass in the automatically generated application. Where this paper uses the ownership construct, it considers modeling custody, possession and availability as discussed by Geerts and O'Leary (2014) and Scheller and Hruby (2016) out of scope although they merit a proper discussion.

VI. CONCLUSION

This paper introduced a new approach for formalizing the REA axioms, which is based on the MERODE model-driven engineering (MDE) methodology. The fact that MERODE is an MDE approach allows for generating software that

can be used to validate the operational behavior of the models. Additionally, this paper demonstrates how the notions of existence dependency and event propagation allow for modeling and operationalizing the fundamental nature of increment and decrement semantics in REA, and how they simultaneously affect the semantics of participation, stock-flow, duality and ownership associations. This semantic orchestration based on the REA axioms is, to the best of your knowledge, new to the REA literature and the main contribution of this paper. Finally, the development of the B-MERODE code generator for MERODE is expected to allow for enforcing the REA-axioms in blockchain applications generated from the platform independent model (PIM) introduced in this paper.

Where this paper solely focusses on the operationalization of the REA axioms in an MDE approach, it could be considered a proof of concept for incorporating run-time business rule checking in software. We believe this to be particularly relevant to smart contracts as there is no central authority to check these business rules ex-ante and reverting (erroneous or illegal) transactions in a blockchain is at least cumbersome.

Additionally, the ontology-based model-driven engineering approach with integrated coherence checking has the potential to make smart contracts and software in general more robust through transparence and validation, as the approach presented in this paper uses an ontology as a PIM, replacing a conceptual

model that is potentially created by a single modeler by an ontology, which is shared within, supported by and validated by a community. Moreover, replacing manual coding by a code generator, replaces the model interpretation of a single programmer with a set of validated transformation rules, which is expected to make the model-to-code or model-to-contract transformation more transparent and less error-prone. Finally, the MERODE approach used in this paper checks the conceptual model for certain types of logical errors (e.g. deadlock, infinite loops, indeterministic behavior).

VII. BIBLIOGRAPHY

Allisson, I. 2019. *Coindesk*. Retrieved May 26, 2019, from https://www.coindesk.com/louis-vuitton-owner-lvmh-is-launching-a-blockchain-to-track-luxury-goods

Amaral de Sousa, V., C. Burnay, and M. Snoeck. 2020. B-MERODE: A Model-Driven Engineering and Artifact-Centric Approach to Generate Blockchain-Based Information Systems. In Y. E. Dustdar S. (Ed.), *32nd International Conference on Advanced Information Systems Engineering. 12127.*, 117-133. Grenoble, France: Lecture Notes in Computer Science, Springer, Cham.

Borst, W. 1997. Construction of Engineering Ontologies. *PhD thesis, Institute for Telematica and Information Technology*. Enschede, The Netherlands: University of Twente.

Gal, G., and W. E. McCarthy. 2018. Implementation of REA Contracts as Blockchain Smart Contracts: An Exploratory Example. *Proceedings of the 12th International Workshop on Value Modeling and Business Ontologies* (pp. 107-112). Amsterdam, The Netherlands: CEUR Workshop Proceedings.

Geerts, G. L., and W. E. McCarthy. 2000. The Ontological Foundation of REA Enterprise Information Systems. Alabama: https://msu.edu/user/mccarth4/Alabama.pdf.

Geerts, G. L., and D. E. O'Leary. 2014. A supply chain of things: The EAGLET ontology for highly visible supply chains. *Decision Support Systems, 63*, 3-22.

Gordijn, J. 2002. *Research Publications.* Retrieved June 2nd, 2019, from e3value: https://research.e3value.com/docs/bibtex/pdf/e3valueNutshell.pdf

Guarino, N., D. Oberle and S. Staab 2009. What is an ontology? In S. Staab, & R. Studer, *Handbook on Ontologies* (pp. 1-17). Berlin: Springer-Verlag.

Haugen, R. 2007. Beyond the Enterprise: Taking REA to Higher Levels. *REA-25 conference.* Newark, Delaware, USA: http://www.aisvillage.com/rea25/program/haugen.pdf.

Hruby, P. 2006. *Model-driven Design Using Business Patterns.* Heidelberg: Springer.

ISO/IEC. 2015. *ISO/IEC 15944-4:2015 Information technology — Business Operational View — Part 4: Business transaction scenarios — Accounting and economic ontology.*

ISO/IEC. 2007. *ISO/IEC 15944-4:2007 Preview Information technology -- Business operational view -- Part 4: Business transaction scenarios -- Accounting and economic ontology.* Retrieved March 13th, 2019, from International Organisation for Standardization: https://www.iso.org/standard/67199.html

Laurier, W. P., and G. Poels. 2014. An Enterprise Ontology Based Conceptual Modeling Grammar for Representing Value Chain and Supply Chain Scripts. *International Journal of Conceptual Structures and Smart Applications*, 18-35.

Laurier, W. P., J. Kiehn and S.Polovina. 2018. REA 2: A unified formalisation of the Resource-Event-Agent ontology. *Applied Ontology,13(3)*, 201-224.

McCarthy, W. E. 1982. The REA accounting model: A generalized framework for accounting systems in a shared data environment. *Accounting Review,57(3)*, 554-578.

McCarthy, W. E. 2003. The REA modeling approach to teaching accounting information systems. *Issues in Accounting Education,18(4)*, 427-441.

McCarthy, W. E. 2004. *An REA Model of an Economic Exchange.* Retrieved May 27, 2019, from Michigan State University: https://msu.edu/user/mccarth4/cookie--elmo--basic%20REA.ppt

Scheller, C. V., and P. Hruby. 2016. Business Process and Value Delivery Modeling Using Possession, Ownership, and Availability (POA) in Enterprises and Business Networks. *Journal of Information Systems,30(2)*, 5-47.

Schmidt, D. C. 2006. Model-driven engineering. *IEEE Computer, 39*(2), 25-31.

Schreynen, N. 2016. *Construction of web services using the MERODE approach.* KULeuven, Faculteit Economie en Bedrijfswetenschappen. Leuven: KULeuven.

Snoeck, M. 2014. *Enterprise Information Systems Engineering, The MERODE Approach.* . Heidelberg: Springer.

Stratopoulos, T. C., and V. X. Wang. 2018. *Blockchain Technology Adoption.* Retrieved May 26, 2019, from SSRN: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3188470

VIII. APPENDIX

Appendix A: Starting a Traceability Chain

Table A shows how to start a traceability chain, for example, when the origin of Elmo's cookie is not known. The ownership lemma requires that the

cookie has an inflow. Therefore, the traceability chain needs to start with a paired event that is the origin of the cookie. This event is initially paired with a dummy event and can then later be paired in duality with information about how Elmo acquired the cookie, to backtrack the origin of the cookie. It could for example be paired in duality with events consuming flour, butter, milk, baking time, etc., when Elmo baked the cookie himself, or with a purchase in case Elmo bought the cookie.

event	Economic-Resource	Economic-Event	Economic-Agent	Stock-Flow	Ownership	Partici-pation	Duality	Increment	Decrement	Dependent-View
cr_Economic-Resource	Cookie: exists									
cr_Economic-Event	Cookie: exists	cookieInflow: business event								
cr_Economic-Agent	Cookie: exists	cookieInflow: business event	Elmo: exists							
cr_Stock-Flow	Cookie: exists	cookieInflow: economic event	Elmo: exists	CookieP: use						
cr_Ownership	Cookie: exists	cookieInflow: economic event	Elmo: exists	CookieP: use	byElmo: image					
cr_Partici-pation	Cookie: exists	cookieInflow: economic event	Elmo: exists	CookieP: use	byElmo: image	ofElmo: undirected				
cr_Economic-Resource	Cookie: exists dummy: exists	cookieInflow: economic event	Elmo: exists	CookieP: use	byElmo: image	ofElmo: undirected				
cr_Economic-Event	Cookie: exists dummy: exists	cookieInflow: economic event dummy: business event	Elmo: exists	CookieP: use	byElmo: image	ofElmo: undirected				
cr_Stock-Flow	Cookie: exists dummy: exists	cookieInflow: economic event dummy: economic event	Elmo: exists	CookieP: use dummy: use	byElmo: image	ofElmo: undirected				
cr_Duality	Cookie: exists dummy: exists	cookieInflow: economic event dummy: economic event	Elmo: exists	CookieP: use dummy: use	byElmo: image	ofElmo: undirected	dummy : image			
cr_Increment	Cookie: exists dummy: exists	cookieInflow: economic event dummy: economic event	Elmo: exists	CookieP: cookieInflow dummy: use	byElmo: inOwnership	ofElmo: recipient	dummy : image	forElmo: exists		

Table A: Example Scenario part 0: Starting a Traceability Chain.

Table A starts with the creation of the `Cookie:EconomicResource` object. Since the `EconomicResource` object type was not assigned a

customized FSM, it is assigned the MERODE default FSM, which has a single state called `exists`. Next, the `cookieInflow:EconomicEvent` object is created. This object receives the initial state of `EconomicEvent` objects defined by the FSM in Figure 4, which is `business_event`. Thereafter, the `Elmo :EconomicAgent` object is created. As the `EconomicAgent` object type was not assigned a particular FSM, it is also assigned MERODE's default FSM.

Subsequently, the `Cookie:EconomicResource` is associated with the `cookieInflow:EconomicEvent` through the creation of the `CookieP:StockFlow` object. The creation of this object results in a state change of the `inflow` object, from `business_event` to `economic_event` as defined by the FSM in Figure 4. Since this `cookieInflow:EconomicEvent` object now affects identifiable resources, it is considered a fully-fledged economic event. The `CookieP:StockFlow` object, on the other hand, is assigned the initial state `use` as defined by the FSM in Figure A. Then the `byElmo:Ownership` object is created, its initial state (i.e. `image`) is defined by the FSM in Figure 3. After that a `ofElmo:Participation` object with an initial state `undirected` as defined by the FSM in Figure 6 is created.

Then a `dummy:EconomicResource`, `dummy:EconomicEvent` and a `dummy:StockFlow` object that relates them are created such that the `cookieInflow:EconomicEvent` object satisfies the stock-flow axiom. The `dummy:EconomicEvent` is then associated with the

cookieInflow:EconomicEvent through a dummy:Duality object. Since we have no information about Elmo obtaining the Cookie:EconomicResource through an exchange or conversion, nor the participation nor the duality axiom cannot be satisfied at this point. Transactions data that do not satisfy all REA axioms can only be accepted at the start of a traceability chain where they highlight that the origin of an economic resource is not fully documented within the system or at the end where they signal that a new transaction is ongoing or has not been fully recorded in the system yet.

Finally, the forElmo :Increment object is created that assigns inflow semantics to the CookieP:StockFlow object on which it depends, inOwnership semantics to the byElmo:Ownership object on which it depends, and recipient semantics to the ofElmo:Participation object on which it depends, as respectively defined by the FSMs in Figure A, 3 and 6. The inOwnership semantics of the Ownership association between the Elmo:EconomicAgent and the cookie:EconomicResource object, will subsequently allow Elmo to sell the cookie he now officially owns, as shown in Table 2 and 3. The same procedure needs to be executed for the acquisition of the cash Monster is about to spend.

Appendix B: The transfer lemma

The *transfer lemma* instructs that each transfer is seen as an inflow (i.e. increment) by a receiving economic agent and an outflow (i.e. decrement) by a

providing economic agent. Figure A then shows the transfer lemma as an FSM for the `StockFlow` object type. It shows that `StockFlow` objects can be assigned increment semantics or decrement semantics or both. When the `StockFlow` object receives increment semantics through the creation of a dependent `Increment` object, it's an `inflow` of a resource. When the `StockFlow` object is assigned decrement semantics, it's an `outflow` of a resource. However, when a `StockFlow` object is assigned both increment and decrement semantics simultaneously, it has to be a `transfer` assigned increment semantics by a recipient and decrement semantics by a provider with an opposing view because: (1) `Increment` and `Decrement` objects both depend on `StockFlow`, `Participation` and `Ownership` objects simultaneously, which have been shown to change the state of these objects in Figure A, 6 and 3 respectively, and (2) a `Participation` object either has `provider` or `recipient` semantics, but not both.

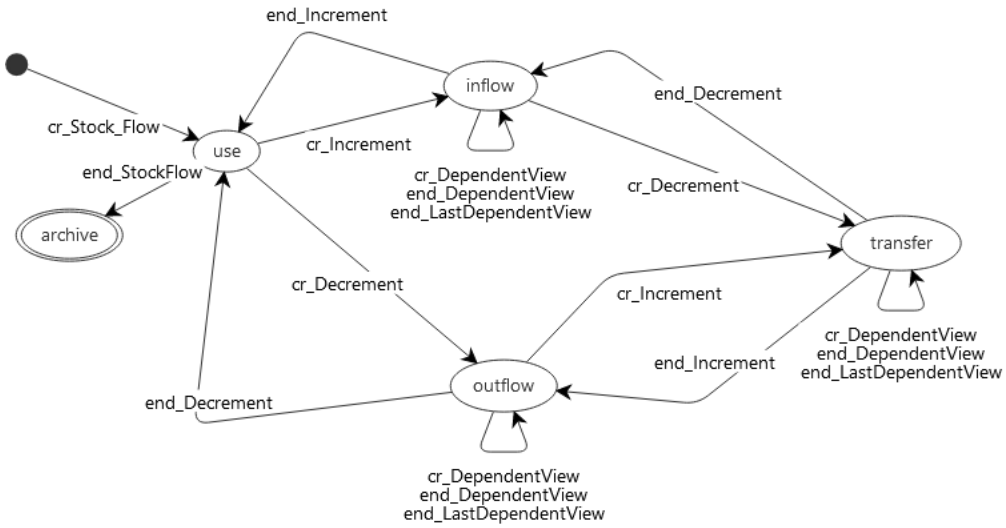

Figure A: The transfer lemma as an FSM for the `StockFlow` object type.

Appendix C: Multiple Propagation Constraints

Operationalizing the REA axioms with the MERODE methodology, this paper aims at developing a formal (i.e. in the sense of computer readable) specification of the shared conceptualization of the world of business offered by the REA ontology. The notion of a formal specification of a shared conceptualization refers to Borst's (1997) definition of ontology. In order to obtain the best semantic fit between the set of intended models, which represent the shared conceptualization of business, and the set of possible models (i.e. the extensions) according to the formal specification developed with MERODE a proper set of formal constraints is required (Guarino, Oberle, & Staab, 2009). Unfortunately, the EDG, OET and FSMs are insufficient to fully disambiguate this formal

specification. Therefore, additional constraints are needed. In particular, multiple propagation constraints (MPCs) allow for constraining the number of objects and the number of ways object are affected by MERODE event propagation in case these events can propagate up the EDG via multiple paths composed of the existence dependency relationships formalized by the EDG.

`EconomicResource`, `EconomicEvent` and `EconomicAgent` are the top-level object types in Figure 2, as they don't depend on any other object type. Consequently, their MERODE (i.e. creating, modifying and ending) events do not propagate to any other object on which they depend. Hence, they do not require MPCs.

The `StockFlow`, `Ownership` and `Participation` object types in Figure 2 respectively are MERODE contracts between the `EconomicResource` and `EconomicEvent`, between the `EconomicResource` and `EconomicAgent` and between `EconomicEvent` and `EconomicAgent` object types. As their events can only propagate to their master object type (i.e. object type) via a single path, no MPCs are needed either. For example, the events of the `StockFlow` object type can only propagate to the `EconomicResource` object type and to the `EconomicEvent` object type through the existence dependency relationship types that links them to each other. The events of the

Ownership object type can only propagate to the EconomicResource object type and to the EconomicAgent object type through the existence dependency relationship types that links them to each other. The events of the Participation object type can only propagate to the EconomicEvent object type and to the EconomicAgent object type through the existence dependency relationship types that links them to each other.

The Duality object type, on the other hand can propagate events to the EconomicEvents object type via two different paths (i.e. the claim and settle existence dependency relationship types). At runtime, these paths allow Duality objects to propagate their events to either a single (i.e. to an A:EconomicEvent object through the claim existence dependency relationship and to the same A:EconomicEvent object through the settle existence dependency relationship) or two distinct EconomicEvent objects (i.e. to a B:EconomicEvent object through the claim existence dependency relationship and to a different C:EconomicEvent object through the settle existence dependency relationship). As in REA a duality relates two distinct objects via the claim and settle associations to EconomicEvent. Consequently, the former case in which A:EconomicEvent object is dual to itself is an unintended model instantiation that should not be included in the possible model instantiations according to our formal specification. As a result, the latter case has to be enforced

with the following multiple propagation constraint[20] for the `Duality` object type shown in MPC 1.

```
self.claim ≠ self.settle
```

MPC 1: The MPC for propagation from `Duality` to `EconomicEvent`

The symbol ≠ in MPC 1 represents "not equal to" semantics. MPC1 expresses that for any `Duality` object (i.e. `self`) the `EconomicEvent` object (i.e. `EconomicEvent`) it is related to via the `claim` existence dependency

---

[20] Navigation paths from a dependent to its master have per default the following (simplified) syntax:

`self.[parent object type name].` In case two associations exist between the same dependent an master, the names of the associations are used instead: `self.[existence dependency relationship name].`

For longer paths the path notation above is extended with `.[parent object type name]` or `[existence dependency relationship name]` blocks until the desired parent object type is reached:

`self.[parent object type name] .[parent object type name]…`

relationship, which represents the path between them, should differ from the `EconomicEvent` object it is related to through the `settle` existence dependency relationship.

The events of the `Increment` object type can propagate to the `EconomicResource` object type through the `StockFlow` and `Ownership` object types. Consequently, these events can propagate to the same or different `EconomicResource` objects. However, as we want the creation of an `Increment` object to change the semantics of the `StockFlow` (i.e. `use` to `inflow`) and `Ownership` (i.e. `image` or `hadOwnership` to `inOwnership`) associations pertaining a same `EconomicResource` object simultaneously, a multiple propagation constraint that enforces their propagation to one and the same `EconomicResource` object is needed. This constraint is MPC 2.

```
self.StockFlow.EconomicResource=
self.Ownership.EconomicResource
```

MPC 2: The MPC for propagation from `Increment` to `EconomicResource`

The symbol = in MPC 2 directly above represents “equal to” semantics. Hence it is the opposite of the ≠ notation that we find in MPC 1. It expresses that every `Increment` object (i.e. `self`) that relates to an object of type

EconomicResource via the path that comprises the existence dependency relationship between the Increment object and its master StockFlow object and the existence dependency relationship between the StockFlow object and its master EconomicResource should relate to the same object of type EconomicResource via the path that comprises the existence dependency relationship between the Increment object and its master Ownership object and the existence dependency relationship between the Ownership object and its master EconomicResource.

As the Decrement object type is the mirror image of the Increment object type, we need a similar constraint (i.e. MPC 3) for the Decrement object type.

```
self.StockFlow.EconomicResource =
self.Ownership.EconomicResource
```

MPC 3: The MPC for propagation from Decrement to EconomicResource

Like the events of the Increment and Decrement object type can propagate to the EconomicResource object type via two distinct paths, they can also propagate to the EconomicAgent object type via two distinct paths. The events of the Increment object type can propagate to the EconomicAgent

object type through the Participation and through the Ownership object types. Consequently, these events can propagate to the same or different EconomicAgent objects. However, as we want the creation of an Increment object to change the semantics of the Participation (i.e. undirected to recipient) and Ownership (i.e. image or hadOwnership to inOwnership) associations pertaining an EconomicAgent object simultaneously, a multiple propagation constraint that enforces their propagation to one and the same EconomicAgent object is needed. This constraint is MPC 4. As the Decrement object type is the mirror image of the Increment object type, we need a similar constraint for the Decrement object type, which is MPC 5.

```
self.Participation.EconomicAgent =
self.Ownership.EconomicAgent
```

MPC 4: The MPC for propagation from Increment to EconomicAgent

```
self.Participation.EconomicAgent =
self.Ownership.EconomicAgent
```

MPC 5: The MPC for propagation from Decrement to EconomicAgent

The events of the Increment object type can propagate to the EconomicEvent object type via four different paths (i.e. one through the StockFlow object type, one through the Duality object type via the claim existence dependency relationship type, one through the Duality object type via the settle existence dependency relationship type, and one through the Participation object type), as the claim and settle existence dependency relationship types are mutually exclusive, the multiple propagation constraints have to be divided in two groups of three paths (i.e. one through the StockFlow object type, one through the Duality object type via claim or settle, and one through the Participation object type) that propagate to a single EconomicEvent object. As we want the creation of an Increment object to change the semantics of the Participation (i.e. undirected to recipient), and StockFlow (i.e. use to inflow) associations pertaining an EconomicEvent object simultaneously, MPC 6 that enforces their propagation to one and the same EconomicEvent object, also through the Duality object type (although its semantics are not affected) via the claim or settle existence dependency relationship type, is needed. As the MERLIN CASE tool does not support the exclusive choice between the claim and settle propagation paths, this functionality had to be inserted in the code semi-manually by generating two distinct models (i.e. one with propagation through settle, one with propagation through claim) and merging both codes. As the Increment and Decrement

object type are each other's mirror image, MPC 6 also applies to the Decrement object type.

```
self.StockFlow.EconomicEvent =
self.Participation.EconomicEvent AND
(self.StockFlow.EconomicEvent = self.Duality.claim XOR
self.StockFlow.EconomicEvent = self.Duality.settle)
```

MPC 6: The merged MPC for propagation from Decrement and Increment to EconomicEvent via Stockflow, Duality and Participation.

Finally, as the bottom most object type in the existence dependency graph (i.e. Figure 2), and as the equivalent of the Duality object type in the eyes of a single EconomicAgent, the DependentView object type is also subject to a number of MPCs.

MPC 7 guarantees that every DependentView object indirectly depends on a single Duality object, since the DependentView object is the equivalent of a Duality object as perceived by a single economic agent.

```
self.Increment.Duality = self.Decrement.Duality
```

MPC 7: The MPC for propagation from DependentView to Duality

MPC 8 enforces that a single `DependentView` object relates to two distinct `Participation` objects (i.e. one providing and one receiving participation).

`self.Increment.Participation ≠ self.Decrement.Participation`

MPC 8: The MPC for propagation from `DependentView` to `Participation`

Combined with MPC 8, MPC 9 enforces that a single `DependentView` object relates to two distinct `Participation` objects (i.e. one providing and one receiving participation) in the eyes of a single `EconomicAgent` object. The latter two MPCs could be rephrased as *in each dependent view, one agent should play both a provider and a recipient role in distinct participations*.

`self.Increment.Participation.EconomicAgent = self.Decrement.Participation.EconomicAgent`

MPC 9: The MPC for propagation from `DependentView` to `EconomicAgent`

Combined with MPC 9, MPC 10 enforces that a single `DependentView` object relates to two distinct participations (i.e. one providing and one receiving `Participation` object) in distinct `EconomicEvent` objects in the eyes of a single `EconomicAgent` object. This MPC could be rephrased as *in his view an economic agent participates in distinct economic events (i.e. one perceived as an increment and one perceived as a decrement) that are paired in duality*.

`self.Increment.Participation.EconomicEvent ≠ self.DecrementParticipation.EconomicEvent`

MPC 10: The MPC for propagation from `DependentView` to `EconomicEvent`

MPC 11 enforces that a single `DependentView` object relates to two distinct stock-flows (i.e. one inflow and one outflow). Where this MPC requires two distinct stock-flows, it does not require two distinct economic resources. Consequently, it is possible to model a dependent view of a duality that shows two distinct in- and outflows of the same resource. This enables the possibility to model the return of products that were not accepted by the prospect buyer.

`self.Increment.StockFlow ≠ self.Decrement.StockFlow`

MPC 11: A MPC for propagation from `DependentView` to `StockFlow`

Appendix D: Limitations of MERLIN

As the MERLIN CASE-tool does not yet support all aspects of the MERODE theory, some features of the models shown in this paper had to be realized by semi-manually modifying the generated to code. For example, the concurrent propagation of increment and decrement semantics through the `claim` and `settle` existence dependency relationships had to be implemented by manually merging (i.e. copy-paste while adding an OR constraint between them and deleting duplicate code) the automatically generated code for both scenarios (i.e. claim is increment, settle is decrement and claim is decrement and settle is increment). This concurrent propagation represents that REA does not impose any ordering between the increment and decrement semantics (i.e. the decrement can happen before the increment or the increment before the decrement).

In addition, the pre-conditions constraining the invocation of the `end_lastDependentView` MERODE event, which is an integral part of the MERODE theory was encoded manually (i.e. with a `DependentView` object counter in each `EconomicEvent` object) as such constraints are not yet supported by the CASE-tool and its code generator. Similarly, uniqueness constraints on the MERODE contracts, which are an integral part of the MERODE theory that has yet not been implemented in the CASE-tool, will need to be added manually.

Finally, a bug in the code generator erroneously generates code for operations that are disabled through MPCs. Amongst others, this led to the inaccessibility of the `conversion` state in the `Duality` state machine (i.e. Figure 5). Consequently, the double propagation of events that made the state machine skip the `conversion` state was disabled manually (i.e. by commenting out duplicate code) such that the behavior of the application became congruent with the behavior depicted in Figure 5, making the `conversion` state accessible again.

As the MERLIN CASE-tool and accompanying code generator are still under development, we are confident that the issues described above will be resolved in the near future.