Composition-based Modeling and Analysis of REA Business Patterns

Vahid R. Karimi, Donald D. Cowan
David R. Cheriton School of Computer Science
University of Waterloo, Waterloo, ON, Canada
vrkarimi@cs.uwaterloo.ca, dcowan@csg.uwaterloo.ca

1 Abstract

This paper contains a description of an approach to formalizing REA patterns, their composition, and an analysis of the resulting product. When patterns are used in the design of business models and processes, pattern composition is an essential operation because one pattern is not usually enough to define the complete set of business models and operations. In composition, we want to ensure that the results produced implement the (business) rules that are intended. Thus, there must be a clear and unambiguous specification of patterns and their composition so questions about the combination can be accurately formulated and answered. In this brief paper, we use an example to outline our formal approach to specification, composition, and analysis of REA patterns using Alloy. We present a demonstration of how the formalism can be used to show whether the resulting policies conform to our business rules.

Key Words: Alloy, (Business) Patterns, Formal Specifications and Compositions, Logic, Resource-Event-Agent (REA), REA Ontology, Resource Description Framework (RDF), Unified Modeling Language (UML), Web Ontology Language (OWL)

2 Introduction

We provide an example of REA patterns and their composition. Figure 1, similar to the ISO presentation [EDI06], shows the UML class diagram or structure of an exchange pattern for renting a car from the perspective of a car rental business. The pattern involves two resources (cash and car), two dual events (receive and rent) and two agents (customer and car rental business), and the relationships among them. The receive cash event has cash as the inflow from the customer to the car rental business while the rent event has a car as the outflow from the car rental business to the customer. Duality is indicated by the relationship between receive and rent events. The identification of receive as an increment event and cash as an inflow indicates that this pattern is being viewed from the perspective of the car rental business.

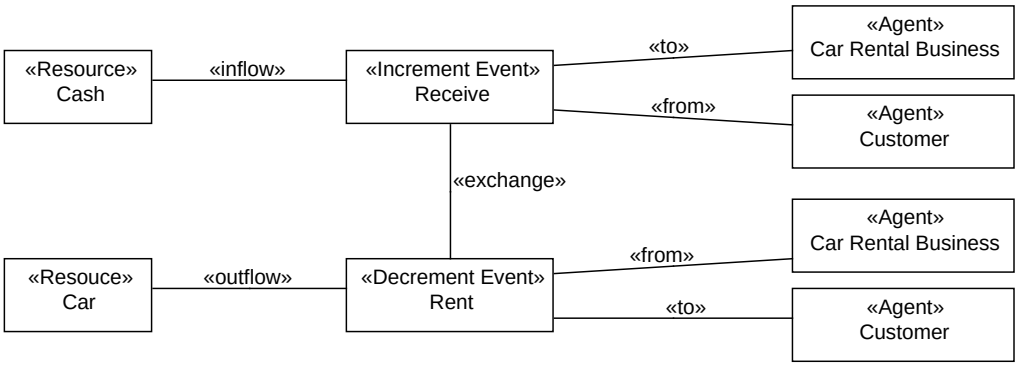

Figure 1: A Business Pattern of Renting a Car (from the Perspective of the Car Rental Business)

Figure 2, a partitioned activity diagram with car and cash object flows, shows a possible corresponding behaviour for the structure in Figure 1, namely cash is transferred from the customer to the car rental business before the car is transferred in the other direction. We show the activity diagram to indicate that we intend our work to encompass sequencing of events as well as the relationships among resources, events and agents.

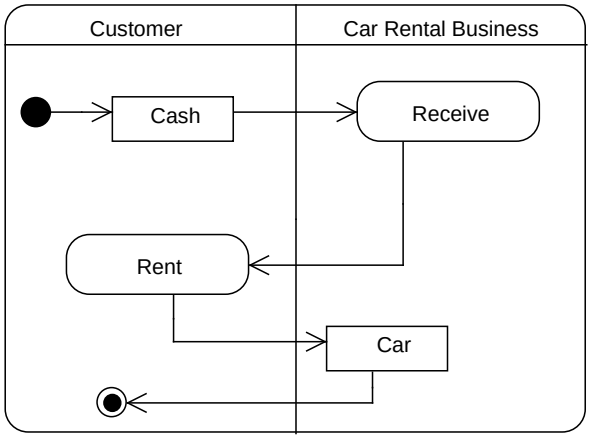

Figure 2: An Activity Diagram for Renting a Car

Although the exchange pattern is fundamental to the description of the operation of a business where cash is offered for the purchase or use of a commodity, there are often other business rules as well. For example, frequent customers may receive special discounts during certain periods; thus, resulting in other patterns being composed with the exchange. Similarly, Geerts and McCarthy [GM06] describe typification and grouping in conjunction with policies, and Hruby et al. [HKS06] also define a policy that applies to event, agent, and resource groups.

Figure 3 shows a composition where three REA patterns of exchange, group, and policy are composed to provide a 10 percent discount on economy cars on weekdays when rented by gold club customers. An exchange pattern (previously shown in Figure 1) is the top portion of Figure 3; three REA group patterns (a resource group pattern of economy cars, an agent (customer) group pattern of gold-club members, and an event group pattern of weekday receive) are represented as three vertical collaboration diagrams[1]. A policy pattern is shown as

[1]A collaboration diagram (a dashed ellipse) is the representation of patterns in UML 2.0. The pre-UML 2.0 collaboration diagrams

the horizontal collaboration diagram; the policy applies to economy cars rented on weekdays by gold-club members.

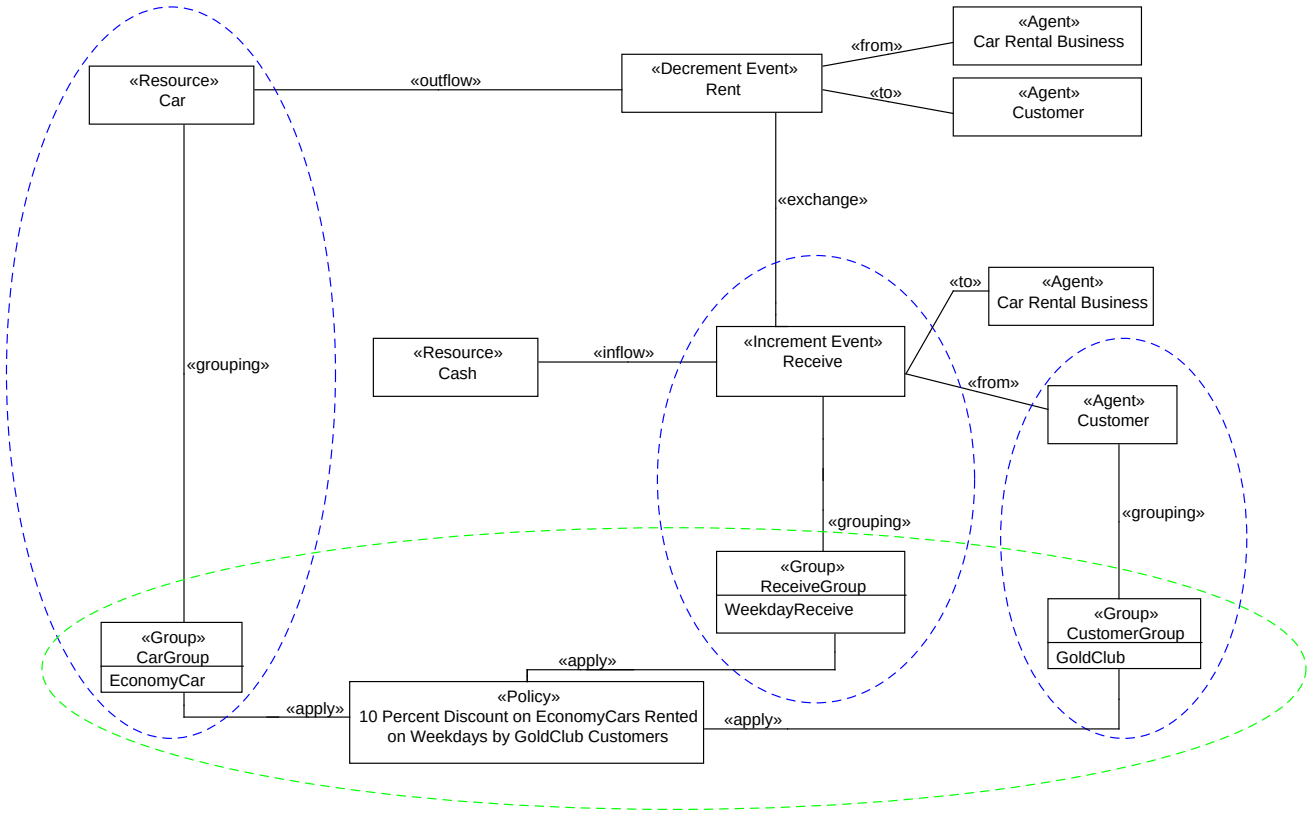

Figure 3: The Composition of REA Patterns

3 REA Pattern Formalism

This section contains an example of a formal pattern specification, composition, and reasoning about properties of patterns using Alloy [Jac00, Jac]. Alloy is a relational logic (i.e., a logic that combines the operators of the relational calculus with the quantifier of first-order logic [Jac06]).

3.1 An Example of Formal REA Pattern Specification

We first provide a brief introduction to Alloy by describing the class diagram of the exchange pattern (Figure 1) presented in Alloy in Figure 4. More details of Alloy can be found in several sources (e.g., [Jac00, Jac, Jac06]).

The first line in Figure 4 represents the relationship "inflow" between the resource "cash" and the increment event "receive." The relationship between "cash" and "receive" is indicated by an arrow (->) with the name "inflow." Similarly, the "outflow" relationship between the resource "car" and the decrement event is represented in the second line. The "exchange" relationship shows duality between the events "rent" and "receive." The car rental business must "receive" cash to "rent" a car and "rent" a car when "cash" is received. Thus, the

are now called communication diagrams [RBJ05, BME+07].

exchange relationship is the union of two simpler event relationships as represented by the plus (+) sign. The "from" and "to" relationships are also unions as both involve a simpler relationship between two events (receive and rent) and two agents (customer and car rental business). Other Alloy operators, keywords, and constraints are described when they are introduced.

```
inflow: Cash -> Receive,
outflow: Car -> Rent,
exchange: (Receive -> Rent) + (Rent -> Receive),
from: (Receive -> Customer) + (Rent -> CarRentalBusiness),
to: (Receive -> CarRentalBusiness) + (Rent -> Customer)
```

Figure 4: An Exchange Pattern in Alloy

3.2 An Example of Formal REA Pattern Composition

Relationships among patterns (e.g., design patterns) are described in numerous ways; e.g., a pattern uses, refines, or is similar to another pattern [Nob98]. It is difficult to understand these relationships clearly without a formal description.

Figure 5 shows the composition of five REA patterns (three group pattern, one exchange pattern, and one policy pattern) in Alloy (the UML version of it is provided in Figure 3). Figure 5 also includes the language element of Alloy such as *sig* (to declare sets) and organization of the structure of a model (e.g., module header and commands); the diagram created by the specification of Figure 5 is shown in Figure 6.

In Figure 5, Exchange is declared *sig* (*sig* stands for signature and declares a set; in addition, a *sig* can introduce relationships). Therefore, Exchange is declared as a set in which all the relationships (inflow, outflow, to, from, exchange) are specified. In addition, the three fundamental building blocks of REA (Resource, Event, and Agent) are shown as *abstract sig* (an *abstract sig* does not have any other elements except those belonging to its extensions.) Receive and Rent extend Event (the keyword *extends* creates subsets which are mutually disjoint); the same explanation holds for the use of *extends* with Car, Cash, CarRentalBusiness, and Customer. Furthermore, PolicyA is a subset of Policy and applies, using union operators (+), to EconomyCar, GoldClubCustomer, and WeekdayReceive.

3.3 An Example of Reasoning about Some Properties of the Resulting Combinations

We use Alloy and its analyzer to reason about properties of patterns when they are combined to determine whether the business rules that were intended are actually implemented. The Alloy Analyzer is a compiler that translates a problem into a boolean formula handed to a SAT Solver. The Alloy Analyzer does a reverse translation of the SAT Solver's solution into Alloy's language [Jac06, Jac].

```
module models/MyExamples/exchange2
abstract sig Resource, Agent, Event{}
sig Receive extends Event{}
sig Rent extends Event{}
sig CarRentalBusiness extends Agent{}
sig Customer extends Agent{}
sig Car extends Resource{}
sig Cash extends Resource{}
sig Exchange {inflow: Cash -> Receive,
              outflow: Car -> Rent,
              exchange: (Receive -> Rent) + (Rent -> Receive),
              from: (Receive -> Customer) + (Rent -> CarRentalBusiness),
              to: (Receive -> CarRentalBusiness) + (Rent -> Customer)}
sig EconomyCar{grouping: Car}
sig GoldClubCustomer{grouping: Customer}
sig WeekdayReceive{grouping: Receive}
sig Policy{}
sig PolicyA extends Policy{apply: EconomyCar + GoldClubCustomer +
                                  WeekdayReceive}
pred showPolicy[p:Policy]{}
run showPolicy}
```

Figure 5: A Composition of REA Patterns

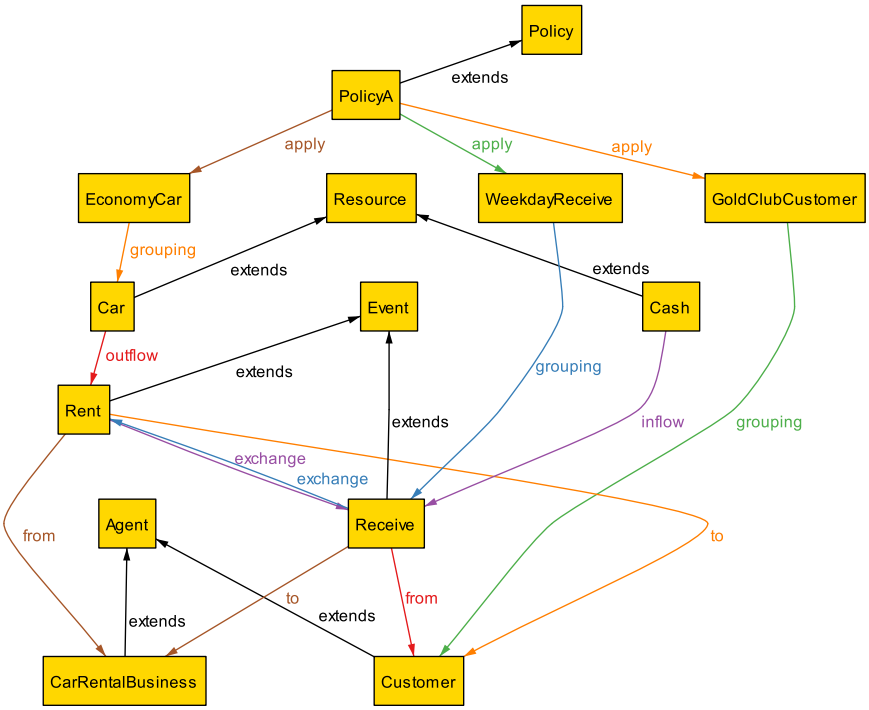

Figure 6: A Composition of Three Patterns using Alloy

Policies constantly change, and even a single seemingly trivial policy change can create inconsistencies. For example, McWhorter [McW08] examines a simple change of an existing policy (i.e., "no rental policy can be longer than 30 days") so that the same policy now allows car rentals for a period longer than a month. He examines how this simple policy change affects other existing policies such as the policy which states cars must be examined for damage monthly or another policy that requires the maintenance of vehicles every month.

As an example, we show how adding a new policy can create inconsistencies. Assume that there is a new policy that applies to economy cars rented during week-days and to all customers (not just gold-club members in the previous example).

Now, assume that another policy is added such that gold-club members get a different discount:

```
sig PolicyB extends Policy{apply: GoldClubCustomer}
```

Assume that each customer can have at most one discount (a new policy). To express this constraint, we use Alloy's assertion (the keyword "assert"). An assertion in Alloy is a constraint intended either to follow from the facts expressed in a model or to be self-contained describing essential properties [Jac06]. The following assertion, the latter usage of assert described above, expresses the desired property of not having a customer (c) who can have two different discount policies (p, p'), shown by the conjunction of two relationships: $p->c$ and p' -> c. The keyword "check" examines the assertion.

```
assert onlyOneDiscount{all c:Customer, p,p':Policy| no (p -> c & p'-> c)}
check onlyOneDiscount
```

Checking the assertion that each customer can get at most one discount creates a counterexample:

```
Counterexample found. Assertion is invalid.
```

4 Related Work

REA Formalism: McCarthy and Geerts [GM00] provided predicate-like definitions of REA elements of an exchange pattern and used Prolog to implement an instantiation of their definitions. They also used Prolog queries to answer questions such as "which events may lead to spending?" Bialecki [Bia01] also described an early work on REA formalism and provided a RDF schema of REA ontology. Gailly and Poels [GP07] mapped a UML class diagram of REA to the Web Ontology Language (OWL) formalism and stated that the consistency of concept specifications could be verified by the use of this formalism and a Description Logic reasoner.

5 Conclusion

We expect that our approach to formal specifications, compositions, and reasoning about properties of REA patterns and their building blocks will support a clear correct description of business models and processes.

We presented one example of the composition and analysis of REA patterns and intend to investigate other structural and behavioural properties and their formal specifications. We also plan to examine the specification of some behavioural properties in a language such as one based on a process calculus.

References

[Bia01] Andrzej Bialecki. REA ontology. Available at http://www.getopt.org/ecimf/contrib/onto/REA/, 2001.

[BME+07] Grady Booch, Robert Maksimchuk, Michael Engel, Bobbi Young, Jim Conallen, and Kelli Houston. *Object-oriented analysis and design with applications*. Addison-Wesley, 3 edition, 2007.

[EDI06] EDI Management, Project Editor William McCarthy. *ISO/IEC FDIS 15944-4:2006 Information Technology - Business Operational View – Part 4: Business Transaction Scenarios – Accounting and Economy Ontology*, October 2006.

[GP07] Frederik Gailly and Geert Poels. Ontology-driven business modelling: Improving the conceptual representation of the REA ontology. In *Lecture Notes in Computer Science*, volume 4081, pages 407–422, November 2007.

[GM00] Guido Geerts and William McCarthy. Augmented intensional reasoning in knowledge-based accounting systems. *Journal of Information Systems*, 14(2):127–150, 2000.

[GM06] Guido Geerts and William McCarthy. Policy-level specifications in REA enterprise information systems. *Journal of Information Systems*, 20(2):37–63, 2006.

[HKS06] Pavel Hruby, Jesper Kiehn, and Christian Vibe Scheller. *Model-Driven Design Using Business Patterns*. Springer, 2006.

[Jac] Daniel Jackson. http://alloy.mit.edu.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 6-10, 2000.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[McW08] Neal McWhorter. Business architecture and accountability to business intent. *Cutter IT Journal, The Journal of Information Technology Management*, 21(3):11, 2008.

[Nob98] James Noble. Classifying relationships between object-oriented design patterns. In *1998 Australian Software Engineering Conference (ASWEC)*, pages 98 – 107, Adelaide, Australia, November 1998.

[RBJ05] James Rumbaugh, Grady Booch, and Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2 edition, 2005.