İş Süreci Kurallarının JsonLogic ile Geliştirilmesi için bir Modelleme Ortamı

Katira Soleymanzadeh[1], Yiğit Bul[2], Sedat Kulduk[2], Sarper Bağcı[2], Geylani Kardaş[1]

[1] Ege Üniversitesi Uluslararası Bilgisayar Enstitüsü, 35100 Bornova, İzmir, Türkiye
katirasole@gmail.com, geylani.kardas@ege.edu.tr

[2] Hermes İnternet İletişim Yazılım San. ve Tic. A.Ş., Pasaport, İzmir, Türkiye
{yigit.bul, sedat.kulduk, sarper.bagci}@hermesiletisim.net

Özet. İş Süreci Yönetimi (BPM) yazılımlarında iş süreçleri ve iş akışlarına ait işletim kurallarının mantık tabanlı oluşturulması için JSON temelli JsonLogic yapısının BPM yazılımlarında yakın zamanda kullanılmaya başlandığı gözlemlenmektedir. Her ne kadar JsonLogic ile iş süreci kurallarının oluşturulması BPM'ler içerisinde bu kuralların çeşitli veritabanlarında saklanmasına ve paylaşımına imkan verse de bu kuralların metinsel sözdiziminin yazılım geliştiricilerin kullandığı genel amaçlı programlama dillerinden farklı olması, geliştiricilerin özellikle komplike iş kurallarının yazımı sırasında zorlanmalarına ve kural hazırlama sürecinin hem uzamasına hem de hatalara daha açık bir hale gelmesine neden olmaktadır. İş süreci kurallarının JsonLogic ile geliştirilmesinde karşılaşılan bu zorlukları minimize etmek amacıyla, bu çalışmada JsonLogic yapılarının bir model-güdümlü mimari kapsamında görsel modellenmesini ve sonra bu modellerden JsonLogic kurallarının otomatik oluşturulmasını sağlayacak bir alana-özgü modelleme dili ve bunu destekleyen bir araç tanıtılmaktadır. Sunulan modelleme ortamının örnek bir durum çalışması üzerinden gerçekleştirilen değerlendirmesinde, sadece JSON ile modelleme imkanı sunan ve JsonLogic'i desteklemeyen mevcut bir araca göre ilgili iş kurallarının daha az görsel bileşen ile daha kolay bir şekilde oluşturulabildiği gözlenmiştir.

Anahtar Kelimeler: JsonLogic, Model-güdümlü Mimari, Alana-özgü Modelleme Dili, İş Süreci Yönetimi, İş Süreci Kuralı.

A Modeling Environment for the Development of Business Process Rules with JsonLogic

Abstract. Use of JSON-based JsonLogic structures has been recently emerged during the construction of the rules required in the business processes and business workflows inside the Business Process Management (BPM) software. Although, creating business process rules with JsonLogic enables both storing

these rules in databases and sharing them between front-end and back-end systems, the developers may encounter difficulties during the preparation of complicated business rules with JsonLogic due to its intricate textual syntax which is too different from the well-known general-purpose programming languages. This unfamiliar way of rule creation may also lead to a time-consuming and error-prone development process. In order to eliminate these deficiencies and facilitate the business rule creation with JsonLogic, we introduce a domain-specific modelling language and its supporting tool for the visual modelling and automatic generation of JsonLogic rule structures within a model-driven architecture. The evaluation, performed inside a case study, showed that the proposed modelling environment is capable of the construction of business rules with far less number of components comparing with modelling the same rules in an existing tool for JSON.

Keywords: JsonLogic, Model-driven Architecture, Domain-specific Modeling Language, Business Process Management, Business Process Rules.

1 Giriş

İş Süreci Yönetimi (ing. Bussiness Process Management) (BPM) iş süreçlerinin tasarımı, hayata geçirilmesi ve yönetilmesi için çeşitli yöntem ve araçların kullanımını içermektedir. BPM'ler birçok alanda yaygın olarak kullanılan iş akışı yönetimi sistem ve yaklaşımlarının birer uzantısı olarak düşünülebilirler [1]. BPM yazılımlarında iş süreçleri ve iş akışlarına ait işletim kuralları çoğunlukla mantık tabanlı olarak kurgulanmakta ve yönetilmektedir [2]. Görev ve kaynakların durumunun belirlenmesi, bunların aktörlere atanması, işletimsel kısıtların belirtilmesi, süreçlerin yönlendirilmesi, olayların tetiklenmesi gibi bileşenler için kuralların oluşturulmasında tümevarım / tümden gelimli veya şartlı akıl yürütmeyi sağlayacak mantık kurgularına ihtiyaç vardır.

Karmaşık iş süreci kurallarının mantık tabanlı oluşturulması için JavaScript Nesne Gösterimi (ing. JavaScript Object Notation) (JSON) [3] mantığı ve bunun üzerine geliştirilen JsonLogic [4] yapısının profesyonel BPM yazılımlarında yakın zamanda kullanılmaya başlandığı (örneğin IBM BPM [5], Camunda [6]) gözlemlenmektedir. Her ne kadar JsonLogic ile iş süreci kurallarının oluşturulması BPM'ler içerisinde bu kuralların çeşitli veritabanlarında saklanmasına ve paylaşımına imkan verse de bu kuralların metinsel sözdiziminin yazılım geliştiricilerin kullandığı genel amaçlı programlama dillerinden farklı olması, geliştiricilerin özellikle komplike iş kurallarının yazımı sırasında zorlanmalarına ve kural hazırlama sürecinin hem uzamasına hem de hatalara daha açık bir hale gelmesine neden olmaktadır. İş süreci kurallarının JsonLogic ile geliştirilmesinde karşılaşılan bu zorlukları minimize etmek amacıyla, bu çalışmada JsonLogic yapılarının bir model-güdümlü mimari kapsamında görsel modellenmesini ve sonra bu modellerden JsonLogic kurallarının otomatik oluşturulmasını sağlayacak bir alana-özgü modelleme dili ve bunu destekleyen bir araç tanıtılmaktadır.

İş süreçlerinin ve iş akışlarının koreografisi ve yönetilmesinde model-güdümlü mühendislik yaklaşımlarının uzun bir süreden beri izlendiği görülmektedir [7].

BPM'ler için çeşitli model-güdümlü mimari uygulamaları ve modelleme dili geliştirme çalışmaları da (örneğin [8-12]) ilgili araştırma alanında güncelliğini korumaktadır. Ancak mevcut çalışmalarda iş süreci ve kurallarına ait mantığın JsonLogic ile modellenmesi ve üretilmesi göz önüne alınmamaktadır. Bildiğimiz kadarıyla ilk kez bu bildiride anlatılan çalışma ile JsonLogic yapılarının modellenmesi için bir üstmodel geliştirilmiştir ve bu üstmodeli temel alan bir görsel modelleme ortamı yazılım geliştircilere sağlanarak JsonLogic iş kurallarının otomatik elde edilmesi mümkün hale gelmektedir.

Bildirinin 2. bölümünde kısaca JsonLogic hakkında bilgi verilmiştir. Geliştirilen JsonLogic üstmodeli 3. bölümde tanıtılmaktadır. Üstmodele dayalı olarak iş süreci kurallarının görsel modellenmesini sağlayan sözdizim ve işletimsel semantik sırasıyla 4. ve 5. bölümlerde anlatılmıştır. Geliştirilen dilin ve modelleme aracının kullanımını örnekleyen bir durum çalışması 6. bölümde yer almaktadır. 5. bölümde, ilgili literatürdeki önceki çalışmalar anlatılmış; mevcut çalışmalara göre farklar ve katkılar belirtilmiştir. Son bölümde çalışmadan elde edilen sonuçlar ve ileriye yönelik çalışma hedefleri yer almaktadır.

2 JsonLogic

JSON, ilkel ve yapısal veri türlerine sahip, metin tabanlı ve dilden bağımsız bir veri değişim biçimidir [3]. Anahtar-değer çifti şeklinde tanımlanan JSON veri biçimi, diğer veri tiplerine göre (örneğin XML verisi) daha az yer kaplamaktadır. JsonLogic ise JSON mantığı üzerinde tanımlanan bir yapıdır ve karmaşık mantık kurallarını inşa etmek, onları birer JSON verisi olarak serileştirmek, uygulamaların ön-uç (ing. front-end) ve arka-uçları (ing. back-end) arasında paylaşmak ve veritabanlarında saklanmak için kullanılmaktadır [4]. Her JsonLogic kuralı bir JSON nesnesidir ve operatör-veri çifti biçiminde tanımlanmaktadır. Operatör bir JsonLogic yapısında anahtar konumunda iken bir veya bir dizi argüman değer konumunda yer almaktadır. Her bir argümanın kendisi de bir mantık kuralı olabilir ve böylece karmaşık kuralları JsonLogic ile tanımlamak mümkün olmaktadır. Her ne kadar tam bir programlama dili olmasa da JsonLogic özellikle kuralların birer veri şeklinde saklanmasına ve böylece kullanıcı etkileşimi ile dinamik olarak oluşturulabilmelerine imkan vermesi nedeniyle iş süreçlerinin kural tabanlı oluşturulmasında güçlü bir alternatif olmaktadır.

Aşağıda basit bir JsonLogic iş kuralı örneği verilmiştir. Kuralda eğer bir kişinin e-posta adresi boş (ing. null) değilse ve bu kişeye gönderilen e-posta sayısı üçten azsa (yani kuralın işletim sonucu mantıksal doğru (ing. true) olacaksa), ilgili kişiye yeni bir e-posta gönderilecektir.

JsonLogic kuralı: {"and" : [{ "!=" : [{"var" : "email_address"},null] },{ "<" : [{"var" : "number_of_sent_mail"},3] }] }

JsonLogic kural ayrıştırıcıları (ing. parser) [4] yazılan kuralların otomatik olarak Python, JavaScript, PHP ve Ruby'ye dönüştürülmesine ve bu ortamlarda

çalıştırılmalarına olanak sağlamaktadır. Örneğin yukarıda JsonLogic kullanılarak hazırlanan kuralın ayrıştırıcı kullanılarak elde edilen Python'daki eşdeğer kodu şu şekildedir:

Python kodu: ((email_address != null) and (number_of_sent_mail < 3))

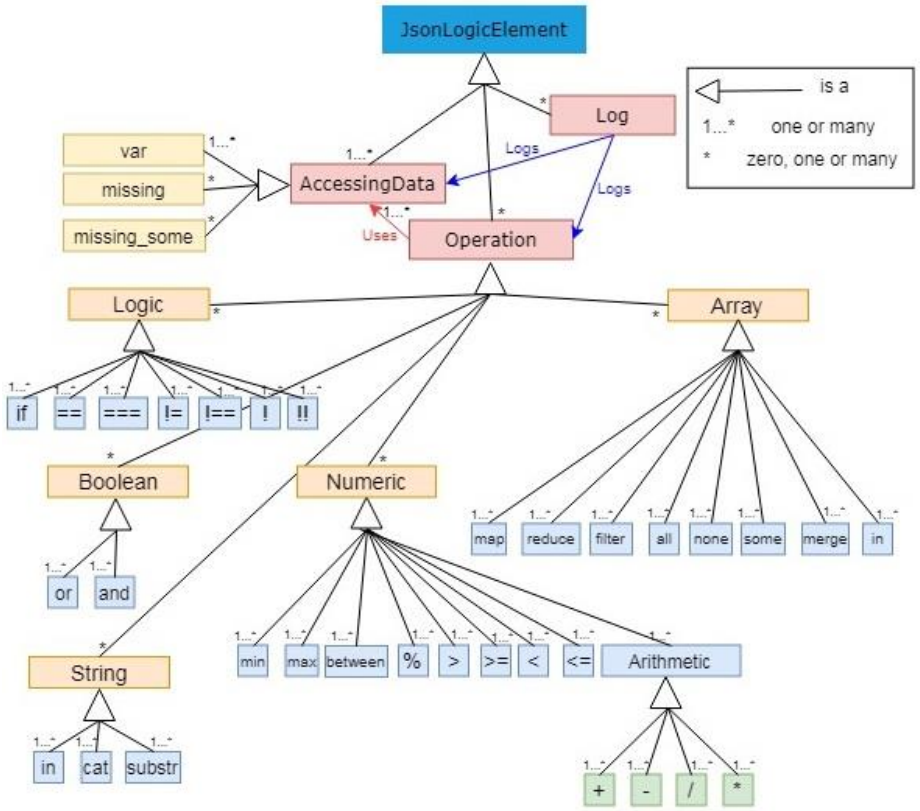
3 JsonLogic Yapıları için bir Üstmodel

JsonLogic mantık kurallarının model-güdümlü olarak geliştirilmesini sağlamak amacıyla JsonLogic modellerinin oluşturulabileceği bir üstmodel (ing. metamodel) bu çalışmada hazırlanmıştır. Üstmodelin türetilmesi için JsonLogic yapıları [4] göz önüne alınmış; ve bu yapıların modellenmesini sağlayacak üstvarlıklar (ing. meta-entity) ve bunların ilişkileri bu üstmodel içerisinde tanımlanmıştır. Geliştirilen üstmodel Nesne Yönetim Grubu (ing. Object Management Group) (OMG) Üst-varlık Ortamı (ing. MetaObject Facility) (MOF) [13] uyumludur. Böylece bu üstmodele göre oluşturulan JsonLogic iş kuralı modelleri OMG Model-güdümlü Mimarisi (ing. Model-driven Architecture) (MDA) [14] kapsamında XML ile serileştirilebilir ve Eclipse Modelleme Çerçevesi [15] gibi üst-üstmodelini MOF'un oluşturduğu modelleme ortamlarında kullanılabilir. Şekil 1'de söz konusu üstmodel resmedilmiştir. Anlam kaymalarını önlemek amacıyla bildiride üstmodeldeki varlıklar ve ilişkiler İngilizce orjinal isimleri kullanılarak anlatılmaktadır ve metinde eğik (italik) olarak yazılmışlardır.

Her bir *JsonLogic Element* temelde bir *AccessingData*, *Operation* veya *Log* olabilir. *var*, *missing* ve *missing_some* tiplerinden birine sahip olabilecek bir *AccesingData* elemanı, kullanıcı tarafından girilen verinin işlenmesini ve sonucun döndürülmesini sağlar. Genellikle nesne formatında olan her veri, özel ismi olan bir *var* kavramı ile tanımlanır. *var* kavramı kullanıcı tarafından sağlanan veriyi tanımlar. *missing*, JsonLogic modelinde tanımlanan bir veri dizisi içinde bulunmayan elemanları belirlemede kullanılır. *missing_some* girdi olarak verilen minimum sayı kadar verinin bir anahtar dizisi içerisinde olup olmadığını sorgulamayı sağlamaktadır. Tablo 1'de AccesingData çeşitlerinin JsonLogic kural cümlelerinin oluşturulmasında kullanımı ve örnek veriler üzerinde bu kuralların işletilmesi sonuçları listelenmiştir.

Üstmodeldeki *Operation* üstvarlığının örnekleri bir JsonLogic modelindeki ana elemanları temsil etmektedir ve JsonLogic kuralları *Operation* çeşitleri kullanılarak oluşturulmaktadır. Üstmodelde bu varlığın *Logic*, *Boolean*, *Numeric*, *Array* ve *String* isimli beş alt tipi bulunmaktadır. Bu tiplerin de yine üstmodelde üst sınıf – alt sınıf kurgusu ile oluşturulan çeşitli alt tipleri vardır.

Boolean üst-varlığı adından anlaşılacağı üzere mantıksal doğru ve yanlış testleri için modelde yer alır; *or* ve *and* alt-sınıflarına sahiptir. *Logic*'in tiplerini modelde *if*, *==*, *===*, *!=*, *!==*, *!* ve *!!* varlıkları temsil eder. *if* klasik programlama dillerinde olduğu gibi bir koşulu ve bu koşulun doğru veya yanlış olması durumunda karşılık gelen işlemleri JsonLogic'te göstermek için kullanılır. *Logic*'in geriye kalan diğer alt sınıfları ise girdi yapılan veriler arasındaki eşitlik ya da eşitsizlik durumlarını kontrol etmeyi sağlar.



Şekil 1. JsonLogic yapıları için bir üstmodel

Tablo 1. *AccessingData* çeşitleri ile hazırlanan bazı JsonLogic kural örnekleri

JsonLogic Üstvarlığı	**Örnek Kural**	**Örnek Veri**	**İşletim Sonucu**
var	{"var":"name"}	{"name":"Ali","age":32}	Ali
missing	{"missing":["name", "age", "count"]}	{"name":"Ali"}	["age", "count"]
missing_some	{"missing_some":[3, ["name", "age", "count", "Tel"]]}	{"name":"Ali", "age":32}	[]

Numeric işlem türleri (*min*, *max*, *between*, vb.*)* girdiler arasındaki büyüklük, küçüklük gibi durumların kontrolünü ve bir veri kümesindeki en küçük veya en büyük verinin belirlenmesini sağlayan yapıların JsonLogic'te oluşturulmasını sağlar. Veriler üzerindeki cebirsel işlemler *Numeric*'in diğer bir alt sınıfı olan *Arithmetic* tipindeki üstvarlıklar ile modelde temsil edilmiştir. *map*, *reduce*, *merge* gibi *Array* işlemleri diziler şeklinde ifade edilen veri kümleri üzerine uygulanır. Örneğin *map* bir dizi içerisindeki her elemana bir işlemin (toplama, çarpma, vb.) uygulanmasını sağlarken

reduce bir dizideki tüm elemanları tek bir elemana indirger. *merge* ise birden fazla diziyi birleştirir.

Üstmodelde mantık kurallarında yer alan kelimeler üzerindeki işlemleri *String* varlığının alt sınıfları temsil etmektedir. Her bir *in* varlığı tanmlanan bir kuralda veri konumunda olan ilk argümanın "String" tipinde olan ikinci argümanın içinde olup olmamasını test eder. Argümanların birleştirilmesi *cat* işlemi ile sağlanır. "String" parçalarını elde etmek için *substr* işlemleri uygulanır. Tablo 2'de bazı Operation çeşitlerinin JsonLogic iş kuralı cümlelerinin oluşturulmasında kullanımı ve örnek veriler üzerinde bu kuralların işletilmesi sonuçları listelenmiştir.

Tablo 2. *Operation* çeşitleri ile hazırlanan bazı JsonLogic kural örnekleri

JsonLogic Üstvarlığı ve Alt sınıfı		**Örnek Kural**	**Örnek Veri**	**İşletim Sonucu**
Logic	if	{"if" : [{"==" : [{"var": "count"}, 3]}, send email, send sms]}	{"conut":3}	send sms
	==, ===, !=, !==, !, !!	{"!=":[{"var":"name"},"Ali"]}	{"name": Ali}	true
Boolean	or ve and	{"and":[{"==":[{"var":"count},3]}, {"==" : [{"var": "name"}, Ali]}, {"<": [{"var": "age"}, 30]}]}	{"conut":3, "name":Ali,"age" :32}	false
Numeric	>, >=, < ve <=	{"<=" : [{"var": "age"}, 30]}	{"age":32}	false
	between	{"<": [30, [{"var": "age"}, 30], 40]} // 30<age<40	{"age":32}	true
	max ve min	{"min":[-1,2,-3]}	null	-3
	+ , -, *, / ve %	{"+":[2,3,2,3,2]}	null	12
Array	all, some, none	{"none":[[3,2,1],{">":[1,{"var":" "}, 5]}]}	null	true
	merge	{"merge":[{"var":"count"},{"var " : "name"},{"var" : "age"}]}	{"count":3, "name":"Ali","ag e":32}	[3,"Ali",32]
	in	{"in":["Ali",["Ahmet", "Ayşe", "Ali", "Gül"]]}	null	true
String	in	{"in":["bahar","baharmevsimi"]}	null	true
	cat	{"cat":["send",{"var":"type"}]}	{"type":"email"}	"send email"
	substr	{"substr": [" Email sent by Ali", 15]}	null	"Ali"

Son olarak, geliştirilen üstmodelde yer alan *Log* üstvarlığı örnekleri (ing. instance) özellikle karmaşık JsonLogic kurallarının kontrolü ve hata ayıklamasında çıktıların

görüntülenmesi amacıyla kullanılmaktadır. Mantıksal doğru / yanlış, sayı veya kelime tiplerinde çıktılar üretebilir.

4 Görsel Somut Sözdizimin Oluşturulması

Bir önceki bölümde tanıtılan üstmodele uygun olarak yazılım geliştiricilerin JsonLogic tabanlı olarak iş süreci kurallarını görsel bir şekilde oluşturması için bu çalışmada aynı zamanda görsel bir somut sözdizim geliştirilmiştir.

Alana-özgü diller (ing. Domain-specific language) (DSL) [16] oluşturulurken genellikle bir üstmodele bağlı dil soyut sözdizimini türetme ve devamında bu sözdizimdeki kavramlar ve bunların örnek modeller üzerindeki temsilleri arasında bir eşleme sağlayan bir somut sözdizimin geliştirilmesi süreçleri izlenmektedir. Bizim de bu çalışmada amacımız JsonLogic ile iş süreci kurallarının geliştirilmesinde kullanılabilecek bir DSL'i oluşturmaktır. Geliştirilen somut sözdizim JsonLogic öğelerinin kodlanması yerine görsel olarak modellenmesini sağladığından ortaya konan dilin DSL'den ziyade bir alana-özgü modelleme dili (ing. domain-specific modeling language) (DSML) ([17]) olduğunu belirtmekte yarar vardır.

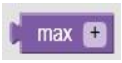
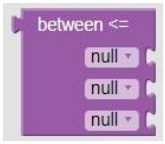
3. bölümde verilen üstmodel OMG MDA'ine [14] göre platforma-özel modelleme seviyesinde bulunmaktadır ve JsonLogic için geliştirdiğimiz DSML'in soyut sözdizimini oluşturmaktadır. DSML'in kullanılmasını sağlayan somut sözdizim MetaEdit+ [18] veya Eclipse tabanlı GMF [19], Sirius [20] gibi model-güdümlü yazılım geliştirme ortamları kullanılarak oluşturulabilir. Daha önce bu ortamları kullanarak gerçekleştirilen DSL / DSML geliştirme çalışmalarımızdan (örneğin [21-23]) MOF uyumlu soyut sözdizimlerin bu araçların bünyesine kolaylıkla dahil edilebilindiğini ve aynı entegre ortamlar içerisinde DSML semantiklerinin de tanımlanabildiğini gözlemledik. Öte yandan modellemenin web tabanlı ve çevrimiçi yapılması ve yazılım modellerine birden çok kullanıcı tarafından müdahele gibi ihtiyaçlar düşünüldüğünde sözü edilen bu araçlar çoğu zaman yetersiz kalmaktadır ve kullanımları karmaşıklaşmaktadır. Tüm bu ihtiyaçlar göz önünde bulundurularak JsonLogic DSML'inin somut sözdizimi bu çalışmada Blockly [24] görsel blok programlama ortamı üzerine inşa edilmiştir. Blocly ([24-25]), Scratch [26], Snap! [27] gibi programlama kütüphaneleri ve araçları son yıllarda giderek popülerleşen blok tabanlı görsel programlamanın önemli temsilcileridir.

Blockly'de yazılım bileşenleri birbirine bağlanabilen görsel bloklar şeklinde tasarlanabilir. Web tabanlı ortamı içerisinde bileşenler bir paletten sürükle-bırak tekniği ile modelleme ortamına taşınabilir ve birer blok olarak modellenebilir. Bloklar halinde oluşturulan bu yazılım bileşenleri Blockly'nin destek verdiği Python, JavaScript, PHP ve Lua gibi dillerde implementasyona çevrilebilir [28]. Ayrıca JSON yapısını üretmek üzere bir Blockly kütüphanesi de bulunmaktadır [29]. Tüm bu özellikler Blockly'nin bu çalışmada JsonLogic DSML'inin görsel modelleme ortamını geliştirmek için kullanılmasına neden olmuştur.

Somut sözdizimin oluşturulması için bir önceki bölümde verilen JsonLogic üstmodeli varlıklarını ve ilişkilerini gösterecek görsel notasyonlar belirlenmiştir. Bir kısım üstmodel kavramı için doğrudan bir görsel notasyon (blok sembolü)

belirlenirken geriye kalan kavramlar, blok sembolü türetilmiş kavramların kapsayacağı şekilde model ortamına dahil edilmektedir. Tablo 3'te JsonLogic DSML'inin görsel somut sözdizimi için bu çalışma kapsamında belirlenen notasyonların bir kısmı görülmektedir.

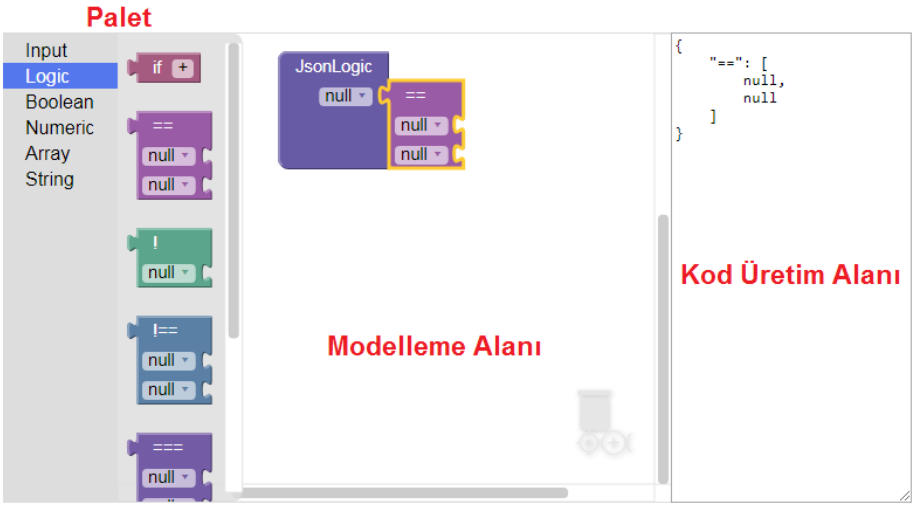
Tablo 3. JsonLogic DSML'i somut sözdiziminin bazı kavramları ve gösterimleri

Kavram	Gösterim	Kavram	Gösterim	Kavram	Gösterim
if		and		less	
==		max		between	
!!		min		var	

Şekil 2'de JsonLogic tabanlı iş süreci kurallarının model-güdümlü geliştirilmesini sağlayan web tabanlı aracımıza ait bir ekran görüntüsü verilmiştir. Blockly ortamı üzerinde geliştirilen araçta sol tarafta JsonLogic somut sözdizimi için belirlediğimiz görsel notasyonların listesi bir palet içerisinde bulunmaktadır. Palet içerisinde benzer anlama sahip modelleme bileşenleri aynı kategori içerisine alınmıştır. Bir yazılım geliştirici bu paletten JsonLogic model elemanlarını sürükle-bırak yöntemi ile araçta orta kısımda bulunan modelleme alanına taşıyarak bu elemanlardan istenilen sayıda örnekler (ing. instance) oluşturabilir.

Geliştirilen araç üzerinde JsonLogic iş kuralı modelleri oluşturulurken bir takım statik semantik kontrolleri de otomatik olarak gerçekleştirilmektedir. Bölüm 3'te verilen JsonLogic üstmodeli varlıkları ve ilişkileri için tanımlı olan bazı kısıtlara göre kontroller gerçekleştirilmekte ve somut sözdizim gösterimi farklılaşmaktadır. Örneğin, "if" kavramına ait görsel Blockly notasyonunun girişi birden fazla olabileceği için "+" operatörü kullanılarak istenen sayıda girdi eklenebilir. Ancak kıyaslama kavramlarının (örneğin "==, !, !!") girdisi maksimum iki ile sınırlı olduğu için modelleme aracı bir geliştiricinin sadece ikili-giriş blokla gösterim yapmasına izin vermektedir. Modelleme sırasında girdi tipi kontrolleri de ("var, string, number, array, true ve false" gibi) araç tarafından yapılmaktadır. Örneğin, "var" kavramı bir girişli bloktur ve kullanıcıdan gelen veriyi tanımlar. Diğer girdiler, sonlu bir bloktan

tanımlanır ve hiç giriş kısmı yoktur. Bir diğer statik semantik kontrol örneği "between" bileşeni için verilebilir. "between"'in üstmodeldeki tanımlamasına göre üçten fazla girdisi olamayacağı için araç bu bileşenin örneklerinin maksimum üç girişli blok ile modele dahil edilmesine izin verir. Ayrıca "between" kavramının girdileri sadece "number" ve "var" olabilir ve modelleme alanında blok tasarımları yapılırken "between" örnekleri için sadece bu tiplerde girdilerin tanımlanmasına izin verilir. Söz konusu bu kısıt kontrollerinin modelleme sırasında otomatik olarak gerçekleştirilmesi geliştiricilerin daha doğru JsonLogic kurallarını elde etmelerine yardımcı olmaktadır.



Şekil 2. JsonLogic modelleme ortamından bir görünüm

5 JsonLogic Kurallarının Otomatik Üretilmesi

Bir DSML ile modellenen yazılımların işletilebilir kodlarının elde edilmesi için bir dizi dönüşüm kuralına dayalı bir semantik kurgulanabilir [30]. Bu çalışmada JsonLogic DSML'i ile hazırlanan modellerden JsonLogic yapılarının ve iş süreci kurallarının otomatik üretilmesi için bir işletimsel semantik de tanımlanmıştır. Hazırlanan iş kuralı modelleri bloklarla modellendikten sonra bu modeller üzerinde işletilen modelden metne kurallar ilgili modellere karşılık gelen JsonLogic'in otomatik üretilmesini sağlarlar. Bu amaçla bir dizi dönüşüm kuralını JavaScript dilini kullanarak hazırladık. Örneğin aşağıdaki JavaScript dönüşüm kodu geliştirilen bir Blockly JsonLogic modeli üzerindeki her bir "if" örneği için karşılık gelen JsonLogic yapısını üretmektedir:

```
1.  Blockly.JSON['if_logic'] = function(block) {
2.      var if_logic = {};
3.      var array = [];
4.      for(var i = 0; i<block.length; i++) {
5.          array[i] = this.BlockToJson( block.getInputTargetBlock('element_'+i));
6.      if_logic["if"] = array;
7.      }
8.  return if_logic;
9.  };
```

İlgili kod parçasında bir döngü içerisinde bir "if" Blockly bloğununun uzunluğunun sonuna kadar (satır 4), bloğun girdileri alınır ve bir dizi içine bunlar kaydedilir (satır 5). Üretilen dizi bir sözlük içine alınır (satır 6) ve ilgili JsonLogic yapısı oluşturulur.

Bir diğer işletimsel semantik kodu örneği "var" model elemanlarının JsonLogic karşılıklarının oluşturulması için verilebilir. Aşağıdaki JavaScript kodu bir "var" bloğunun girdisini belirler ve karşılık gelen JsonLogic yapısını oluşturur.

```
1.  Blockly.JSON['var'] = function(block) {
2.      var var_op = {};
3.      var_op["var"] = this.generalBlockToObj( block.getInputTargetBlock('json0'));
4.  return var_op;
5.  };
```

Bir yazılım geliştiricinin JsonLogic DSML'ini kullanarak hazırladığı iş kuralı modellerinden JsonLogic kodlarını üretmek için bu JavaScript kodlarını, bir başka deyişle DSML'imizin işletimsel semantiğinin çalışma mekanizmasını bilmesine gerek yoktur. Geliştirici iş kurallarına ait görsel modelini hazırlaması yeterli olmaktadır. Model hazırladıktan sonra aracın Şekil 2'de gösterilen sağ tarafında bu modele ait otomatik üretilen JsonLogic geliştiriciye gösterilmektedir.

6 Durum Çalışması

Hazırlanan JsonLogic DSML'inin ve bunu destekleyen modelleme aracının kullanımını örneklemek amacıyla bu bölümde Hermes İletişim [31] firması bünyesinde geliştirilmekte olan Avukat İş Takip Asistanı adlı iş süreci yönetim yazılımının içerdiği kurallardan birinin model-güdümlü geliştirilmesi göz önüne alınacaktır. Söz konusu yazılım borç toplayıcı avukatların karmaşık iş süreçlerinin daha kolay bir şekilde oluşturulması ve yönetilmesi amacıyla kullanılmaktadır. Bir borç toplayıcı avukat başka kuruluşlara ve/veya kişilere borcu olan bir borçlunun borçlarını toplama ve bunları asıl alacaklıya geri verme ile yükümlüdür. Borç toplayıcı avukat, borçluya ulaşmak için telefon görüşmesi, SMS, sesli mesaj veya ulusal kimlik ile SMS gönderme gibi yolları kullanabilmektedir. Ancak bir borçluya

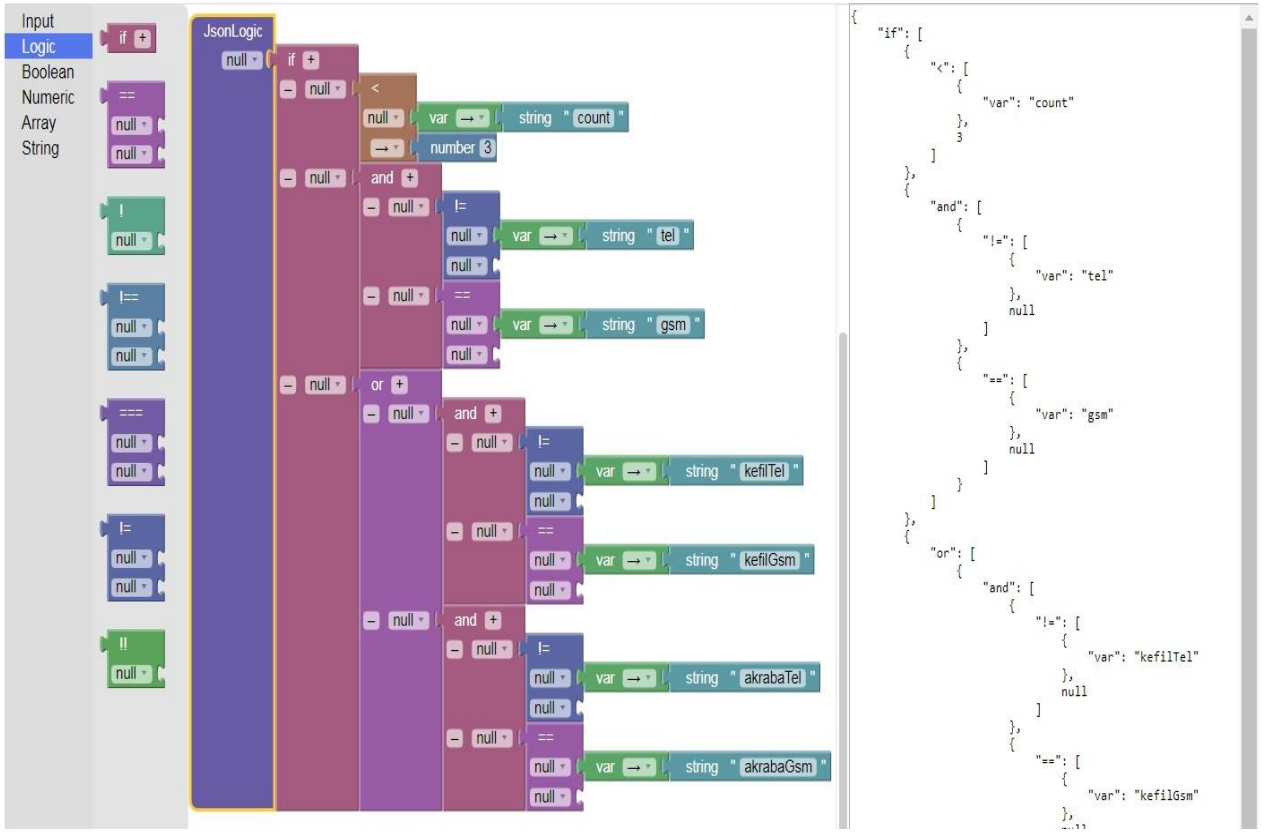
ulaşmak için, yalnız bu iletişim vasıtalarıyla kalmayıp faklı yollardan, örneğin borçlunun kefiline, annesine, babasına veya diğer akrabalarına ulaşmaya da çalışmaktadır. Böyle bir iş sürecinin yönetimi, ilgili iş süreci kuralları iyi kurgulanmadığında çok karmaşık bir hale gelmektedir.

Durum çalışmasında borç toplayıcı avukat tarafından tanımlanan bir borçluya ulaşma sürecinin yönetimi için işletilmesi gereken bir kuralın oluşturulması örnek olarak verilecektir. Bu kural ve karşılık gelen iş tanımına göre, üç gün boyunca borçlunun sabit telefonuna ulaşılmaya çalışılır. Eğer borçluya ulaşılamıyorsa, bu sefer üç gün boyunca borçlunun kefili veya akrabasının telefonuna ulaşılmaya çalışılır. İlgili kuralın formal tanımı aşağıda verilmiştir. Firmada bu iş kurallarını hazırlayan yazılım geliştiriciler bu kuralın ilgili iş süreci yönetimi yazılımının içerdiği kuralların en basitlerinden biri olmasına rağmen JsonLogic notasyonu ile hatasız bir şekilde hazırlanmasının zahmetli ve zaman alıcı olduğunu bildirmişlerdir.

Kural: {"if" : [{"<" : [{"var" : "count"},3]},{"and" : [{"!=" : [{"var" : "tel"},null]},{"==" : [{"var" : "gsm"},null]}]},{"or" : [{"and" : [{"!=" : [{"var" : "kefilTel"},null]},{"==" : [{"var" : "kefilGsm"},null]}]},{"and" : [{"!=" : [{"var" : "akrabaTel"},null]},{"==" : [{"var" : "akrabaGsm"},null]}]}]}]}

Şekil 3'teki ekran görüntüsünde, yukarıda anlatılan borçluya ulaşma iş süreci kuralının JsonLogic DSML'inin modelleme aracında oluşturulması gösterilmektedir. Modele karşılık gelen ve otomatik olarak üretilen JsonLogic yapılarından bir bölüm de şekilde sağ tarafta görülmektedir.

Gerçekleştirilen durum çalışması aynı zamanda JsonLogic için geliştirilen bu modelleme ortamının, Blockly'de sadece JSON modellemeye imkan veren [29]'daki araçla kıyaslanması ve kullanılabilirliğinin değerlendirilmesi açısından da faydalı olmuştur. Aynı iş kuralı [29]'daki modelleme aracı kullanılarak oluşturulmaya çalışılmıştır ancak JsonLogic için hazır yapılar yer almadığından ortaya çıkan blok yapısı, modeldeki "operation" örnekleri göz önüne alındığında, Şekil 3'teki modele kıyasla yaklaşık 3 kat daha büyük olmuştur. Bu model yer kısıtları nedeniyle bu bildiride gösterilememiştir. Örneğin [29]'daki araç kullanılarak modelleme yapılmak istendiğinde Şekil 3'teki iş kuralı modelinin içerdiği her bir "==" JsonLogic yapısı için en az 2 adet JSON bloğunun hazırlanması gerekmiştir.



Şekil 3. JsonLogic modelleme ortamında bir avukatlık iş takip süreci kuralının modellenmesi ve karşılık gelen JsonLogic yapılarının otomatik üretilmesi

7 İlgili Çalışmalar

Literatürde yer alan ilgili çalışmalar kural tabanlı iş süreçlerinin model-güdümlü geliştirilmesi ve JSON/JsonLogic veri ve mantık paylaşımı formatları ile ilgili olarak iki kategoride ele alınabilir.

İş süreçlerinin otomatik olarak oluşturulması ve yönetilmesinde, iyi bilinen BPMN [32], YAWL [33] gibi dillerin notasyon ve yapısallığını kullanan çalışmaların yanında başta model-güdümlü mimarinin kullanıldığı model-güdümlü geliştirme ve DSL / DSML üretme çalışmaları da giderek popülerleşmektedir. Örneğin Brambilla ve Pietro [8] yazılım uygulamalarının özellikle kullanıcı etkileşimlerinin tasarlanmasını ve akışların modellenmesini sağlayan IFML isimli bir dili önermekte ve bu dilin bazı iş alanlarında kullanımını örneklemektedirler. "Back-end" sistemler ile etkileşimde bulunan form-tabanlı ya da veri-tabanlı iş uygulamalarının model-güdümlü geliştirilmesi [9]'da anlatılmaktadır. Benzer şekilde iş uygulamalarının uygulamaya-özel fonksiyonlarının IIS*CFuncLang isimli bir DSL ve araç seti ile platform-bağımsız bir seviyede nasıl tanımlanacağı ve otomatik üretileceği bir eğitim bilgi sisteminin geliştirilmesi durum çalışması üzerinden [34]'te anlatılmıştır. BPMN modellerinin İş Süreci Çalıştırma Dili (BPEL) modellerine eşlenmesi ve BPMN modellerindeki güncellemelerin BPEL modellerine yansıtılması için bir metot [35]'te sunulmuştur.

MDD4CCA isimli DSL [10] bütünleşik içerik uygulamalarının form, gezinme, iş akışı, içerik, vb. bakışaçıları kullanılarak model-güdümlü geliştirilmesini sağlamaktadır. Benzer şekilde Bicevska ve ark. [36] iş süreçlerinin modellenmesini bir DSL kullanımı üzerinden sağlamıştır ve modellerden iş süreçlerinin çalıştırılma tanımları olay tabanlı mimariyi de içerecek şekilde üretilebilmiştir. Yakın zamanda Ferry ve ark. [11] bir "Hizmet olarak Altyapı" (ing. Infrastructure as a Service) (IaaS) için bulut uygulamalarının oluşturulması ve bulut servisi sağlayıcılarından bağımsız olarak bu uygulamaların yönetilmesi için bir model-güdümlü yaklaşım sunmuşlardır. Bir diğer DSL olan MAML [12] farklı mobil uygulamalar için iş süreçlerinin model-güdümlü geliştirilmesini ve bu uygulamalar için yerel kaynak kodların otomatik üretilmesi sağlamaktadır. Son olarak [37]'de servis-yönelimli mimari için iş süreçlerinin yönetimini sağlayan ve çeşitli soyutlama seviyelerine (örneğin mantıksal veya teknik) göre bu mimariye ait yazılım tasarımlarının geliştirilmesini model-güdümlü mühendislik temelleri ile gerçekleştirmeye izin veren bir yaklaşım tanıtılmıştır.

Yukarıda değinilen tüm bu iş akışlarının model-güdümlü geliştirilmesi ve iş akışı DSL'lerini oluşturma çalışmalarında iş akışlarının koreografisi ve yönetilmesi için gerekli olan iş mantığının ve kısıtlarının üretilen çözüme özel bir şekilde sunulduğu ve bu spesifik çözümlerin JSON veri değişimi formatının kullanımını ve bizim çalışmamızdaki gibi JsonLogic'e dayalı iş mantığının modellenmesini desteklemedikleri görülmektedir.

JSON göz önüne alındığında araştırmacıların daha çok bu veri değişim formatının veritabanları vb. kaynaklardan veri çekmede kullanımı üzerine çalışmalar yürüttüğü görülmektedir. Örneğin Bourhis ve ark. [38] JSON dokümanları için yapısal bir model önermişlerdir ve bu modeli kullanarak JSON dokümanlarında gezinmeyi

sağlayacak bir sorgu dili tanımlamışlardır. Özellikle üstveri tanımı eksikliğinden yola çıkılarak [39]'da JSON için bir şema tanımlanmaya çalışılmış; bu şemanın JSON dokümanlarının geçerlenmesinde kullanımı tarif edilmiştir. JSON verilerinin daha etkin ve paralel bir şekilde sorgulanabilmesi amacıyla [40]'da bir sorgu işlemcisi geliştirilmiştir. Çok büyük veri setlerinde bile veri yükleme maliyetlerinin önerilen işlemcinin içerdiği yeniden yazma kuralları ile düşürüldüğü gösterilmiştir. JSON üzerine gerçekleştirilen bu çalışmaların hiçbirinde JSON yapılarının görsel modellenmesi ve bu modeller üzerinden otomatik üretilmesi göz önüne alınmamıştır. Bizim çalışmamıza benzer bir şekilde Blockly [24] altyapısı ile JSON yapılarını modellemeyi sağlayan bir araç [29]'da bulunmaktadır. Ancak bu araçta da JsonLogic mantığı desteklenmemektedir. Bu bildiride tanıtılan model-güdümlü JsonLogic geliştirme mimarisinin hem JSonLogic'in üstmodelini ilk kez tanımlama hem de bu üstmodele dayalı olarak JsonLogic için yeni bir DSML sunma özellikleri ile JSON ve JsonLogic üzerine gerçekleştirilen tüm bu çalışmalara katkı verdiğine inanılmaktadır.

8 Sonuç

İş süreci kurallarının JsonLogic ile oluşturulması sırasında kullanılabilecek bir modelleme dili ve bunu destekleyen bir yazılım aracı geliştirilmiştir. JsonLogic'in bir MDA kapsamında platforma özel modelleme düzeyinde yer alabilecek bir üstmodeli türetilmiş; bu üstmodeldeki varlık ve ilişkilere dayan bir görsel somut sözdizim oluşturulmuştur. JavaScript ile yazılan ve modelden metne dönüşümlere dayalı bir işletimsel semantik sayesinde de modellenen iş kurallarına karşılık gelen JsonLogic yapıları otomatik olarak üretilebilmektedir. Tüm bu sözdizim ve semantiklerin birleşimi ile ortaya konulan JsonLogic DSML'ini yazılım geliştiricilerin kullanabilmesi için Blockly üzerine inşa edilmiş web tabanlı bir modelleme aracı da geliştirilmiştir. Sunulan modelleme ortamının örnek bir durum çalışması üzerinden gerçekleştirilen değerlendirmesinde salt JSON ile modelleme imkanı sunan [29]'daki araca göre ilgili iş kurallarının daha az görsel bileşen ile daha kolay bir şekilde oluşturulabildiği gözlenmiştir.

JsonLogic DSML'inin görsel sözdiziminin kullanışlılığının geliştirilmesi için dildeki notasyonların ifade gücü ve kullanıcılar tarafından benimsenmesi gibi açılardan değerlendirilmesi yakın zamanda hedeflenen çalışmalarımızdan biridir. Bu geliştirmeyi yapmak için Moody'nin "notasyonların fiziği" prensiplerine [41] göre mevcut JsonLogic DSML'inin bir değerlendirmesi planlanmaktadır. DSML'in JsonLogic yapılarını otomatik üretme ve iş kuralı geliştirme zamanını indirgeme açılarından niceliksel bir değerlendirmesinin çoklu durum çalışması üzerinden yapılması da ileriye yönelik planladığımız çalışmalar arasındadır.

Kaynaklar

1. van der Aalst, W. M. P., ter Hofstede, A. H. M., Weske, M.: Business Process Management: A Survey. Lecture Notes in Computer Science 2678, 1-12 (2003).

2. Wang, M., Wang, H.: From process logic to business logic—A cognitive approach to business process management. Information & Management 43(2), 179-193 (2006).
3. JSON, https://www.json.org/, son erişim: 15/11/2018.
4. JsonLogic, http://jsonlogic.com/, son erişim: 15/11/2018.
5. IBM Business Process Manager, https://www.ibm.com/us-en/marketplace/business-process-manager, son erişim: 15/11/2018.
6. Camunda BPM: Workflow and Decision Automation Platform, https://camunda.com/, son erişim: 15/11/2018.
7. Perez, J. M., Ruiz, F., Piattini, M.: MDE for BPM: A Systematic Review. Communications in Computer and Information Science 10, 127-135 (2006).
8. Brambilla, M., Fraternali, P.: Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. Morgan Kaufmann, Waltham, MA, USA (2015).
9. Majchrzak, T. A., Ernsting, I., Kuchen, H.: Achieving business practicability of model-driven cross-platform apps. Open Journal of Information Systems 2(2), 3-14 (2015).
10. Challenger, M., Erata, F., Onat, M., Gezgen, H., Kardas, G.: A Model-Driven Engineering Technique for Developing Composite Content Applications. In: Symposium on Languages, Applications and Technologies, pp. 11:1-11:10. Maribor, Slovenia (2016).
11. Ferry, N., Almeida, M., Solberg, A.: The MODAClouds Model-DrivenDevelopment. In: Di Nitto E., Matthews P., Petcu D., Solberg A. (eds): Model-Driven Development and Operation of Multi-Cloud Applications. pp. 23-33, SpringerBriefs in Applied Sciences and Technology. Springer, Cham (2017).
12. Rieger, C., Kuchen, H.: A process-oriented modeling approach for graphical development of mobile business apps. Computer Languages, Systems & Structures 53, 43-58 (2018).
13. OMG Meta-object Facility, https://www.omg.org/mof/, son erişim: 15/11/2018.
14. OMG Model-driven Architecture, http://www.omg.org/mda/, son erişim: 15/11/2018.
15. Eclipse Modeling Framework Technology, http://www.eclipse.org/modeling/emft/, son erişim: 15/11/2018.
16. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional, Westford, MA, USA (2011).
17. Gray, J., Tolvanen, J-P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. In Fishwick, P. A. (ed): Handbook of Dynamic System Modeling. pp. 1-7, CRC Press, Boca Raton, FL, USA (2007).
18. MetaEdit+ Domain-Specific Modeling (DSM) environment, https://www.metacase.com/products.html, son erişim: 15/11/2018.
19. Eclispe Graphical Modeling Framework, http://www.eclipse.org/modeling/gmp/, son erişim: 15/11/2018.
20. Eclipse Sirius, https://www.eclipse.org/sirius/, son erişim: 15/11/2018.
21. Saritas, H. B., Kardas, G.: A model driven architecture for the development of smart card software. Computer Languages, Systems & Structures 40(2), 53-72 (2014).
22. Kardas, G., Tezel, B. T., Challenger, M.: Domain-specific modelling language for belief-desire-intention software agents. IET Software 12(4), 356-364 (2018).
23. Challenger, M., Tezel, B. T., Alaca, O. F., Tekinerdogan, B., Kardas, G.: Development of semantic web-enabled BDI multi-agent systems using SEA_ML: an electronic bartering case study. Applied Sciences 8(5), 1-32 (2018).
24. Fraser, N.: Ten things we've learned from Blockly. In: 2015 IEEE Blocks and Beyond Workshop, pp. 49-50. IEEE, Atlanta, GA, USA (2015).

25. Pasternak E., Fenichel, R., Marshall, A. N.: Tips for creating a block language with blockly. In: 2017 IEEE Blocks and Beyond Workshop, pp. 21-24. IEEE, Raleigh, NC, USA (2017).
26. Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., Kafai, Y.: Scratch: programming for all. Communications of the ACM 52(11), 60-67 (2009).
27. Snap!, https://snap.berkeley.edu/, son erişim: 15/11/2018.
28. Blockly, https://developers.google.com/blockly/, son erişim: 15/11/2018.
29. An editor for JSON structures, http://ens-lg4.github.io/MenulyJSON/, son erişim: 15/11/2018.
30. Bryant B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., Karsai, G.: Challenges and Directions in Formalizing the Semantics of Modeling Languages. Computer Science and Information Systems 8(2), 225-253 (2011).
31. Hermes İnternet İletişim, https://www.hermesiletisim.net/, son erişim: 15/11/2018.
32. Chinosi, M., Trombetta, A.: BPMN: An introduction to the standard. Computer Standards & Interfaces 34(1), 124-134 (2012).
33. van der Aalst, W. M. P., ter Hofstede, A. H. M.: YAWL: yet another workflow language. Information Systems 30(4), 245-275 (2005).
34. Popovic, A., Lukovic, I., Dimitrieski, V., Djukic, V.: A DSL for modeling application-specific functionalities of business applications. Computer Languages, Systems & Structures 43, 69-95 (2015).
35. Radhakrishnan, G., Liu, L., Aggarwal, A., Saxena, V.: Roundtrip merge of bpel processes and bpmn models. US patent no: US20100057482A1.
36. Bicevska, Z., Bicevskis, J., Karnitis, G.: Models of Event Driven Systems. Communications in Computer and Information Science 615, 83-98 (2016).
37. Benaben, F., Truptil, S., Mu, W., Pingaud, H., Touzi, J., Rajsiri, V., Lorre, J. P.: Model-driven engineering of mediation information system for enterprise interoperability. International Journal of Computer Integrated Manufacturing 31(1), 27-48 (2018).
38. Bourhis, P., Reutter, J. L., Suarez, F., Vrgoc, D.: JSON: Data model, Query languages and Schema specification. In: 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, pp. 123-135. ACM, Chicago, IL, USA (2017).
39. Pezoa, F., Reutter, J. L., Suarez, F., Ugarte, M., Vrgoc, D.: Foundations of JSON Schema. In: 25th International Conference on World Wide Web, pp. 263-273. ACM, Montreal, Quebec, Canada (2016).
40. Pavlopoulou, C., Preston Carman, Jr, E., Westmann, T., Carey, M. J., Tsotras, V. J.: A Parallel and Scalable Processor for JSON Data. In: 21st International Conference on Extending Database Technology, pp. 576-587. Vienna, Austria (2018).
41. Moody, D.: "The "physics" of notations: toward a scientic basis for constructing visual notations in software engineering. IEEE Transactions on Software Engineering 35(6), 756-779 (2009).