# SBVR2Alloy: an SBVR to Alloy compiler

1st Nurulhuda A. Manaf
*Department of Computer Science*
*University of Surrey*
Guildford, United Kingdom
n.amanaf@surrey.ac.uk

2nd Andreas Antoniades
*Department of Computer Science*
*University of Surrey*
Guildford, United Kingdom
a.antoniades@surrey.ac.uk

3rd Sotiris Moschoyiannis
*Department of Computer Science*
*University of Surrey*
Guildford, United Kingdom
s.moschoyiannis@surrey.ac.uk

***Abstract*—We present a compilation tool *SBVR2Alloy* which is used to automatically generate as well as validate service choreographies specified in structured natural language. The proposed approach builds on a model transformation between *Semantics of Business Vocabulary and Rules* (SBVR), an OMG standard for specifying business models in structured English, and the *Alloy Analyzer* which is a SAT based constraint solver. In this way, declarative specifications can be enacted via a standard constraint solver and verified for realisability and conformance.**

***Index Terms*—service choreography, constraints, behavioural modelling, model transformation, verification, realisability, complex interactions**

## I. Introduction

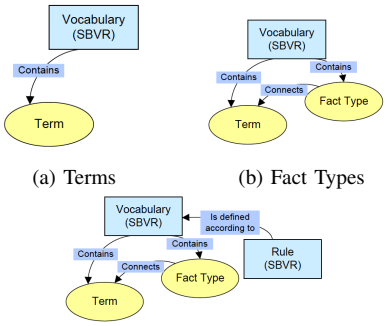
A promising approach to expressing complex business requirements in a declarative manner is the OMG standard *Semantics of Business Vocabulary and Business Rules* (SBVR) [1]. As defined by [2], "*SBVR provides a way to capture specifications in natural language and represent them in formal logic so they can be machine-processed*". Thus, users can validate the underlying service choreography by directly reading the structured natural language used in SBVR models. The idea is that this can then be parsed and executed by a machine.

In this paper we describe an automated tool that translates declarative specifications, given as SBVR models, into Alloy. The *SBVR2Alloy* compilation tool builds on the model transformation described in detail in [3]. The benefit is that Alloy can automatically generate the service choreography, corresponding to the input SBVR model. In addition to verifying conformance to message ordering constraints, the *Alloy Analyzer* [4] can be used to perform realisability checks and assert static constraints on the generated choreography.

Alloy [5] is a language for describing structures. An Alloy model is a collection of constraints that describes a set of structures; e.g., all possible executions that satisfy a collection of message ordering constraints prescribed (implicitly) in a service choreography. The *Alloy Analyzer* [4] is an automated solver that takes the constraints of a model and finds structures that satisfy them.

The main contribution of this paper is a reference implementation of an SBVR parser and its operationalisation into the Alloy Analyzer. This means that SBVR models can be compiled into Alloy syntax and effectively be verified.

The first author is supported by the Malaysian Government and the National Defence University of Malaysia (NDUM)




(a) Terms (b) Fact Types

(c) Rules

Fig. 1: SBVR pillars: Terms, Fact Types, Rules

The paper is structured as follows. Section II briefly introduces SBVR and Alloy. Section III outlines key aspects of realising the implementation of SBVR2Alloy while Section IV demonstrates the tool in the context of a running case study. Related work is discussed in Section V and Section VI concludes the paper.

## II. SBVR and Constraint Solving

We provide a brief introduction to SBVR and the Alloy constraint solver, and then proceed to outline how an SBVR model is built.

### A. SBVR Structured English

SBVR is an artefact of the *Business Rules* approach [6] and as such it follows the mantra: "*Rules build on facts, and facts build on concepts expressed by terms. Terms express business concepts; facts make assertions about these concepts; rules constrain and support these facts*". This can be seen in Figure 1.

While SBVR is a meta-model with models natively expressed as logical formulations, its most common serialisation is the so-called SBVR Structured English (SBVR-SE). Terms (e.g, **customer**), Fact Types (e.g., **customer** *sends* **reservation request**), and Rules (e.g., It is obligatory that the **customer** *sends* exactly one **reservation request**) are combined into SBVR models.

The colouring in the fonts is prescribed in the SBVR spec document [1].

### B. Alloy constraint solver

Alloy's syntax and semantics are documented in [4] but we recall some main notions here that help with understanding the *SBVR2Alloy* tool presented in this paper. Data domains are defined using signatures (denoted by `sig`) and represented as sets. A signature may extend another signature, as in the object-oriented paradigm, in which case the domain defined by the first is a subset of the domain of the extended signature. Extensions of a signature are mutually disjoint. A signature can be `abstract` in which case its domain only contains elements that belong to its extending signatures.

In addition, signatures contain `fields` which are captured by relations. Axioms in Alloy are called `facts` and can be given a name. These must hold at any time. Alloy formulae often use standard connectives from first-order logic, quantifiers `all` (universal) and `some` (existential). In general, expressions in Alloy are built using set theoretic relational operators and constants. This means it is an appropriate target for SBVR which is declarative in nature and has a logical formulation (see ch. 10 in [1]).

### C. Building an SBVR model

We model a well known case study from [7], also studied in [8] in view of declarative specification of service choreographies, of a hypothetical *ACME Travel* scenario and the multi-party conversation involved in arranging travel. In brief, a customer sends a reservation request which may include one or more of airline, accommodation (hotel or apartment), transport (bus or train or taxi), and tour reservations. Once Acme Travel (AT) receives the itinerary request it sends reservation requests to the different providers and awaits for responses. Once all (un)successful reservations are in, AT sends a notification to the customer.

In what follows we outline certain aspects of the SBVR model built for this case study[1].

In order to model service chains and choreographies we need to capture participant services (*participants*) and the messages exchanged between them (*events*). These will be Terms in the Vocabulary of the SBVR model. For example, Term: **participant**, Term: **customer**, Term: **reservation request**.

It is often the case that participants and events in a business activity are grouped together, e.g., the transport reservation concerns a train reservation or a bus reservation. To preserve the semantics we use the Sets definition **set** *includes* **thing** found in the latest SBVR specification document [1]. This can be defined then in the Fact Type: **participant** *includes* **customer**.

Next, Fact Types and the SBVR construct **Term** *verb* **Term** can be used to capture the sending and the receiving of a message. For example, the export message of **customer** in the Fact Type: **customer** *sends* **reservation request** and the import message of AT in the Fact Type: **AT** *receives* **reservation request** capture the interaction between them.

As mentioned previously SBVR Structured English (SBVR-SE) [1] has a logical formulation which draws from first order logic (when ignoring the modalities). The logical connectives for exclusive disjunction (XOR), conjunction (AND), and inclusive disjunction (OR) are used on participants, events (Terms) as well as Fact Types in forming Business Rules in an SBVR model. For example, the following rule in the Acme Travel case study expresses the business constraint that a response to a reservation request is either successful or unsuccessful. Rule: It is obligatory that [...] exactly one **accommodation reservation response** *includes* exactly one **successful accommodation reservation** or exactly one **unsuccessful accommodation reservation** but not both

The conjunction is used to express that both events take place, in no particular order. The general form when unordered events are concerned is the following:
Rule: It is obligatory that [...] exactly one **participant** *sends* exactly one **event_1** and exactly one **event_2**

The inclusive disjunction is used to express choice. For example, the following rule in the Acme Travel case study expresses the business constraint that AT may request an airline reservation or an accommodation reservation or a tour reservation or a transport reservation.
Rule: It is obligatory that the **AT** *requests for* exactly one **airline reservation** or exactly one **accommodation reservation** or exactly one **tour reservation** or exactly one **transport reservation**

In [10] we introduced a notion of precedence in SBVR models drawing upon the Date-Time Vocabulary (DTV) [11] supplement to the SBVR specification. Our approach here builds on that work but also benefits from the latest SBVR specification by OMG, namely SBVR 1.3 [1], which includes a notion of *immediate* precedence from DTV. In short, these developments mean that by definition **event 1** *immediately precedes* **event 2** demonstrates that there is no occurrence of an event after **event 1** and before **event 2**. The *immediately precedes* construct can be combined with the logical connectives, as in:
Rule: It is obligatory that exactly one **event 1** and exactly one **event 2** *immediately precedes* exactly one **event 3**
where and takes precedence over *immediately precedes*.

In modelling multi-party conversations it is often the case that an event is associated with different participants as in the Fact Types **customer** *sends* **reservation request** and **AT** *receives* **reservation request** discussed earlier. It can be seen there is no indication to inform which interaction is performed initially. To overcome this we incorporate a notion of time understood as in the construct **occurrence** *at* **time interval** which is consistent with [11]. This means we can have time declarations:
**customer** *sends* **reservation request** *at* **T1**
**AT** *receives* **reservation request** *at* **T2**
and **T1** *immediately precedes* **T2**

Finally we discuss how static, domain-specific constraints

[1]The full blown SBVR model of the case study can be found following [9].

can be expressed in an SBVR model. In the Acme Travel case study there was a need for specifying certain constraints on the dates of travel arrangements. Dates can be treated like participants and events earlier, so there is a Date set declared as Term: **date**. Other dates such as **start date**, **check-in date**, also declared as terms, can be specified to be members of the Date set using the *set* definition in [1] and the construct *is in* which declares belonging, in the corresponding fact types:
Fact Type: **start date** *is in* **date**
Fact Type: **end date** *is in* **date**
Fact Type: **check-in date** *is in* **date**
The following SBVR rules should be self-explanatory.
It is obligatory that exactly one **start date** *of* exactly one **reservation request** *equals* exactly one **check-in date** *of* exactly one **accommodation reservation**. Similarly,
It is obligatory that exactly one **start date** *of* exactly one **reservation request** *equals* exactly one **outbound date** *of* exactly one **airline reservation**.

The input to our *SBVR2Alloy* compiler is a plain text version of the SBVR model with minimal markup (in the form of comments) to denote whether each line should be interpreted as a Term, a Fact Type or a Rule. To be more precise, we also need to know whether a Term is a participant or an event or a static constraint or a time declaration.

## III. Preprocessing - Realising the Model Transformation

Before the implementation of the tool could commence, we had to formally define the transformation between SBVR and Alloy models. The Alloy model is obtained by considering the abstract syntax of SBVR and associated constraints to generate the exact solution in Alloy that corresponds to the input SBVR model. Due to space limitations we cover only key aspects here. The interested reader is referred to [3], where the transformation is described in detail.

We describe the exact steps taken for the transformation between the SBVR and Alloy models. The translation is a two stage process, with each stage involving multiple steps which need to be combined for generating Alloy code.

### A. First Stage

The first stage encompasses direct Alloy code generation by reading in the input SBVR file. The first step of this stage involves parsing the SBVR 'Term:' and 'Fact Type' keywords and generating the corresponding Alloy structures, namely the *participant* and *event* signatures. A visualisation of how terms are associated with each other can be seen in Fig. 2.

Participants and events can include:

- *abstraction*, e.g., **event** *includes* **transport reservation**
- *inheritance*, e.g., **transport reservation** *includes* **bus reservation**
  **transport reservation** *includes* **train reservation**
  and the rules:
  It is obligatory that exactly one **transport** that *includes* the **bus** *receives* exactly one **transport reservation** that *includes* exactly one **bus reservation** *at* exactly one **t2**

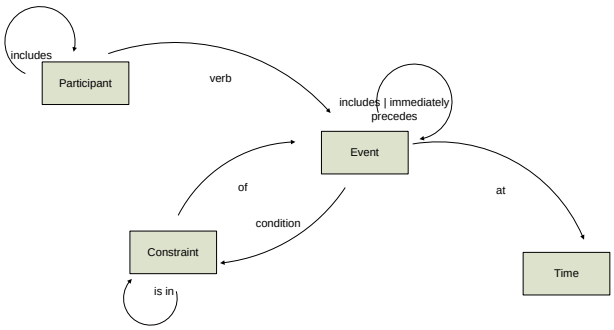



Fig. 2: Association between terms in SBVR and Alloy

  It is obligatory that exactly one **transport** that *includes* the **train** *receives* exactly one **transport reservation** that *includes* exactly one **train reservation** *at* exactly one **t2**

Both cases need to be noted as the generation of Alloy `facts`, which is performed in the Second Stage, heavily relies upon them.

The timing information regarding participants and associated events is also important; throughout this stage when a participant 'Fact Type' includes the 'at' keyword, the time association must be saved in memory to generate corresponding time declarations in Alloy.

After participant and event signatures are generated, the static constraints which are also defined using the 'Term' keyword in SBVR, must be generated for Alloy. This involves defining the cardinality of the constraint and which set it belongs to. For example, Fact Type: **start date** *is in* **date** translates to:

```
1 one sig start_date in date{}
```

The final step in the first stage of the translation involves generating the time declarations stored in memory during the signature generation step. We have used the '_time' postfix to define the time declarations associated with events. Using our case study as an example, the Alloy time declaration for transport reservation is defined as:

```
1 abstract sig transport_reservation_time extends time{}
2 one sig t1_transport_reservation extends
transport_reservation_time {by: one AT,
immediately_precedes: t2_transport_reservation}
3 one sig t2_transport_reservation extends
transport_reservation_time {by: one transport}
```

Note that in order to comply with Alloy, an event that takes place at time `t1` must always precede an event that takes place at `t2`. In addition, we have followed the same approach to dealing with multiple word verbs as we did for terms,

e.g., 'immediately precedes' in the SBVR model is interpreted as `immediately_precedes` (line 3 in the above code snippet) in the Alloy model.

### B. Second Stage

This stage completes the translation process and involves more complex steps that rely on the Alloy code we have generated in the First Stage.

First we append the Alloy `facts` for each participant and event by consulting both the SBVR rules, defined using 'Rule', and the verbs in the signatures we have generated. Any inheritance information also needs to be utilised to generate the facts. An example also mentioned in Sec. III-A is transport reservation which can be handled either by a train or by a bus. This is reflected in the `fact` :

```
–fact
( transport_reservation = bus_reservation and no
train_reservation ) or ( transport_reservation =
train_reservation and no bus_reservation )
```

Next the predicate for constraints associated with time must be generated. This step involves generating an *ordering* of the events based on the initial event, in our case **reservation request** and all associated events.

For example, the following Alloy snippet produced by our transformation shows ordering but also the exclusive disjunction (XOR) and the use of time.

```
4 abstract sig accomresponse extends
event{immediatelyprecedes: one notification, at: one T1_ACR,
at1: one T2_ACR}
–fact
{((accomresponse = succaccomres and no unsuccaccomres) or
( accomresponse = unsuccaccomres and no succaccomres))
and immediatelyprecedes = AT.sends and immediatelyprecedes =
customer.receives}
...
9 one sig notification extends event{at: one T1_N, at1: one
T2_N}
```

Line 9 above defines a signature for the event `notification` which is declared as a term in the SBVR model and implicated in the following rules:
It is obligatory that the **AT** *sends* exactly one **notification** *at* exactly one **t1**
It is obligatory that the **customer** *receives* exactly one **notification** *at* exactly one **t2**
It is obligatory that exactly one **accommodation reservation response** that *includes* exactly one **successful accommodation reservation** or exactly one **unsuccessful accommodation reservation** but not both *immediately precedes* exactly one **notification**

The first two rules declare time explicitly, where **t1** immediately precedes **t2**. This is captured in the `at` fields of the corresponding event signature (here `notification`). In this case it declares the fact that `AT` sending the notification happens before the customer receiving it.

The third rule defines precedence between `accomresponse` and `notification` events and is captured using the `immediatelyprecedes` field of the preceding event's signature. In Alloy, this is `accomresponse`, which in itself includes a XOR on `successful accommodation reservation` and `unsuccessful accommodation reservation`. The accompanying *fact* ensures that only one or the other of these events occurs at any one execution.

After appending all the information in the Alloy signatures, the next step is to generate the service choreography by executing the output Alloy model (the .als file) in the Alloy Analyzer tool. Again this step includes the Alloy signature of the initial event and defines the interactions between associated events.

Finally, concrete requests are identified and translated to Alloy using a combination of the generated Alloy signatures, the SBVR rules and request-specific rules, which can be also defined in the SBVR model.

### C. Implementation Details

After reviewing the two stages of the translation methodology we have defined, we concluded that the tool would follow a state machine paradigm. State machines are thoroughly used in the literature to define models that can behave in different ways depending on their state [12]. This approach is applicable as generating Alloy code for participant signatures differs from that of event signatures. Using the template we defined, the tool parses the SBVR file and changes its state depending on prefixed comments in the code.

For example when the tool reaches the '–Participants Set' comment it changes its state to participant ('p') and handles all participant related processes for translation, the same applies for events, constraints and rules. A total of 12 states were identified to encompass the two stages of translation.

The tool was realised using Java and a simple GUI was created using the Java Swing classes. This enabled us to provide a platform independent tool for translating SBVR to Alloy.

### D. Challenges

The difficulty of translating semantic languages became apparent in the early stages of our analysis and design. A concrete example is the usage of the 'Term' keyword in SBVR, which it is used to define participants, events and constraints (both static such as those on dates discussed earlier, and dynamic such as those relating to message ordering). These concepts carry a distinct meaning in a multi-party conversation and consequently their handling in Alloy is not uniform.

Common use of SBVR models involves using comments to add structure and increase usability as well as readability. Creating a universal SBVR to Alloy compiler would involve taking into account an infinite number of comments. Instead,

we have created a template for building SBVR models that should be followed when using our *SBVR2Alloy* tool. This template serves to identify the different sections in the SBVR model for a more structured approach to translation.

We have found that Alloy is more restrictive than SBVR in a number of ways. For example, SBVR allows for terms comprising multiple words whereas Alloy only allows for single word terms. To overcome this we replaced the white spaces between words in multi-word terms with underscores, e.g., the term 'transport reservation response' in the SBVR model is translated to 'transport_reservation_response' in Alloy.

Another challenge we faced in the translation process had to do with the use of the same verb in different relationships involving the same term. SBVR can reuse the same verb for a relationship between two terms, whereas Alloy allows for a verb to be used only once for each term. For example, the definition for the transport reservation in our case study uses the verb 'has' twice as seen below:

Fact Type: **event** *includes* **transport reservation**
Fact Type: **transport reservation** *has* **departure date**
Fact Type: **transport reservation** *has* **arrival date**

For the corresponding Alloy signature to be valid, we have added numbers to the end of each field capturing the 'has' verb, resulting in the following signature:

```
1 abstract sig transport_reservation extends event{has: one
departure date, has1: one arrival date}
```

Addressing the inheritance of events amongst nested (*includes*) participants increased the complexity of the code as it resulted in an additional dimension in the data structure and one more parsing of related constructs in the SBVR model. However, the Alloy language requires this type of indirection (or inheritance in SBVR) in order to distinguish between inclusive (OR) and exclusive disjunction.

## IV. SBVR2Alloy: Illustration by Example

We demonstrate the core functionality of the *SBVR2Alloy* compilation tool.

### A. Alloy code generation

The input to *SBVR2Alloy* is a plain text version of the SBVR model. The output is the Alloy file (.als) that can be parsed by the *Alloy Analyzer* tool [4]. The input plain text file is uploaded from the local drive and is then shown in the pane on the left of the *SBVR2Alloy* GUI, shown in Figure 3.

The input can be transformed to Alloy code now by pressing the 'Convert' button on the top right. The result is shown in Figure 4.

The Alloy code can be exported locally as .als file which can then be opened with the *Alloy Analyzer*, as shown in Figure 5.

In this way the declarative specification of the multi-party conversation, including constraints on the message ordering,

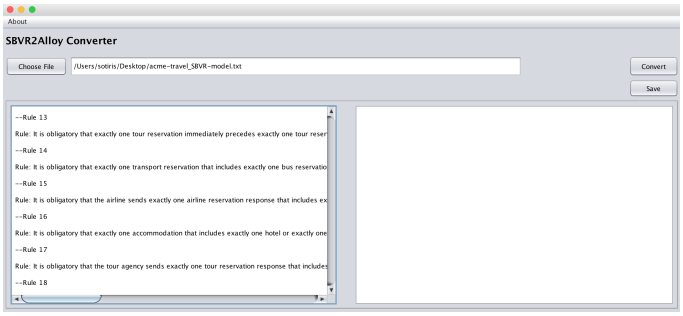



Fig. 3: SBVR model input uploaded as a text file

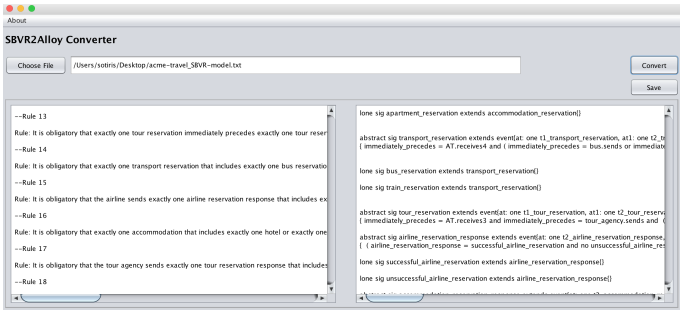



Fig. 4: Alloy code (right) produced for the SBVR model (left)

which was given as an SBVR model is now manifested in the resulting Alloy code. This means the *Alloy Analyzer* can be readily used to find structures that satisfy these constraints, effectively generating the underlying service choreography.

### B. Generating service choreographies in Alloy

A *predicate* is employed in Alloy to find an instance of the global constraints. An instance is a situation in which both the facts in the model and the predicates hold. The following code snippet illustrates the predicate `pred initialevent`.

```
14 pred initialevent[r:reservationrequest, a:
accomreservation, l:airlinereservation, t:tourreservation,
s:transreservation] {(r.immediatelyprecedes = l or
r.immediatelyprecedes1 = a or r.immediatelyprecedes2 = t
or r.immediatelyprecedes3 = s)}
15 run initialevent
```

Figure 6 illustrates the corresponding instance of the choreography (global constraints) generated by Alloy.

The predicate `pred initialevent` applied above declares the reservation request as the initial event. Thus, the diagram should be read starting from this event, i.e., `reservationrequest`. The incoming arrows from participants `customer` and `AT`, annotated by `sends` and `receives` respectively, capture the interaction. Note that we have not included time here, to not clutter the diagram but it can be included. The outgoing arrows of `reservationrequest` depict what happens next; `airlinereservation`, `apartmentreservation`,

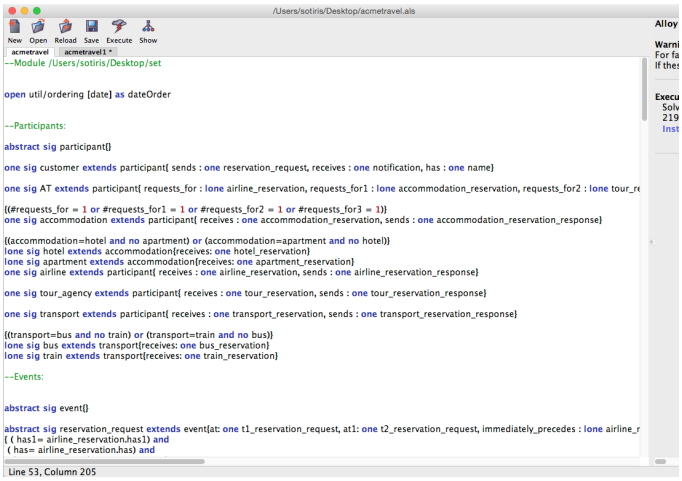



Fig. 5: Resulting Alloy code in the Alloy Analyzer

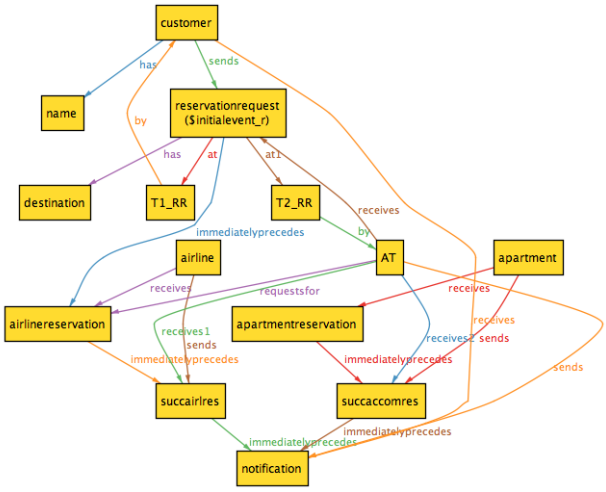



Fig. 6: Graphical representation of the generated choreography

`tourreservation` and `trainreservation` occur immediately after *reservationrequest* in no particular order, i.e., they are *unordered*.

### C. Verification: Realisability and Static Constraints

The Alloy Analyzer can now be used to perform verification on the generated choreography. In particular, it can readily verify whether a specific request can be realised by the given service choreography.

Assume that a user requests either an airline reservation only (`t.requestsfor = a`), or an accommodation reservation restricted to hotel (`t.requestsfor1 = h`) and a transport reservation restricted to a bus (`t.requestsfor3 = b`), and no tour reservation. This request can be formulated as a predicate in Alloy. This is shown in the following code snippet.

```
1 pred concreterequest [t:AT, h:hotelreservation,
a:airlinereservation, b:busreservation]
{(t.requestsfor1 = h and t.requestsfor3 = b and
t.requestsfor = a and no t.requestsfor2)
or (t.requestsfor = a and no t.requestsfor1 and no
t.requestsfor2 and no t.requestsfor3)}
2 run concreterequest
```

Figure 7 depicts one possible execution; `AT`'s requests for bus, hotel and airline reservations have been realised.

The successful responses for each reservation have been sent by each provider (bus provider and hotel provider, and airline provider in the other case). These are immediately preceded by the reservation requests from `AT`. They in turn immediately precede the notification to the customer. Please note that the tour reservation is not part of the choreography as it was not selected.

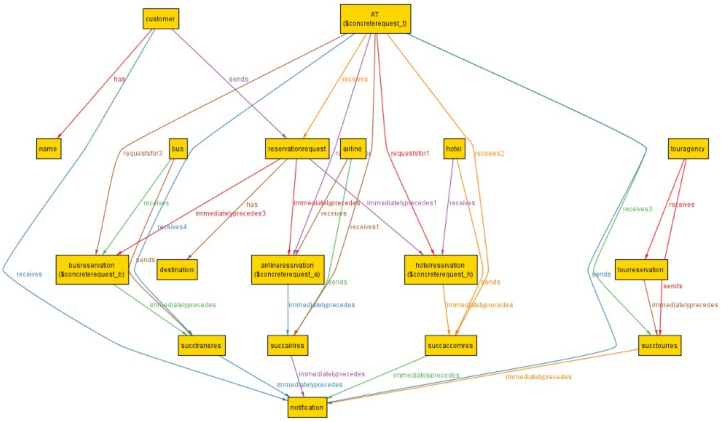



Fig. 7: One possible realisation of `concreterequest`

Global constraints in an Alloy model are verified by applying assertions. The following snippet shows the assertions necessary for verifying the static constraints that were declared in the SBVR rules in Section II-C. In further explanation, the statements in line 12 check whether the start (end) date of the reservation request is equal to the outbound (inbound) date of the airline reservation and the check-in (check-out) date of the accommodation reservation.

```
12 assert flightaccomvalidate {all a:airlinereservation,
r:reservationrequest, c:accomreservation|
(r.startdate = a.outbounddate and r.enddate = c.checkoutdate
and a.outbounddate = c.checkindate) and r.startdate !=
r.enddate}
check flightaccomvalidate
13 assert transvalidate {all disjoint d,d1,d2,d":one Date,
r:reservationrequest, t:tourreservation, p:transreservation,
a:accomreservation |
traveldate[d,d1,d2,d",r,t,p,a] implies ((p.departuredate =
p.arrivaldate) or ( p.departuredate = d and p.arrivaldate =
d1)
or (p.departuredate = d and p.arrivaldate = d2) or
( p.departuredate = d and p.arrivaldate = d") or (
p.departuredate = d1 and p.arrivaldate = d2)
or ( p.departuredate = d1 and p.arrivaldate = d") or
```

```
(p.departuredate = d2 and p.arrivaldate = d"))}
check transvalidate
```

Line 13 is the assertion to validate that the specified dates for the transport reservation are within the start and end date of the reservation request.

If an assertion is not valid, Alloy with produce a counterexample which can be shown in a graph like before. In this case, the assertion is valid.

## V. Related Work

Efforts geared towards specifying service choreographies include W3C's *Web Services Choreography Description Language* (WS-CDL) [13] and the inclusion of choreography diagrams in the most recent specification of BPMN. Like UML design models it provides user-friendly graphical notations but these often exhibit various divergent semantics. UML collaboration diagrams [14] and UML 2 sequence diagrams [15], [16] have also been proposed, but drawing the diagrams typically requires integration with a UML tool.

Work on formal semantics in this area has focused more on the imperative (or procedural) approach [14], [17], [18], [19], [20], [21], [16], [22] and has been geared towards an interleaving semantics which sometimes unnecessarily restricts the possible behaviours, e.g., see *PROPANE* in [20], or forces the modeller into making premature decisions, e.g., see *separated* collaboration diagrams [14]. A *true concurrency* semantics is proposed in [16].

In comparison, work on a declarative approach to interaction-based service choreographies is limited, e.g., [10], [23], [24], [25]. The focus seems to be more on reasoning about the consistency of the rule set, which of course is an important aspect of verification, and less on explicitly capturing the orderings in terms of observable message exchanges. The work on *DecSerFlow* [8] includes a graphical interface for user interaction but this is proprietary notation. In contrast, our approach uses SBVR for this purpose, which was developed with the business user in mind and is a standard maintained by OMG.

In [26] a first-order deontic-alethic logic (FODAL) is provided to express business constraints defined in SBVR and perform a consistency check on the rule set, including *alethic* and *deontic* rules. In our approach we restricted to deontic rules - that is, obligation - as we are targeting service choreographies and obligation is adequate. The deontic modality also covers *prohibition*. This can be addressed in our approach by considering the negation of the corresponding obligation instead.

The case study used to illustrate our approach in this paper is adapted from [7] and [8]. The amends to it concern an extended scenario and involve more complex interactions. This can be seen by comparing with the activities considered in [7] and [8]. In contrast to [8], we consider that all interactions are executed successfully. Compensating behaviour is beyond the scope of the current paper. We refer the interested reader to work in [27], [28] on web transactions and the work in [16] on providing transactional guarantees to service interactions in the absence of a central coordinator.

The Alloy Analyzer [4] has been used in a range of application domains. Of relevance here perhaps is the work [29] where Alloy is used for composing behavioural models from individual UML 2 sequence diagrams. [30] generate OCL (Object Constraint Language) via SBVR structured English and then transform these constraints into Alloy. [31] also applies Alloy to an e-commerce system modelled using UML use-case diagrams and activity diagrams.

SBVR is gaining ground in systems specification and modelling due to its declarative nature. The work in [32] translates UML / OCL into SBVR to bridge the gap between developers and business users. [33] translates SBVR into a process BPMN and decision model which is used to validate SBVR rules. In [34] SBVR is used to generate a relational database and transform SBVR business rules to SQL queries that can be executed against the data set. This widens the spectrum in which to consider user interaction, moving from process-driven to user-driven development of information systems.

We note that Alloy employs a SAT-based back end and performs bounded exhaustive search during analysis. This can be limiting when it comes to analysing higher-order transformations. *F-Alloy* [35] has been proposed to overcome such limitations.

## VI. Concluding Remarks

We proposed the use of SBVR for modelling service choreographies in [10]. This work introduced a notion of precedence, drawing from the *objectification* construct in SBVR 1.2 and the Date-Time Vocabulary (DTV) [11]. The model transformation behind *SBVR2Alloy* draws upon the latest SBVR specification by OMG (SBVR 1.3 [1]) which includes a construct for expressing *immediate* precedence. This is key for verification, e.g., *realisability*. In addition, in [10] the focus was more on modelling choreographies rather than *generating* choreographies and automating the required verification task, as done in this paper.

The model transformation from SBVR to Alloy has been applied to 3 case studies of varying size. The full SBVR model for the *Acme Travel* case study used in this paper features 39 rules and can be found in [9]. The model transformation can be found in [3]. Using a standard specification computer with 2.8 GHz and 16GB RAM, it takes *SBVR2Alloy* 58ms to produce the corresponding Alloy code (.als file). When employing the SAT4J solver of Alloy, the execution of the `pred initialevent` discussed in Section IV-B takes between 23ms and 168ms, depending on what other processes are running.

Our *SBVR2Alloy* tool does not implement the full breadth of the SBVR standard specification [1] but rather a large and usable subset which can be used to express complex rules, with a focus on capturing constraints on the orderings of service invocations. This makes it possible to use the corresponding Alloy code to perform verification tasks such as *realisability*

in service choreographies which involve dynamic constraints, as well as conformance of domain-specific static constraints. The tool can be extended to include less common features of SBVR and indeed this is part of the future work planned.

## REFERENCES


[1] OMG, *Semantics of Business Vocabulary and Business Rules (SBVR), v1.3*, OMG document formal/2015-05-07, http://www.omg.org/spec/SBVR/1.3/, May 2015.

[2] S. Hendryx, *Model-Driven Architecture and the Semantics of Business Vocabulary and Business Rules (SBVR)*, 2005.

[3] https://tinyurl.com/ya9p2h5r.

[4] http://alloy.mit.edu/alloy/.

[5] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*, ser. Revised Edition. The MIT Press, 2012.

[6] R. G. Ross, *The Business Rules Manifesto, Version 2*, 2003.

[7] J. Snell, *Automating business processes and transactions in Web Services: An introduction to BPELWS, WS-Coordination, and WS-Transaction*, IBM, 2006.

[8] W. M. P. van der Aalst and M. Pesic, "Decserflow: Towards a Truly Declarative Service Flow Language," in *Web Services and Formal Methods (WS-FM) 2006, LNCS 4184*, 2006, pp. 1–23.

[9] https://tinyurl.com/y9t8x4cy.

[10] N. A.Manaf, S. Moschoyiannis, and P. Krause, "Service Choreography, SBVR, and Time," in *Foundations of Coordination Languages and Self-Adaptive Systems (FOCLASA 2015)*, 2015, pp. 63–77.

[11] OMG, *Date-Time Vocabulary (DTV), Version 1.3*, OMG document formal/dtc/2016-02-20, http://www.omg.org/spec/DTV/1.3/Beta2, 2016.

[12] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Transactions on Software Engineering*, vol. SE-4, no. 3, pp. 178–187, May 1978.

[13] W3C, *Web Services Choreography Description Language (WS-CDL)*, W3C Working Group, http://www.w3.org/TR/ws-cdl-10-primer/, 2006.

[14] T. Bultan and X. Fu, "Specification of Realizable Service Conversations using Collaboration Diagrams," in *In Service-Oriented Computing and Applications, SOCA*, 2007, pp. 122–132. [Online]. Available: http://dx.doi.org/10.1109/SOCA.2007.41

[15] B. Bauer and J. P. Muller, "MDA applied: From Sequence Diagrams to Service Choreography," in *ICWE 2004*, ser. LNCS, no. 3140. Springer, 2004, pp. 132–136.

[16] S. Moschoyiannis and P. J. Krause, "True Concurrency in Long-running Transactions for Digital Ecosystems," *Fundamenta Informaticae*, vol. 138, no. 4, pp. 483–514, 2015. [Online]. Available: http://dx.doi.org/10.3233/FI-2015-1222

[17] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, "Towards a formal framework for choreography," in *WETICE 2005*, ser. IEEE Computer Society, 2005.

[18] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot, *A Theoretical Basis of Communication-Centred Concurrent Programming*, 2006.

[19] G. Cledou, J. Proenca, and L. Barbosa, "Composing Families of Timed Automata," in *Fundamentals of Software Engineering (FSEN)*, ser. LNCS, 2017, to appear.

[20] X. Fu, T. Bultan, and J. Su, "A Formalism for Specification and Analysis of Reactive Electronic Services," *Theoretical Computer Science*, vol. 328, no. 1-2, pp. 19–37, 2004.

[21] M. Gudemann, P. Poizat, G. Salaun, and L. Ye, "VerChor: A framework for the design and verification of choreographies," *IEEE Transactions on Services Computing*, vol. 9, no. 4, pp. 647–660, 2016.

[22] J. Su, T. Bultan, X. Fu, and X. Zhao, "Towards a theory of web service choreographies," in *Web Services and Formal Methods WS-FM*, 2007, pp. 1–16.

[23] M. Autili and M. Tivoli, "Distributed enforcement of service choreographies," in *Proceedings Int'l Workshop on Foundations of Coordination Languages and Self-Adaptive Systems (FOCLASA)*, 2014, pp. 18–35.

[24] J.-M. Jacquet, I. Linden, , and M.-O. Staicu, "On the introduction of time in distributed blackboard rules," in *Foundations of Coordination Languages and Self-Adaptive Systems, FOCLASA*, 2013, pp. 144–203.

[25] M. Montali, M. Pesic, W. M. P. van der Aalst, F. Chesani, P. Mello, and S. Storari, "Declarative specification and verification of service choreographies," *ACM Transactions on the Web, TWEB*, vol. 4, no. 1, pp. 3:1–3:62, 2010.

[26] D. Solomakhin, E. Franconi, and A. Mosca, "Logic-based Reasoning Support for SBVR," *Fundamenta Informaticae*, vol. 124, no. 4, 2013.

[27] M. J. Butler, C. A. R. Hoare, and C. Ferreira, "A Trace Semantics for Long-Running Transactions," in *Communicating Sequential Processes: The First 25 Years*, 2004, pp. 133–150.

[28] L. Bocchi, C. Laneve, and G. Zavattaro, "A Calculus for Long-Running Transactions," in *Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, 2003, pp. 124–138.

[29] J. K. Bowles, B. Brodbar, and M. Alwanain, "A Logical Approach for Behavioural Composition of Scenario-based Models," in *17th International Conference on Formal Engineering Methods (ICFEM)*, ser. LNCS, vol. 9407. Springer, 2015, pp. 252–269.

[30] S. Malik and I. S. Bajwa, "A rule based approach for business rule generation from business process models," in *Rules on the Web - RuleML*, 2012, pp. 92–99.

[31] A. K. Dwivedi, A. Gardizy, and S. K. Rath, "Formalization of e-Commerce Patterns using State-based and Event-based Approaches," in *Computing, Communication and Automation (ICCCA2016)*. IEEE, 2016, pp. 127–132.

[32] J. Cabot, R. Pau, and R. Raventós, "From UML/OCL to SBVR Specifications: A challenging transformation," *Inf. Syst.*, vol. 35, no. 4, pp. 417–440, 2010.

[33] K. Kluza and K. Honkisz, "From SBVR to BPMN and DMN models," in *Artificial Intelligence and Soft Computing*, 2016, pp. 453–462.

[34] S. Moschoyiannis, A. Marinos, and P. J. Krause, "Generating SQL queries from SBVR rules," in *Semantic Web Rules - RuleML*, 2010, pp. 128–143. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16289-3_12

[35] L. Gammaitoni and P. Kelsen, "F-Alloy: An Alloy based model transformation language," in *Theory and Practice of Model Transformations (ICMT 2015)*, ser. LNCS, no. 9152, 2015, pp. 166–180.