Validation of Visual Contracts for Services

José D. de la Cruz, Lam-Son Lê, Alain Wegmann

School of Computer and Communications Sciences
Ecole Polytechnique Fédérale de Lausanne
CH-1015 Lausanne, Switzerland

Abstract. Visual modeling languages have specialized diagrams to represent behavior and concepts. This diagram specialization has drawbacks such as the difficulty in representing the effects of services (or actions). We have developed a Visual Contract notation that describes, within one diagram, a service and its effects. This paper presents the semantics of this notation and the verification and validation capability provided by the mapping between our notation and the Alloy lightweight specification language. This validation and verification capability of the Visual Contracts contributes to business/IT alignment.

1 Introduction

One of the important trends in business/IT alignment is service specification. Services can represent the responsibilities of companies in their market, of IT systems in business processes, or of software components in IT systems. Our Visual Contracts provide a graphical representation of services [1]. To be able to verify and validate these service specifications, it is important to define the semantics of the Visual Contracts. This is the goal of this paper. The semantics is defined by providing a mapping to the Alloy specification language [2]. The Alloy Analyzer tool can then be used to verify and validate the specification. For the verification, the modeler tests whether an instance model can be generated from the service specification (i.e. the Visual Contract has no inconsistencies). For the validation, the modeler asks the service’s stakeholder to check if the instance models generated by the service “execution” on the Alloy Analyzer tool correspond to the business needs.

Throughout the paper we will show the example of an IT system (BoardingIT-System) that controls the boarding process of passengers on an airplane. Initially, the aircraft is empty. Then the systems processes the requests from passengers that ask to go onboard; passenger admission requests are processed one by one until the aircraft’s maximum capacity is reached. As a safety rule, none of the passengers can board or disembark more than once.

Section 2 introduces the Alloy specification language. Section 3 defines the Visual Contract primitives and their semantics (in Alloy). Section 4 presents the Visual Contracts for the BoardingITS System example and demonstrates how our approach enables their verification and validation. Section 5 describes the state of the art.

1 This work was partially supported with grants from the CFBE and the EPFL-SOC.
2 Alloy Specification Language and Tools

Alloy Analyzer[2] is a tool for performing automatic verification of software specifications. The Alloy analyzer is built on a SAT prover. As such it can generate instances of models that correspond to a specification (and hence, check that the specification has no inconsistency). It can also check model properties.

A system specification in Alloy is composed of the following parts:
- signatures (keyword “sig”): declares types, sets and relations among types
- facts: global constraints, invariant properties
- functions: parameterized constraints
- assertions: theorems to check, concerning system properties
- commands: run function and/or check assertion

We will visually identify these parts in the definition of semantics of the Visual Contracts. The signatures and facts describe the structural and invariant properties of the system. The functions describe the dynamics of the system. The model checking is done by “executing” the functions and assertions found in the commands section; this will be explained and illustrated further in section 4.2.

3 Visual Contracts

We define Visual Contract (VC) as the visual model that represents both the pre- and the post-conditions for an action, and that makes explicit the changes from the pre to the post state. Our Visual Contract notation is developed in the context of the SEAM notation. SEAM (Systemic Enterprise Architecture Methodology) is a method designed for reasoning about business and IT alignment [3]. A more detailed definition of Visual Contracts and related concepts is found in [1].

3.1 Basic Concepts

In our approach, we model the behavior together with the state of the systems. Systems are modeled as working objects. A working object has information objects (that define possible states), set associations (that define instances) and actions (that change the state of the instances represented on the set associations).

Information Objects. An Information Object (IO for short) is equivalent to a property of the system. It captures the type of the possible states of the observed system. The symbol in figure 1 represents a generic information object.

```
Generic_IO { id: Int,
           attrib1: Int,
           attrib2: Int
// Int (integer) is a // built-in type in Alloy
}
```

Fig. 1. Symbol and Alloy specification for the Information Object Generic_IO.
Each instance of an IO has an identity, encoded in the attribute \textit{id}. As this is a default attribute, we will omit it in diagrams except when absolutely necessary.

By extending our visual notation, we can also represent state information. We show this in figure 2 for the IO \textit{Person} – that will be used in the example—. The bold line around the attribute \textit{boarded} means that its state specification is complete: the only possible states of \textit{boarded} are \textit{offBoard} and \textit{onBoard}.

\begin{verbatim}
<table>
<thead>
<tr>
<th>\textbf{Person}</th>
<th>\textbf{signed BoardingState}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{boarded}</td>
<td>\textit{offBoard}</td>
</tr>
</tbody>
</table>
\end{verbatim}

\begin{verbatim}
sig Person { id: Int, \textit{boarded}: one BoardingState }{ boarded in (OnBoard + OffBoard) }
\end{verbatim}

\textbf{Fig. 2.} Extended notation and Alloy equivalent for the Information Object \textit{Person}. In the extended notation the state information of the IO is included.

\textbf{Set-Associations.} Relationships between information objects are represented as set-associations (SA). The purpose of set-associations is to capture information about instances. The instances of the IOs exist only in the context of SAs.

\begin{verbatim}
set passenger1 = \{ Irina, Angela \} set passenger2 = \{ Gil, Lam-Son \} set passenger3 = \{ HK007 \}
\end{verbatim}

\begin{verbatim}
class Person { state = onBoard }{ \{ Irina, Angela \} \{ Gil, Lam-Son \} }
\end{verbatim}

\textbf{Fig. 3.} UML object diagram for 4 passengers and state diagram for the class \textit{Person}.

We can give an intuitive feeling of our approach, with a UML [4] example (Figure 3) representing 4 people (Irina, Angela, Gil and Lam-Son) who are on the list of passengers of an aircraft. As the state diagram shows, the \textit{Person} can be either \textit{onBoard} or \textit{offBoard} hence two lists are required. This is an implicit requirement of the problem that is not visible in the UML specification (figure 3).

In UML, it is not clear how to instantiate the state information of objects. However, we just saw that we must include state information in order to differentiate the lists of \textit{passengers} – one for \textit{offBoard} (at the dock), and another for \textit{onBoard} (already on the plane). Two of the options to represent the instance (object+state) information in one diagram are shown in Figure 4.

\begin{verbatim}
Aircraft passenger_list \{ state = onBoard \} \{ Irina, Angela \} \{ Gil, Lam-Son \} Person
\end{verbatim}

\begin{verbatim}
Aircraft passenger_list \{ state = offBoard \} \{ HK007 \} Person
\end{verbatim}

\textbf{Fig. 4.} SEAM representation of 4 passengers using (a) instance identifiers and (b) instance cardinalities + explicit IO state information.

\textbf{Actions.} An \textit{action} is the modeling element that specifies the effects of system’s reactions to a set of events for a given system state. It is equivalent to a service.
Contexts of Existence. We have already said that all SAs constitute the context where instances actually exist. This adds a temporal frame for reasoning about the system and, in particular, what happens when an object is deleted. We distinguish two cases:

- **Tight binding** (filled diamond, as in figure 4): the instances actually exist in the SA, and whenever the SA (or the referencing IO) is deleted, the instances are deleted, too.

- **Loose binding** (hollow diamond): the instances are only referenced. This means that when the SA is deleted (or the referencing IO), the actual instances are not deleted but their references do.

Parameters. We define a parameter as a special information object used either to communicate through the boundary of the system or among action boundaries. Every parameter is either consumed by another action or sent to the environment. A parameter is represented by a stereotyped IO. The stereotype label can be «Par In» or «Par Out».

Select Operator. Very often we will have to select subsets from an original set. The «select» operator allows us to do this. It describes visually how a set (set-association) is created from another SA. It is drawn as a box containing a predicate (filter), linking the source set, the filter set and the target (resulting) set, as shown in figure 9.

3.2 Building Correct-by-Construction Systems

First, by introducing state information, the modeler extracts additional requirements. For instance, we elicited a hidden requirement that says that there are two lists of passengers (and two lifecycles): one for onBoard, and another for offBoard.

Second, the reasoning process takes into account lifecycles because we make explicit the contexts of existence. In figure 5, the system itself is represented —via the IO Myself—since it permits establishing how many instances of each IO are present in a single system. Thus, one Plane exists in the context of BoardingITSystem. Then, 2 lists of Person (passenger) exist in the context of a single Plane (i.e. SAs onBoard_passenger_List and offBoard_passenger_List). Last, multiple instances of Person may exist in the context of each Passenger_List.

Third, since almost all IOs correspond to physical systems, when reasoning we can take into consideration the fact that the sets must be limited by the capacities of IOs. This is why the Plane (IO in the IT system) has an attribute Capacity.

![Fig. 5. SEAM notation for property definitions in the BoardingITSystem specification.](image-url)
Modeling Change via the Set-Associations. Our set-associations evolve over time by changing their states (number of IOs and state of IOs in the set). We distinguish three main classes of SAs:

- **IO-IO**: as described until now, it makes explicit relationships between IOs.
- **Action-IO**: it allows describing instances that exist only during an action.
- **SA-SA**: equivalent to a simple relational algebra among SAs (sets).

To give an intuition of state change of set associations, let us model a system that has one instance of `Person` that is `offBoard` at the beginning ([@pre]) of an action `op1`; at the end ([@post]) of the same action, the system has one instance of `Person` in state `onBoard`. Figure 6 shows the UML way to represent this scenario. Part `a` corresponds to the pre-condition ([@pre]) and part `b` corresponds to the post-condition ([@post]). Figures 6.a and 6.b are identical. No change is noticeable.

We argue that this happens because UML diagrams are not tightly coupled. The action and the object instances are linked implicitly. A textual OCL document may be added to the model in order to make explicit the changes due to action `op1`.

![Fig. 6. UML object, activity, state and class diagrams for `a)` before and `b)` after action `op1`. The effect of action `op1` is not evident.](image)

Compare figures 6 and 7. Figure 7 is the Visual Contract of action `op1`. It shows the action and the objects in a single diagram (tight coupling). We can now write assertions about this action and the configurations of objects. The brackets identify the predicates that are necessary conditions (pre-conditions) for the changes to take place.

Initially, the SA `offBoard_passenger_List` includes one instance of IO `Person` that is `offBoard`. At the end, the same `Person` IO instance is `onBoard`. Figure 7 summarizes this state change using the «change» operator (`Æ`) for cardinalities and a «transfer» operator (curved, wide arrow) between lists. However, we need to provide a mechanism to indicate what instances should be affected (changed/transferred) for complex cases (i.e. the `offBoard_passenger_List` consists of more than one instance of `Person`). This will be discussed in section 4.1.

![Fig. 7. Action changes the cardinality of both SAs for `passenger` lists. A transfer has been made, meaning also a change of state of the respective instances of IO `Person`.](image)
4 Verification, Simulation and Validation of the Specification

In this section, we describe how to “execute” the services specified via the Visual Contracts. The instance models resulting from the verification can then be used by the stakeholders to validate the services.

4.1 Visual Contracts for Actions Init and Board

Figure 8 shows the SEAM Visual Contract for operation Init. It states that: “The aircraft is initially empty”; in other words, the number of Person in the onBoard_passenger_List is set to zero, as shown in the VC in figure 8. This means that the cardinality of SA onBoard_passenger_List goes from some initial value (any, symbolized by the character ‘*’) to 0.

The operation Board is more complex. The specification is the following: “The preconditions are: a) an input parameter represents the person that desires to go on board, b) this person is not on the list of persons that are already onboard, and c) the number of people onboard has not reached the maximum capacity of the aircraft. The post-condition are a) this person is now onboard, b) the system emits a message confirming the entry of this person onto the plane”.

The Visual Contract for Board is shown in figure 9. The intermediate processing is kept in the final contract in order to make the changes more understandable: the passenger id is validated and the instance is selected via the SA selected. Note that the constraint regarding the aircraft capacity (precondition c) is the guard for transferring—and for changing the state of—the instances from SA selected.
4.2 Alloy code and Analysis for Actions Init and Board

In order to write an Alloy equivalent of the Visual Contract for action Board, we have to introduce the notion of time.

Modeling time. We model time as an ordered vector. Time enables differentiating the model instances before and after every action is executed. Figure 10 shows the Visual Contract code of action Board already modified for modeling time.

```
open util/ordering[Time]
sig Time { }
sig BoardingState {}
one sig OnBoardState, OffBoardState extends BoardingState {}
sig Id_Person {}
sig Person {
id: Id_Person,
state: BoardingState one -> Time
}
fact uniqueID { all p, q: Person | p != q <=> p.id != q.id }
one sig Plane {
capacity: Int,
onboard_passenger_List: set Person -> Time,
offboard_passenger_List: set Person -> Time
}
all t: Time | all p: Person | p in onboard_passenger_List.t => p.state.t = OnBoardState
all t: Time | all p: Person | p in offboard_passenger_List.t => p.state.t = OffBoardState
all t: Time | int capacity >= #onboard_passenger_List.t + #offboard_passenger_List.t
all t: Time | no p: Person | p in onboard_passenger_List.t and p in offboard_passenger_List.t
all t: Time | all p: Person | p !in onboard_passenger_List.t => p in offboard_passenger_List.t
}
pred Board (ps: set Id_Person, a: one Plane, pre, post: one Time) {
pre != post
// pre-condition
all pid : ps | one p: Person | p.id = pid and p in a.offboard_passenger_List.pre
// post-condition
let psl = { p: Person | p.id in ps } |
// onboard list increased
a.onboard_passenger_List.post = a.onboard_passenger_List.pre + psl and
// offboard list decreased
a.offboard_passenger_List.post = a.offboard_passenger_List.pre + psl
}
```

**Fig. 10.** Alloy equivalent of the Visual Contract for action Board.

Verification. In order to create a complete basic scenario we must ask the Alloy Analyzer to do an action Init followed by one or several actions Board (action Init_and_Board), as shown in figure 11. It is essential to guarantee a minimum size of the state space, in order to avoid the trivial solutions.

The Alloy Analyzer tool creates all the model instances that can be generated from this scenario. First, the Alloy Analyzer tool will generate all the instances in the defined scope. Second, it will verify the Visual Contracts by finding all generated instances that are inconsistent with the VCs under study. Last, if any instance is found to be inconsistent (i.e. two or more constraints are incompatible or contradictory), the Alloy analyzer can generate counterexamples that illustrate the inconsistencies.
pred Init (a: one Plane, post: one Time) {
    no a.onboard_passenger_List.post
    // at least 3 people are offboard at the end of action Init
    #a.offboard_passenger_List.post > 2
}
pred Init_and_Board (ps: set IDPerson, a: one Plane, pre, post: one Time) {
    #ps = 2
    post = next(pre)
    Init (a, pre) and Board (ps, a, pre, post)
}
run Init_and_Board for 5 but 2 BoardingState, 2 Time

**Simulation and Validation.** The Alloy Analyzer will explore the constrained state space of instances for each object as indicated in the last line of the code in the figure 11. The tool will execute the sequence (action Init followed by action Board) and check that all the instances in the scope respect the constraints.

We extract the snapshots since they allow us simulating the behavior of the system in time. Each of the time points correspond to one point in a vector. We project the resulting instance models in time, in order to retrieve an ordered sequence of snapshots.

![Simulation and Validation](image)

**Fig. 12.** Results of simulating the action Board in the Alloy analyzer. At Time_0 the passengers (Person0, Person1, and Person2) have already checked-in. At Time_1 two of them (Person0, Person1) have effectively embarked on the plane. Capacity of the plane is 3.

Finally, it is the user who validates the service: she can see the results of this simulation and decide whether the behavior is adequate or not. Two resulting snapshots of the simulation are shown in figure 12. These instance models can now be discussed with the stakeholders to validate the service specification.

**5 Related Work**

UML [4] is the industry standard in object-oriented analysis and design. UML provides a rich set of notations. Relating these notations is often the responsibility of the designer. But its diversity is a potential source of misunderstandings and errors in the implementation of the system [5, 6]. In our approach, we try to minimize this problem.
by representing conceptual and behavioral specifications in one diagram. For this, we extend the initial proposition of [7] to include state information in the class diagrams.

The use of contracts for object oriented (OO) programming approaches [8, 9] was followed by the use of contracts in OO methods [10, 11], using a textual notation for the logics. Only recently [12, 13] have independently proposed the concept of “visual contracts”. Work by [12] focuses on the generation of executable code that can perform run-time verification of constraints; our approach is declarative, allowing us to execute and evaluate very high-level model specifications.

Notations used for workflow systems [14, 15] represent specifications of actions mainly in natural language; therefore, they are user-friendly but not directly translatable in terms of what changes in the information system (IS). Our approach is also visual but attempts to make explicit the effects of services in terms of the supporting IS, facilitating the subsequent automation and setting a more concrete reasoning framework.

Finally, a Visual Contract cannot be compared to an UML snapshot because a Visual Contract is generic: It can be applied to a whole family of snapshots and even scenarios. Furthermore, the similarity with other diagrammatic works—such as the graph transformation languages—is only apparent since their foundations are orthogonal to our systemic approach, and our goal is to add semantic content to visual specs.

6 Conclusions and Future Work

Business/IT alignment requires specifying services. We propose SEAM Visual Contracts as a means to specify services. The verification and validation requires either testing (manual and not complete) or model checking (automatic and theoretically complete). We propose a mapping between our visual contracts and Alloy, a lightweight formal language. Thanks to this, it is possible to use the Alloy Analyzer as model checker in order to verify, validate and even simulate the Visual Contracts.

The main advantages of SEAM Visual Contracts are:

• They are not IT specific. They can represent services of companies, departments, IT systems, etc [3]. This makes them applicable as well to business processes [15] specification.

• They can be validated and verified at a high level of abstraction. This reduces the time when compared to testing specific, full-fledged implementations[2].

• They are UML-like [4]. Thus, UML practitioners may adopt them quickly.

Our future work includes defining how to express the contexts of existence, developing CAD tool support [16] for automating the translation process, and validating the approach usability in concrete projects. At present we are developing a tool for translating UML specifications to VCs and evaluating a technique to refine VCs.

References