Declarative Specification and Alignment Verification of Services in ITIL

Irina Rychkova[1], Gil Regev[2], Alain Wegmann[3]
Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Communication and Computer Science, CH-1015 Lausanne, Switzerland
{[1]irina.rychkova@epfl.ch, [2]gil.regev@epfl.ch, [3]alain.wegmann@epfl.ch}

Abstract

IT organizations that wish to implement the best practices recommended by the IT Infrastructure Library (ITIL) need to specify the level of service provided to their customers. The implemented service needs to comply with this specification. We propose a method for describing declarative specifications of services, and of their planned constructions. These specifications can then be used to verify the alignment between the specification and the construction. This method is based on first-order logic and on refinement theory for alignment verification. The alignment verification is automated using the Alloy specification language and the Alloy Analyzer tool. The method is illustrated with the example of a utility emergency service inspired from a real project.

1 Introduction

The Information Technology Infrastructure Library (ITIL) is a collection of good practices for the management of IT services. The perceived value of ITIL is the improvements of the relationship between the business and its IT service providers. The relationship between the business and its internal IT department is defined with the use of a Service Level Agreements (SLA). Similar agreements define the relationships between sub-departments of the IT department (Operational Level Agreements, OLA) and between the IT departments and its external providers (Underpinning Contract, UC). For the IT department to be able to live up to its obligations defined in the SLA, it has to make sure that the SLA is implementable with the existing and envisioned infrastructure and with its OLAs and UCs. In this paper we propose a formal method for specifying SLAs, OLAs and UCs and for verifying the alignment between an SLA and a set of OLAs that implement this SLA. Alignment verification is considered as important by the design teams: when a design team negotiates what an IT service should do (i.e. negotiate the SLA), the team needs to understand how they will implement the service (i.e. they define the relevant OLAs). Not checking the alignment between the SLA and the OLAs can lead to the specification of services that cannot be implemented.

The method we present is based on SEAM [1]. SEAM is a visual modeling language designed to model business and IT system. We illustrate our method with a concrete ITIL project currently in progress. This project is done for the public utility of Geneva: SIG (http://www.sig-ge.ch/) in collaboration with the consulting company Itecor and the EPFL University. Whereas we are inspired by this real project, the formal techniques we describe in this paper are in their early development, we have therefore substantially simplified the actual processes. To account for the fact that the example is an academic illustration only, we use the name City Industrial Service to refer to the utility company.

We illustrate first how an IT service can be specified with a visual model; this specification corresponds to the ITIL SLA. We then show how the planned implementation of the IT service can also be specified using the same notation; with this we define a set of OLAs. The challenge is then the verification of the alignment of these specifications (i.e. of the SLA with the set of OLAs). For this, we define a formal semantics for our notation, and we map this semantics to a specification language called Alloy (http://alloy.mit.edu). To be able to check the alignment between two specifications in general (and between the SLA and the OLAs in particular), we reduce the problem of alignment verification between visual specifications to the problem of verification of refinement correctness between specifications written in Alloy. We implement this verification with the Alloy Analyzer – a tool for analysis of specifications written in the Alloy specification language.

In Section 2, we present an example of SLA and the corresponding OLAs specified using SEAM. In Section 3, we present the formal semantics for the SEAM notation, the mapping of visual specifications to Alloy, and how the alignment is validated using the Alloy Analyzer tool. In Section 4, we present the related work.

Presented at First International Workshop on Dynamic and Declarative Business Processes (DDBP) in 12th IEEE International EDOC Conference (EDOC2008), Muenchen, Germany, September 15-19, 2008

2 Case study: Gas Incident Service

The City Industrial Service provides, among other services, water, gas, and electricity to Geneva residents. One of the important responsibilities is the management of gas incidents, i.e. leaks from gas machinery or pipes. The gas incident service is specified as: "The CIS has to neutralize a gas escape reported by a witness. The CIS shall guarantee that, if the incident site is not secured within 45 min from the witness call registration by the CIS operator, an emergency call is made to the local Fire Brigade. An IT system monitors the process and calls automatically the Fire Brigade when the deadline is reached. The IT system needs to log the incidents

The IT department provides the IT system `IT_GasIncident`. This IT system provides the service `GasIncidentS` support for this responsibility, it is the IT service called `GasIncidentService`. This service is provided by. When we specify the SLA of a service and the OLAs of a service implementation, we model twice the IT system: once as a whole and once as a composite. When modeled as a whole, we analyze the IT system as a black box and we specify the service provided by the system, i.e. the SLA (Section 2.1). When modeled as a composite, we analyze the construction of the IT system, i.e. the OLAs (Section 2.2).

2.1 Service Specification

Fig. 1 describes, in SEAM, the service provided by the IT system: the `ITGasIncident_w` (the postfix "w" indicates that the IT system is represented as whole) provides a service `LA_GasIncidentService` (the prefix "`LA`" indicates that this is a localized action – an action executed by a system alone). The IT system manages an `incidentList` and a `locationList`. The `incidentList` contains a list of `Incident`. Each incident has characteristics such as `callTime`, description, etc… The `LA_GasIncidentService` has an association to an Incident (referenced as `newInc`). This association refers to the new incident that is created when a witness calls. The service is also associated with input and output parameters. For example, `WitnessCall` represents the information provided by the witness when she notices a gas escape. The parameter `SiteSecuredTime` represents the event that indicates to the IT system that the situation is secured. The `EmergencySignal` is the event sent to the Fire Brigade if the situation is not secured after 45 minutes.

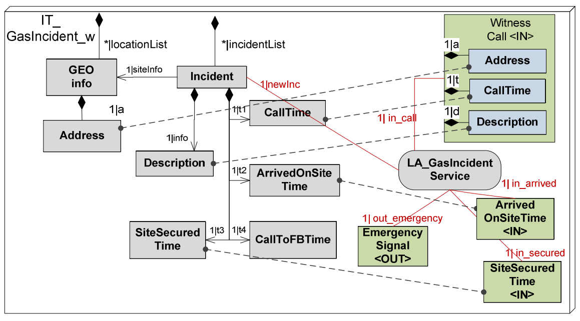

Figure 1: SEAM specification of the service LA_GasIncidentService (ITIL SLA)

As explained, the service modifies the properties of the system. For example, when the service creates a new incident, `newInc`, it sets `t1` (i.e. `CallTime)` of `newInc` with the parameter `a` (i.e. `CallTime)` from the `WitnessCall`. These changes are described with *preconditions and postconditions*. The precondition specifies the system state for the action to be executed; the postcondition specifies the result of the action. To express preconditions and postconditions, we use a specification language called *Alloy* (http://alloy.mit.edu/). Alloy is a declarative specification language that will be further described in Section 3. The preconditions and the postconditions are specified for all services. An example of a postcondition written in Alloy is:

```
((newInc.t3 - newInc.t1 <= 45)
   and (out_emergency=0)) or
  ((newInc.t4 = newInc.t1 > 45)
     and (out_emergency=1))
```

This postcondition specifies that if `newInc.t3` (i.e. the `SiteSecutedTime` – see Fig. 1) minus newInc.t1 (i.e. the `CallTime` – see Fig. 1) is smaller or equal to `45`, then `out_emergency=0` which means that no emergency signal is sent. Otherwise, an emergency signal is sent. We use the syntax of Alloy in the expression. Using this language, the service can be fully specified in a declarative style (see Fig. 5).

2.2 Service Construction

Once the service is specified, it is important to understand how the service can be implemented. To do so, the design team analyzes the IT system as a composite.

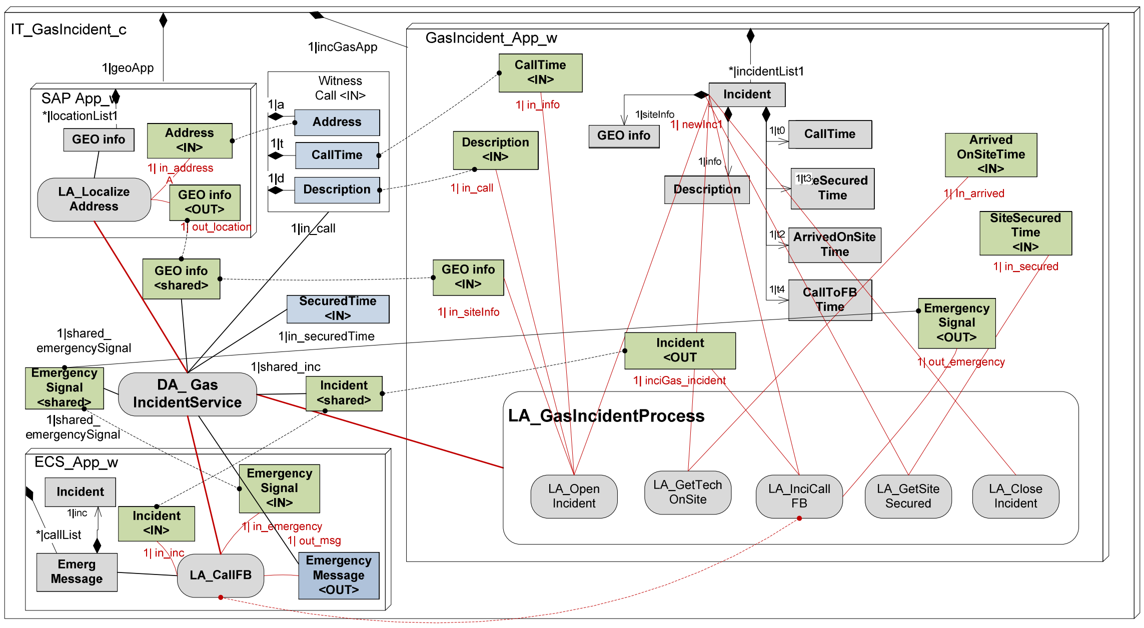

Figure 2: Service implementation modeled as SEAM distributed action

This is illustrated in Fig.2 that represents `ITGasIncident_c` (the postfix "c" means that the IT system is represented as a composite). The goal is now to understand how the SLA that defines the service is implemented by OLAs that define the responsibilities of each application that constitutes the IT system. Concretly, `IT_GasIncident_c`, which describes the planned construction of `IT_GasIncident_w`, has three component applications: (1) `SAP_App`, the SAP application, which processes the data from the help desk and provides the CIS operator with the GPS coordinates of the site; (2) the `ECS_App` application (Emergency Call Service), which provides an automated call service to the local fire brigade; and (3) `GasIncident_App` application that coordinates the incident processing, triggers the call to the fire brigade after and maintains the incident record in the incident list. Specifications of the services offered by these applications correspond to Operational Levels Agreements (OLAs).

The action `DA_GasIncidentService` specifies how the responsibilities in the incident securing are distributed between the applications (the prefix "DA" indicates that this is a distributed action – an action executed by multiple systems that collaborate). The shared properties attached to `DA_GasIncidentService` represent the kind of information shared by the systems; it is a declarative representation of the information exchange between them.

`GasIncidentService` is a **declarative process specification** that defines the conditions and the results of the process but does not impose any constraints on how this process has to be conducted in a particular environment. It is the implementation of the service `LA_GasIncidentService` (Fig. 1). The `LA_GasIncidentProcess` action specifies the responsibility of the `GasIncident_App` application and is also modeled declaratively: we describe the actions to execute but not the control flow.

Let's illustrate how an incident is processed. `WitnessCall` represents the input parameters to the system. The behavior of all the systems is specified as follows: In `GasIncident_App`, the `Gas_IncidentApp` controls the process. In `SAP_App`, the `Address` is transformed in geographical information (`GeoInfo`). The `ECS_App` is responsible to convert an `Incident` into an `EmergencyMessage` (sent to the Fire Brigade) when an `EmergencySignal` is received.

As explained in Section 2.1, the visual specification is not sufficient to fully specify the behavior. For this reason, we have – for each action – an Alloy description of the preconditions and the postconditions of the action. For lack of space, we do not provide an example of these descriptions in this paper.

The SEAM specification of `LA_GasIncidentService` in Fig. 1 corresponds to the SLA. The SEAM specification of `DA_GasIncidentService` together with `LA_GasIncidentProcess`, `LA_LocalizeAddress` and `LA_CallFB` in Fig. 2 show the implementation of this SLA by the three applications. The specification of the service offered by each application is an OLA. The transition from the specification of the SLA (Fig. 1) to the specification of the multiple OLAs (Fig. 2) is a result of the **specification refinement**. We consider that the SLA and the OLAs are aligned when the refinement is correct. Intuitively, this is relatively obvious: input and output parameters should be the same between the SLA and the combined OLAs, the preconditions and the postconditions should be compatible, etc… In this paper, our goal is to illustrate how we can define formally this notion of alignment. The refinement relation that we introduce in Section 3 makes explicit this relation between the SLA and the OLAs.

3 Alignment and Semantics of Visual System Specifications

We claim that the service construction (OLAs) is aligned with its specification (SLA) if the service construction represents a correct refinement of the service specification. This is the notion of the **correct refinement** adopted from software engineering [4][5][6].

To mechanically check if the refinement is correct, we proceed as following:

We provide a formal semantics for SEAM based on first-order logic (FOL) and set theory. This semantics specifies as a set of mapping rules that transforms the SEAM specifications of the SLA (Fig. 1) and of the OLAs (Fig 2) into predicated written in Alloy (note that the SEAM specification already includes annotations written in Alloy; using these annotations we specify the preconditions and the postconditions). The correspondence between the SEAM specification and Alloy is illustrated in Fig. 3: P1 is the translation in Alloy of the SLA, P2 is the translation in Alloy of the OLAs. We can verify the refinement correctness between P1 and P2 using the Alloy Analyzer tool (http://alloy.mit.edu/). We use the Alloy Analyzer to validate that the model of the service construction **does not violate** but **does simulate** the model of the service specification. This corresponds to the alignment verification.

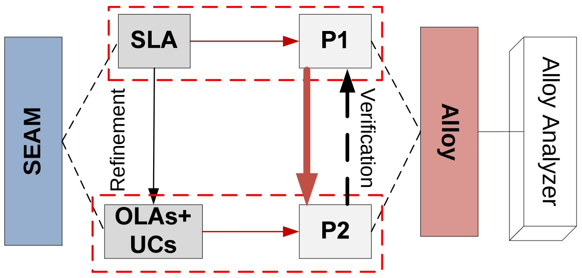

Figure 3: Refinement verification

3.1 First Order Logic Semantics for the SEAM Notation

First, we define the concept of working object that represents the IT system in the SEAM model. Note that working objects are also used to represent other kinds of systems (such as companies, value system, etc…).

For every action A of a working object we define a precondition and a postcondition. **Postcondition** A_{post} is a condition that a working object meets after the action termination. **Precondition** A_{pre} specifies a condition that must hold upon the action execution: *If A is started in a state satisfying A_{pre}, it is guaranteed to terminate in a state satisfying A_{post}*

Precondition and postcondition are modeled as predicates over state space Σ :

$$A_{pre} : \Sigma \to \{true,\ false\},$$
$$A_{post} : \Sigma \times \Sigma \to \{true,\ false\} \qquad (1)$$

The precondition of the action A specifies a set of states of a working object, where A is applicable. This set is called the set of pre-states for A; it represents a subset of a state space $\sum$ of the working object and denoted: $\Sigma_{A_{pre}} \subseteq \Sigma$. A state $\overline{X}$ of the working object satisfies the precondition of the action A if and only if it belongs to the set of pre-states of A:

$$\forall \overline{X} \in \Sigma \mid A_{pre}(\overline{X}) \Leftrightarrow \overline{X} \in \Sigma_{A_{pre}} \qquad (2)$$

A postcondition of the action A defines a relation between the states of a working object before and after this action respectively. A set of action post-states $\Sigma_{A_{post}}$ is defined as all states $\overline{X}'$ of the working object after the action termination and can be denoted as follows:

$$\forall \overline{X}' \in \Sigma \mid \forall \overline{X} \in \Sigma_{A_{pre}} \mid A_{post}(\overline{X}, \overline{X}') \Leftrightarrow \overline{X}' \in \Sigma_{A_{post}} \qquad (3)$$

Here $\overline{X}$ is a pre-state of A.

Action A defines a transition of the working object from state $\overline{X}$ to state $\overline{X}'$ (pre- and post-states respectively). We define a SEAM action as a binary FOL-formula $A:\Sigma\times\Sigma\rightarrow\{true, false\}$. We specify the SEAM action using logical implication between precondition and postcondition:

$$A(\overline{X},\overline{X}'): A_{pre}(\overline{X})\rightarrow A_{post}(\overline{X},\overline{X}') \quad (4)$$

If at a given state $\overline{X}$ the precondition A_{pre} of the action A holds, then the working object will be transited to a state $\overline{X}'$, for which the postcondition of A - A_{post} - holds.

For actions with invariants we write:

$$A(\overline{X},\overline{X}'): S_{inv}(\overline{X})\wedge A_{inv}(\overline{X})\wedge A_{pre}(\overline{X})\rightarrow$$
$$A_{post}(\overline{X},\overline{X}')\wedge A_{inv}(\overline{X}')\wedge S_{inv}(\overline{X}')$$

3.2 Refinement

We formulate the problem of refinement verification using a theory of Data Refinement [5][6].

The relationship between the service specification and the service construction is captured by the notion of **refinement**, adopted from software engineering [10][4]. In software engineering, a program specification development is considered as a sequence of refinement steps, leading from the abstract specification towards its implementation. Along those lines, SEAM model development can be considered as a stepwise refinement of its graphical specifications [2]. More precisely, refinement in SEAM specifies a transition from one organizational level, where the working object is presented as a whole, to another organizational level, where the same working object is presented as a composite. A specification of a working object as a whole is usually called '***abstract***' and a specification of a working object as a composite – '***concrete***'. We say that *the concrete specification refines the abstract specification*. A relation between the state spaces of the abstract and the concrete working objects is called a ***refinement relation***.

Verification of refinement is largely based on the use of simulation techniques [7]. By the **simulation** we understand a correspondence between the states of two systems, abstract and concrete, where the concrete system is considered an implementation and the abstract system – its specification.

Let us consider a working objects W seen as a whole, and specified on the state space Σ_a with a localized action A_a, and a working object W', seen as a composite, and specified on the state space Σ_c with a distributed action A_c.

Definition 1: Given a refinement relation between state spaces, *W'* is called a ***correct refinement*** of *W* if and only if for each run of the $R:\Sigma_a\times\Sigma_c\rightarrow\{true, false\}$ concrete action A_c of *W'*, which starts at $\overline{X}_c\in\Sigma_c$ and terminates at $\overline{X}_c'\in\Sigma_c$, there exists a run A_a of *W*, which starts at $\overline{X}_a\in\Sigma_a$ such that $R(\overline{X}_a,\overline{X}_c)$ holds and terminates at $\overline{X}_a'$, such that $R(\overline{X}_a',\overline{X}_c')$ holds.

This refinement is illustrated in Fig. 4. A_c correctly refines A_a if, when A_c makes a transition from its pre-state $\overline{X}_c$ to its post-state $\overline{X}'_c$, A_a is also making a transition from its pre-state $\overline{X}_a$ to its post-state $\overline{X}'_a$, and these states are related by R.

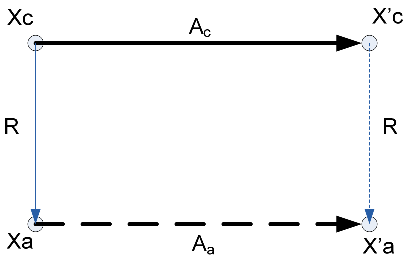

Figure 4: The refinement in SEAM

This definition can be expressed as follows:

$$R:\Sigma_a\times\Sigma_c\rightarrow\{true, false\};$$
$$\forall\overline{X}_c,\overline{X}'_c\in\Sigma_c,\overline{X}_a\in\Sigma_a\mid$$
$$\left(R(\overline{X}_a,\overline{X}_c)\wedge A_c(\overline{X}_c,\overline{X}'_c)\right)\Rightarrow\exists\overline{X}'_a\in\Sigma_a\mid \quad (5)$$
$$A_a(\overline{X}_a,\overline{X}'_a)\wedge R(\overline{X}'_a,\overline{X}'_c)$$

if refinement relation is a total function $R:\Sigma_c\rightarrow\Sigma_a$, we rewrite (6):

$$\forall\overline{X}_c,\overline{X}'_c\in\Sigma_c\mid A_c(\overline{X}_c,\overline{X}'_c)\Rightarrow \quad (6)$$
$$A_a(R(\overline{X}_c),R(\overline{X}'_c))$$

The expression in Eq. (6) will be validated with the Alloy Analyzer tool.

3.3 Mapping to Alloy Specification Language

Alloy is a declarative specification language developed by the Software Design Group at MIT. Alloy is a language for expressing complex structural constraints and behavior based on first-order logic (FOL). Therefore, SEAM specifications, formalized in FOL, can be directly mapped to Alloy.

We map SEAM specifications, formalized in FOL, to the Alloy specification language [3] and so it is possible to check the refinement automatically.

SEAM actions, formalized as first-order formulae, with their preconditions, postconditions and invariants, formalized as predicates are mapped to Alloy **predicates**. Action parameters are mapped to **parameters of predicates** in Alloy.

We specify abstract and concrete actions performed by a system as Alloy predicates with variables representing the system states before and after the action. We define refinement relations between system states of the abstract and concrete specifications as Alloy **predicates.** We validate the refinement correctness from Definition 1 specified as Alloy **assertion**.

Based on the formal semantics, visual specification of a service and its planned construction can be represented as Alloy data structures with a behavior expressed as predicates. The assertion that one behavior can be always simulated by the other (a correct refinement) is expressed as an Alloy assertion. Technically, verification of refinement correctness between two visual specifications is reduced to **a proof of validity** of this assertion.

3.4 Refinement Verification in Alloy Analyzer

To proceed with the specification analysis and alignment verification, we map the SEAM visual specifications to Alloy. Figure 5 illustrates the result of translation of the `LA_GasIncidentProcess` (Fig.1) to Alloy specification language.

In mapping the SEAM specification to the Alloy specification language, the annotations made to the diagrams are used to specify the action in Alloy

In Fig. 2, lines 1-2 defines the Alloy signature that specifies the action, line 3 specifies the action precondition, and lines 4-13 specify the action postcondition.

An Alloy predicate specifies a logical formula. Executing this predicate, Alloy Analyzer decides whether the formula is satisfiable. Mechanically, Alloy analyzer attempts to find a binding of the variables to values - that makes the formula true. For example, the output trace for the execution of a distributed action specification `DA_GasIncidentService` looks as follows:

```
Executing "Run DA_GasIncidentService for 5"
Solver=sat4j Bitwidth=4 MaxSeq=5 Symmetry=20
8632 vars. 1035 primary vars. 22316 clauses. 238ms.
Instance found. Predicate is consistent. 92ms.
```

```
1. pred LA_GasIncidentService [incidentList_pre,
incidentList_post: set Incident,
locationList_prepost: set GEOInfo, in_call:one
WitnessCall, in_securedTime: one Int,
2.out_emergency: one Int, out_incident: one
Incident] {
3.  ((in_call.t >0 )and (in_securedTime >0)) =>
4.(one newInc: Incident | //fresh inc created
5.(!(newInc in incidentList_pre)) and //Added to the
list:
6.(incidentList_post = incidentList_pre + newInc )
and
//Initial values from the witness call:
7.(newInc.t1 =  in_call.t) and (newInc.info =
in_call.d) and
//GPS data is obtained from the Address
8.(one loc: GEOInfo | (loc in locationList_prepost)
and
9.(loc.a = in_call.a) and (newInc.siteInfo = loc))
and
//secured time as an income call from the technician
10.((newInc.t3 =  in_securedTime))and
 //either the site is secured within 45 min or
emergency sent
11.(((newInc.t3 - newInc.t1 <= 45) and
(out_emergency=0)) or
12.((newInc.t4 = newInc.t1 > 45) and
(out_emergency=1))) and
13.(out_incident = newInc))}
```

Figure 5: Alloy specification of the localized action LA_GasIncidentService.

To relate the SLA - with the combination of OLAs, we have to guarantee the correct refinement from the localized action `LA_GasIncidentService` to the distributed action `DA_GasIncidentService`. In our example, by the Definition 1, we consider the distributed action `DA_GasIncidentService` as the concrete specification and the localized action `LA_GasIncidentService` as the abstract specification. We rewrite (5) as an Alloy **assertion** that stands the correct refinement.

`R_LA_to_DA` (Fig. 6) is a refinement relation that relates state spaces of the `IT_GasIncident_w` and `IT_GasIncident _c`. `R_Input` and `R_Output` are relations between input and output parameters respectively.

Checking the `LA_DA assertion` (Fig. 7), the Alloy Analyzer tries to find a counterexample – a set of variables of the model that will falsify this assertion. Given our specification, Alloy Analyzer produces the following log:

```
Executing "Check LA_DA"
   Solver=sat4j Bitwidth=4 MaxSeq=4 Symmetry=20 6141
vars. 630 primary vars. 17014 clauses. 416ms.
   No counterexample found. Assertion may be valid.
168ms.
```

At a given test space, our assertion is valid, i.e. the distributed action that specifies the service construction correctly refines the service specification modeled as the localized action.

If the analyzer finds no counterexample – this means that the assertion may be valid (i.e. it is valid for a limited scope considered by the analyzer but none knows if it is valid out of this scope). To prove refinement correctness, the same assertion can be examined by theorem provers. Successful case here (i.e. when prove is found), means assertion validity.

```
pred R_LA_to_DA[incidentList_t: set Incident,
locationList_t: set GEOInfo, mList_t: set
EmergencyMsg,  // model concrete
incidentList1_t: set Incident,
locationList1_t: set GEOInfo]{ // model abstract
 ( incidentList_t=  incidentList1_t) and
 ( locationList_t=  locationList1_t)}

pred R_Input[in_call: one WitnessCall,
in_call1: one WitnessCall, in_securedTime,
in_securedTime1: one Int ]{
in_call = in_call1 and
in_securedTime = in_securedTime1}

pred R_Output[out_emergencyCall: EmergencyMsg,
out_Incident: one Incident, out_emergency: one
Int]{
(out_emergency = 1) =>
(out_emergencyCall.inc = out_Incident )}
```

Figure 6: Alloy specification of a refinement relation between the localized and the distributed actions.

```
assert LA_DA{
all incidentList_pre: IDB, mList_pre: MSG,
in_call: WitnessCall, in_securedTime: Int,
incidentList1_pre: IDB, in_call1: WitnessCall,
in_securedTime1: Int|
all incidentList_post: IDB, mList_post: MSG,
out_emergencyCall: EmergencyMsg,
locationList_prepost: GEO,
 incidentList1_post: IDB, out_Incident:
Incident, out_emergency:Int,
locationList1_prepost: GEO|
(DA_GasIncidentService[incidentList_pre.v,
incidentList_post.v, locationList_prepost.v,
mList_pre.v, mList_post.v, in_call,
out_emergencyCall, in_securedTime] and
R_LA_to_DA[incidentList_pre.v,
locationList_prepost.v, mList_pre.v,//model
concrete
incidentList1_pre.v, locationList1_prepost.v]
and R_Input[in_call, in_call1, in_securedTime,
in_securedTime1])=>
(LA_GasIncidentService_w[incidentList1_pre.v,
incidentList1_post.v, locationList1_prepost.v,
in_call1, in_securedTime1, out_emergency,
out_Incident]
and R_LA_to_DA[incidentList_post.v,
locationList_prepost.v,
mList_post.v,incidentList1_post.v,
locationList1_prepost.v] and
R_Output[out_emergencyCall, out_Incident,
out_emergency])}
```

Figure 7: Alloy assertion that stands the correct refinement from abstract to concrete specification based on Definition 1.

4 Related Work

There are two main approaches to formal verification: model checking [11] and a theorem proving based on logical inference [12]. Model checking is an approach for verifying requirements and design for a vast class of systems, including real-time embedded and safety-critical systems. Model checkers include such tools as [9][13]. The major drawback of the model checking is a state explosion problem, which originates from the fact that for real systems the size of the state space grows exponentially with the number of processes [14].

The second approach is an automated theorem proving based on logical inference. Within this approach, the fact that the system specification (a model) satisfies a certain property is expressed as a logical formula. The task is to prove the validity of this formula, deducing it from a set of axioms exist for the underlying logic (e.g. first-, second-, higher-order logic etc), and hypotheses made about the system. Theorem proving for the first-order logic is well developed and widely represented in the literature (see for example [15]). Higher order and other logics are more expressive and appropriate for wider range of problems then first-order logic; however the automated theorem proving for these logics is more complicated [16].

In our method we can use both techniques. The paper presents an implementation based on model checking.

Visual modeling methods discussed below, share common concepts with SEAM, i.e. hierarchical structure of models. These methods also specify the semantics of transitions between their hierarchical levels similarly to refinement in SEAM. Although, up to our knowledge, none of these approaches uses the formal specification languages to provide an automated analysis of specifications.

Design & Engineering Methodology for Organizations (DEMO) [17] is an EA framework based on the organizational theory called Language/Action Perspective. DEMO defines its organizational levels based on a communication paradigm. Functional levels are defined in DEMO based on the view of business processes as transactions. DEMO defines functional and constructional decompositions as techniques for dealing with complexity of the modeled system. Decomposition techniques for DEMO models can be associated with refinement.

Object-Process Methodology (OPM) [18] proposes a method for the complete integration of the systems' states and behaviors within a single graphical model. OPM defines abstracting and refining of its specifications as subtypes of the process called scaling. In OPM, there exist three types of hierarchies:

aggregation-participation, exhibition-characterization, and generalization-specialization. Transition to the next (lower) hierarchical level in OPM is a result of the refinement mode called unfolding.

Object-oriented modeling method for software called ADORA (Analysis and Description of Requirements and Architecture) is presented in [19]. Models in ADORA are composed of hierarchically structured abstract objects. The mechanism of hierarchical decomposition is applied to views [20]. ADORA defines a formal refinement calculus semantic for the structural, behavioral, and user views. Basic refinement types as well as refinement rules for each of these views are defined.

In Goal-Oriented Requirements Engineering, the KAOS method [21] bears much resemblance to our approach. KAOS formally aligns high level goals with requirements for an IT system. It is based on temporal logic and is mainly used for the requirements of real-time embedded systems.

In [22] the authors represent the business process as a trajectory in a state space. The authors attempt to describe declaratively the dynamics of a business process by defining a notion of a valid state and planning rules that make a state valid. This notation has similar roots with SEAM representation of a system as a working object having a state and an action that changes this state.

5 Conclusions

In ITIL, The service specification is described with a service level agreement (SLA). To support this agreement, the IT service providers need to specify the planned implementation – with OLAs. ITIL does not propose specific technique to verify whether an SLA is adequately supported by the UCs and the OLAs. This work addresses process definition and the relation between the process definition and the process goal (the SLA in our case).

In this paper, we have shown how a software engineering technique, i.e. refinement theory, can be applied to verify the alignment between the service specification (SLAs) and the service construction (OLAs), modeled declaratively [8]. Our declarative specifications define an IT service and its construction. We have also illustrated how Alloy can be used to verify the alignment.

Future work includes the development of a simpler notation for use in business workshops, the extension of the technique to larger scale examples, and the separation of ITIL utility (functional requirements) and ITIL warranty (non functional requirements).

References

[1] Wegmann, A.: On the Systemic Enterprise Architecture Methodology (SEAM), International Conference on Enterprise Information Systems (ICEIS), Angers, France, 2003.

[2] I. Rychkova, A. Wegmann, “Refinement propagation. Towards automated construction of visual specifications”, International Conference on Enterprise Information Systems (ICEIS) (2007)

[3] D. Jackson, “Software Abstractions: Logic, Language, and Analysis”, MIT Press. Cambridge, MA. March 2006. ISBN 0-262-10114-9.

[4] Wirth, N.: Program development by stepwise refinement. Communications of the ACM, 1971

[5] Spivey, J.M.: The Z notation: A reference manual. Prentice Hall, 1989.

[6] He, J., Hoare, C., Sanders, J.: Data refinement refined. ESOP 86 Lecture Notes in Computer Science 213 (1986) 187–196

[7] Nancy A. Lynch, Frits W. Vaandrager: Forward and Backward Simulations: I. Untimed Systems Inf. Comput. 121(2): 214-233 (1995)

[8] Rychkova I., RegevG., Wegmann A. : High-Level Design and Analysis of Business Processes. The Advantages of Declarative Specifications, The Second IEEE International Conference on Research Challenges in Information Science (RCIS), 2008.

[9] Alloy Analyzer 4.0, http://alloy.mit.edu/alloy4/

[10] Dijkstra, E. W. “Notes on structured programming”. 1971

[11] Clarke E., Emerson E.A., Sistla A.P.: Automatic verification of finite state concurrent systems using temporal logic, ACM Trans. on Programming Languages and Systems, 1986.

[12] Gordon M.J.C., Melham T.F.: Introduction to HOL: a theorem proving environment for higher order logic, 1993, Cambridge University Press New York, NY, USA.

[13] Holzmann, Gerard J.: The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley Professional, 2003

[14] Clarke E., Grumberg O., Jha S., Lu Y., Veith H.: Progress on the state explosion problem in model checking. In Informatics, 10 Years Back, 10 Years Ahead, 2001.

[15] SPASS: An Automated Theorem Prover for First-Order Logic with Equality http://spass.mpi-sb.mpg.de/

[16] Paulson L.: Isabelle: A Generic Theorem Prover, Springer, 1994.

[17] Dietz, J. L. G.: DEMO: towards a discipline of Organisation Engineering. 1999.

[18] Dori, D. Object-Process Methodology. A Holistic Systems Paradigm, Springer Verlag, 2002

[19] Glinz M., Berner S., Joos S., Ryser J., Schett N., Xia Y., "The ADORA Approach to Object-Oriented Modeling of Software", Lecture Notes in Computer Science, 2001

[20] Xia, Y., Glinz, M.: Extending a Graphic Modeling Language to Support Partial and Evolutionary Specification. 11th Asia-Pacific Software Engineering Conference (2004)

[21] Dardenne, A., van Lamsweerde A. and Fickas, S.: Goal Directed Requirements Acquisition, Science of Computer Programming, Vol. 20, No. 1-2, 1993

[22] Khomyakov M., Bider I.: “Achieving Workflow Flexibility through Taming the Chaos”. OOIS 2000