IT Service Alignment Verification of Quantitative Properties in Service Design

Gorica Tapandjieva, Alain Wegmann
École Polytechnique Fédérale de Lausanne,
CH-1015 Lausanne, Switzerland
{gorica.tapandjieva, alain.wegmann}@epfl.ch

***Abstract*—Despite many years of research, alignment of business and IT services remains a challenge. In this paper we show how to verify the quantitative properties of a service against stakeholder requirements during service design. We model the service with the Systemic Enterprise Architecture Method (SEAM). This allows us to specify the service alignment constraints with what we call a feasibility constraint. We translate the SEAM model into Scala code, where the feasibility constraint is mapped to a constraint of a Scala function. We then check the Scala function's verification condition with the Leon verification tool. An alignment is achieved if no counterexample is found. If a counterexample exists, it allows to detect which service component is at the source of the misalignment.**

Keywords*—*service design, quantitative properties, business and IT alignment, service specification, alignment verification, service modeling, service implementation, SEAM, Leon

I. Introduction

Service design is the activity in which stakeholder requirements are transformed into specifications that define a set of service design constraints [1]. One of the challenges in service design, more precisely the service specification phase, is to make sure that the service constraints are aligned among themselves and with the requirements. We see these constraints as properties of the service. To simplify this challenge, we use the Systemic Enterprise Architecture Method (SEAM) [2] to model service hierarchies with service properties and their relationships. We call these hierarchies:

- **level 1** specification of the service offering showing the relationship between a service provider and a customer;
- **level 2** specification of the service implementation showing the relationships between the components responsible for delivering the service.

This simplifies the alignment into verifying that:

1) the properties of the service offering meets the customer requirements, and
2) the relationships between the specified service implementation components satisfy the constraints set by each individual component.

We define the verification at each level as the feasibility at that level, and the conjunction of feasibilities from both levels, as the alignment of the service.

To prove the feasibility, we translate the SEAM model into Scala code [3]. More precisely, for each level we represent the relationships between the components as a Scala function. The feasibility is asserted by the verification of a constraint on that function. This code is passed to a Scala verification tool called Leon [4]. If all the functions constraints hold, we say that the translated SEAM service model is aligned. Otherwise, Leon returns a counter-example that violates a constraint, and with this counter-example we can detect which component is the cause for misalignment.

In this paper we explore the refinement and verification of the properties that must hold at any time during the execution of the service (invariant properties), whereas Rychkova [5] proposed mechanisms for alignment in SEAM based on the refinement of pre and post conditions, i.e. properties that hold only before and after a service was executed. Many of these invariant properties, such as performance characteristics (response time, throughput), capacity, power consumption, number of users, financial characteristics (budget, cost), are quantified. We do not distinguish between functional and non-functional because in SEAM there is no such classification. We propose to quantify the service properties at the beginning of the design process because the quantities have a direct impact on design choices. As a result, a limitation of our approach is the inability to consider properties that cannot be quantified.

The paper is structured as follows: in Section II we discuss the related work on approaches that validate the alignment between the service offering and the service implementation. In Section III we present the example, and then in Section IV we describe the way to model, define and formalize quantitative properties in SEAM. In Section V we map the SEAM graphical model to Leon code and show the output of the verification, and in Section VI and VII we present the future work and conclusions.

II. Related Work

Our approach verifies the alignment between stakeholders expectations and the service implementation, so we therefore see our work related to requirements tracing, alignment, and verification of specifications (constraints) in models that describe the service architecture in a company. Another related work is the verification of behavioral business and IT alignment with SEAM. We therefore split the related work in:

A. Requirements traceability and verification

Plataniotis et al. [6] present the EA Anamnesis metamodel that enables traceability of requirements and design decisions. The traceability gives the capability to check the alignment between the architectural decisions taken and the requirements.

The metamodel presented includes a problem and a solution space, both of them bridged with the Functional and Non-functional sub-classes. Unlike our approach, authors make the classification of functional and non-functional requirements, and their scope does not include verification of requirements.

An ArchiMate extension is described in white paper from The Open Group [7]. This extension deals with requirements management in the context of enterprise architecture. The white paper describes a combination of existing requirements engineering techniques within the TOGAF ADM phases. It emphasizes the importance of the alignment and traceability between requirements and the architectural elements of a solution because they shape the overall architectural design, but it does not provide an example of requirements alignment verification.

Ramesh and Jarke [8] present an analysis of requirements traceability tools and establish reference models for traceability, a low-end and a high-end model. The first one targets users that care about a concrete relationship between a requirement and a system component, whereas the second one targets users that care about the full life-cycle traceability of a requirement. We can relate the low-end traceability model to the SEAM hierarchy level showing the relationships between components responsible for delivering the service, and the high-end model to both SEAM hierarchies.

Hallerstede et al. [9] describe an approach, based on the reference model for requirements and specifications' by Gunter et al.[10] for requirements modeling and validation for a target system description complemented by "traces". The system description is defined as a description of the system and the environment it interacts with. The model for tracing requirements is based on WRSPM. Authors also include formal specifications based on Event-B, that incorporates formal and informal reasoning. The target system description incorporates the conclusions about the achieving the requirements and the correctness of the specification. If we consider that the system offers a service, we can see the environment of the system as the first level of our SEAM service hierarchy, and the system itself as the second level.

In [11], Almeida et al. provide a methodological framework for requirements tracing in model-driven development process. The model-driven design process can have different levels of abstraction, so authors simplify the problem of tracing with introducing notion of conformance between models of different levels of abstraction. In our approach we see this conformance as alignment between the models.

In this paper, we provide another way to "trace" requirements between levels. Requirements are sometimes considered as part from the business domain, and specification as part from the IT domain. Since our method shows both, we use the term properties to denote the requirements and the specifications.

B. Modeling and verifying specifications

UMLtoCSP [12] is a fully automated tool used for formal verification of UML/OCL class-diagram models. The tool first translates the model into a Constraint Satisfaction Problem (CSP), and then tries to find a solution where all constraints are satisfied. In our method we map the constraints to a constraints of a Scala function, and then for the verification we use Leon, that generates a verification condition for that function. The difference in the verification approach is that UMLtoCSP tries to at least one solution where all constraints are satisfied, whereas Leon, the verification tool we use, tries to prove the verification condition, and if this condition is true, the model is correct for all constraints.

In [13], requirements verification is done with XSLT over requirements managed in an XML based RE tool. The tool and the verification proposed are lightweight, but the approach is not described on a visual model example. Our SEAM visual models are saved in an XML format, so [13] gives us a new perspective for doing alignment verification directly on the model, without using additional tools.

C. Alignment with the Systemic Enterprise Architecture Method

Wegmann et al. use SEAM to present a behavioral business and IT alignment between the different views of a same system [14]. Rychkova et al. use the same method to align the applications to be developed with the business requirements [15]. The SEAM modeling technique is also presented in [5], where Rychkova et al. verify the alignment using the Alloy tool [16] in terms of properties that describe a Service Level Agreement (SLA) and Operational Level Agreements (OLA). Our approach is mostly inspired by the work of Rychkova, and can be seen as complementary to it [17].

The novelties presented here come from the evolution of the SEAM method. The verification in [17] is done with behavior properties conditions of the hierarchical systems, split in preconditions and postconditions, whereas in this paper we extend the properties scope to quantitative properties, which hold at any time, i.e. invariant properties. We additionally use Scala, a recently developed programming language, and we use the Leon verification tool.

III. Example: EPFL Storage Service for Students

A. Description of EPFL Storage Service for Students

The example we use to illustrate our method is based on a concrete project done at the Ecole Polytechnique Fédérale de Lausanne (EPFL). The goal of the project is to replace the technical solution for an existing central **storage service** provided by the *EPFL storage organization*.
The actors involved in the storage service are:

- *EPFL students* – customers of the storage service. They need a way to store their documents while studying at EPFL, so EPFL wants to give them an alternative service to Dropbox or Google Drive.
- *Storage service steering committee* – a governance body that makes decisions concerning the storage service. The people in this committee set the annual global **budget** for the operation of the central storage service. For simplicity, we don't consider the budget needed for the development of the service. This committee also knows the strategic directions of EPFL, the current **number of students** and the expected growth of this number.

- *EPFL IT infrastructure department* – an EPFL department that offers all IT services necessary for the everyday work at EPFL, including the storage service. The steering committee allocates the budget to this department for the operation of the storage infrastructure. This budget constrains the storage *capacity* provided by this department.
- The *EPFL help desk* – the single point of contact for all IT services provided at EPFL including the storage service. The help desk has its own running budget for the storage service assistance set by the steering committee.

B. Quantitative Properties of EPFL Storage Service for Students

We take into account the following properties: number of students, storage capacity and amount of budget. The values and ranges given for these properties serve only as illustration:

- An EPFL *student* expects to get and use between 25 and 40 GB of storage provided by the EPFL storage organization.
- The *storage project steering committee* considers allocating between 150'000 and 200'000 Swiss Francs, for the central storage service. The committee also specifies that this storage service has to be available for 9500 to 10500 students.
- The *EPFL IT infrastructure department* should provide storage infrastructure with capacity between 400TB and 500TB, for a total cost of ownership between 120'000 and 150'000 Swiss Francs per year (approximately 300 Swiss Francs per TB per year).
- The *EPFL help desk* needs up to 10'000 Swiss Francs per year for the operation of their help and assistance service for storage.

The storage service has the following constraints:

1) The total capacity provided by the IT infrastructure department, is evenly distributed to the number of students.
2) The storage capacity per student has to be greater than what each student requirements.
3) The total budget allocated to the IT infrastructure department and the help desk can not exceed the budget decided by the steering committee.

There exists a counter-example for these chosen values and constraints for the presented service. This is not obvious from the textual representation of the properties. In the following sections we show how to model, formalize and automatically verify these properties.

IV. Modeling Quantitative Properties in SEAM

With SEAM we conceptualize an organization as a **hierarchy of systems**[1] that provide services (from business down to IT, also known as organizational level hierarchy).

[1]To clarify, we use **system** to refer to an observed entity: an organization, an employee, an IT system, or an application [18].

In this hierarchy, we conceptualize a system as a whole, denoted as [w] or as a composite, denoted as [c]. In a system as a whole, we ignore the system's components and represent the service provided. In a system as a composite, the components and their relationships are visible, together with the process that combines the services provided by each component. The process represents a service implemented by that system. Systems that represent a business entity (company, organization, department, etc.) are modeled with block arrows, whereas people are modeled with a rectangle and a stickman (see Fig. 1). Concerning links, the whole and composite view of a

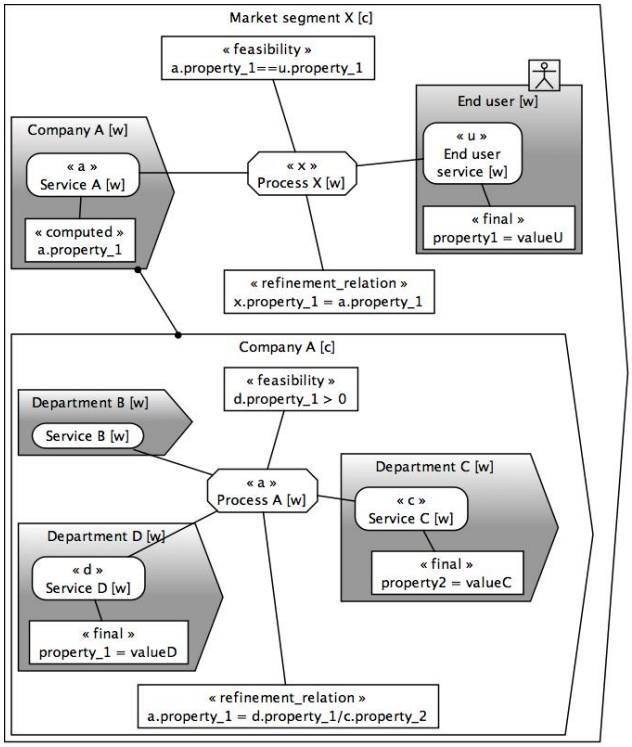

Fig. 1. A SEAM example of a *Market segment X[c]* system modeled as a composite, including *Company A[w]* and an *End user[w]* systems as a whole. *Company A[w]* is furthermore refined to *Company A[c]*, a system as a composite, composed of three systems as a whole: *Department B[w]*, *Department C[w]* and *Department D[w]*, and *Process A[w]*. *Service A[w]* and *End user service[w]* are services needed to perform *Process X[w]*, which combines them. The same holds for *Service B, C and D[w]* connected to *Process A[w]*, where *Process A[w]* implements *Service A[w]*.

system are connected with a refinement (decomposition) link. For example, the link between *Company A[w]* and *Company A[c]* in Fig. 1. We use a plain link for connecting services to a process. This link means that the process combines (uses) that service. For example, in Fig. 1, the links between *Process A[w]* and *Service B, C and D[w]* are of this kind. Plain link is also used for connecting properties to a service and a process.

SEAM behavior properties specifications and the verification of their alignment within different hierarchical levels is introduced in [17]. Here we extend the SEAM notation with quantitative properties and the $\ll$*feasibility*$\gg$ boolean expression property. We say that the specification of the quantitative properties in the model is correct if and only if the $\ll$*feasibility*$\gg$ expression is true.

In SEAM, services and processes have properties, which

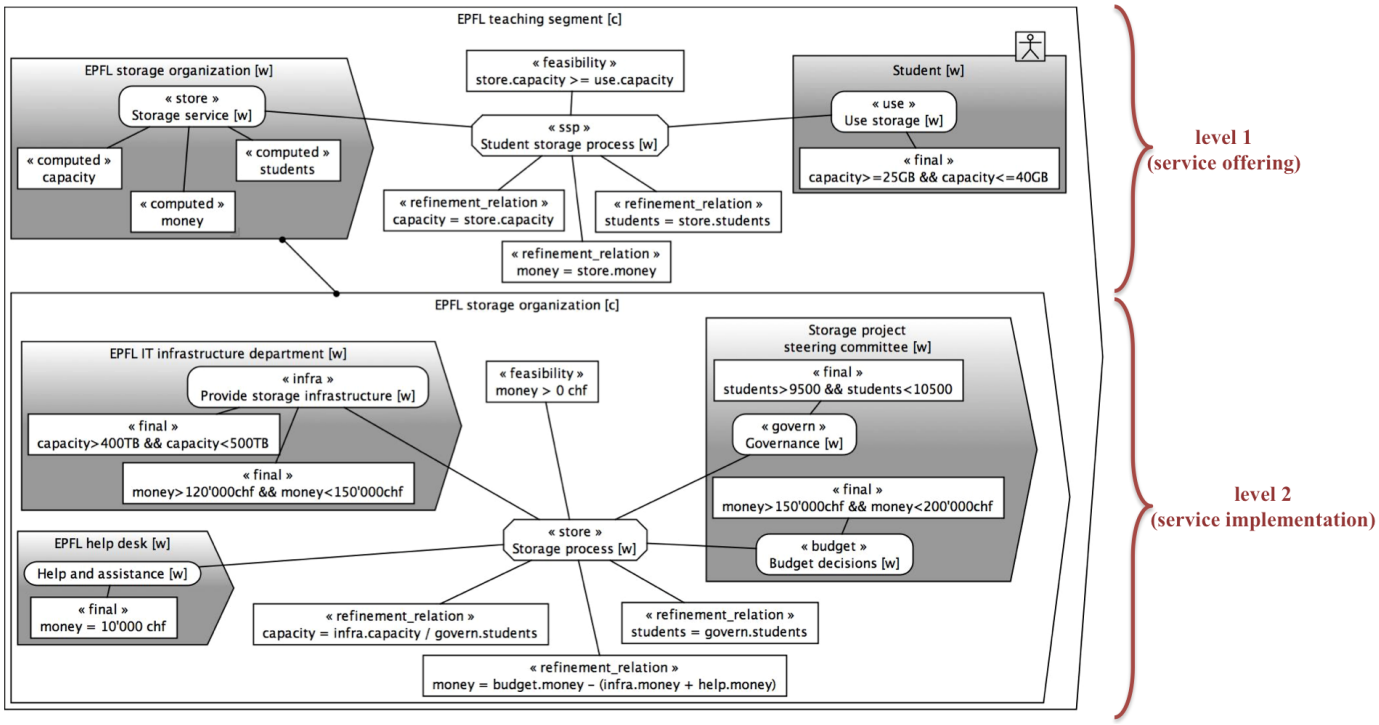

Fig. 2. SEAM model for the EPFL storage service example, showing the storage service offering in the first level, and the storage service implementation in the second. The components in each level have properties that describe the constraints of the level.

are modeled with rectangles linked to the service or a process. The quantitative properties we introduce have a stereotype, written on top of the rectangle between ≪...≫, to denote the type of property. There are four types of properties, two types for a service, and two types for a process.

Service property types:

- ≪*final*≫ - A property that gives a set of values independent of other properties in the model. It is usually a range, or even one value.
- ≪*computed*≫ - A property whose values are computed by the service implementation process ≪*refinement_relation*≫ properties, and then transferred to the service level.

Process property types: The process is connected to several services, so the logic of the model execution is in the following two types of properties:

- ≪*refinement_relation*≫ - A property that computes the quantity (value) by using the properties values of services connected with the process. The computed value is then transferred to the ≪*computed*≫ property of the service being implemented.
- ≪*feasibility*≫ - A property, containing a boolean expression, which by definition is present in a composite system, in the level where a service is implemented. It defines the verification of that level, which is usually a comparison constraint. We define the alignment of a service as conjunction of two feasibilities: (1) from the level where the service is used, and (2) from the level where the service is implemented.

Following, we describe the SEAM model for the example in Section III and we go in detail about SEAM properties computations and their relationships.

A. Service Implementation Level

In Fig. 2 *level 2* we model the *EPFL storage organization[c]* as a composite system. This system contains the *Storage process[w]* that implements the *Storage service[w]* offering from *level 1*. The *EPFL storage organization[c]* contains the following systems as a whole:

- *EPFL IT infrastructure department[w]* with the *Provide storage infrastructure[w]* service,
- *EPFL help desk[w]* with the *Help and assistance[w]* service, and
- *Storage project steering committee[w]* with the *Governance[w]* and *Budget decisions[w]* services.

These services are needed to perform the *Storage process[w]*, so we model a link between them and the *Storage process[w]*. All ≪*final*≫ and ≪*computed*≫ properties in Fig. 2 refer to **capacity, money** and number of **students**, and there is one ≪*feasibility*≫ and three ≪*refinement_relation*≫ properties connected to the *Storage process[w]*.

The ≪*final*≫ properties of the services in *EPFL storage organization[c]* are specified with a range or a value (e.g. the *capacity* property within ≪*infra*≫*Provide storage infrastructure[w]* is set to be between 400TB and 500TB).

The ≪*refinement_relation*≫ properties of a process depend on the service properties of services connected with the process (e.g. the value of the *capacity* property within

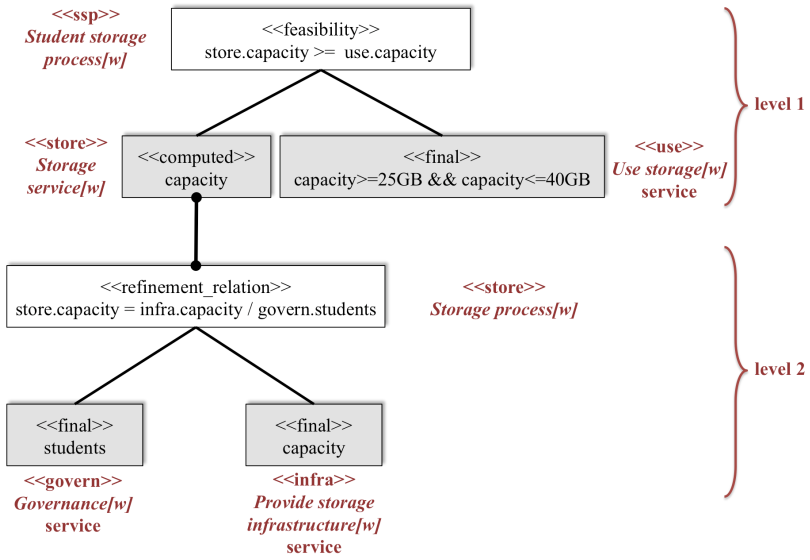

Fig. 3. Compute the ≪*refinement_relation*≫ *capacity* property in the ≪*store*≫*Storage process[w]* in *level 2* and passing this value to the ≪*store*≫*Storage service[w]* ≪*computed*≫ *capacity* property in *level 1*, which is then used to verify the *Student storage process[w]* ≪*feasibility*≫.

≪*store*≫*Storage process[w]* is computed by dividing the capacity provided by *Provide storage infrastructure[w]* with the number of students set in the *Governance[w]* service, *storage_infra.capacity/govern.students*).

The ≪*feasibility*≫ property is a boolean expression that depends on the service properties of services connected with the process (e.g. the *Storage process[w]* ≪*feasibility*≫ expression is *money>0 chf*, where $store.money = budget.money - (storage\_infra.money + help.money)$, as defined in a ≪*refinement_relation*≫).

The properties of a process compute values that:

1) together with the properties of the services are verified with the boolean expression in ≪*feasibility*≫,
2) are transferred to the *Storage service[w]* ≪*computed*≫ properties, in *level 1*, implemented by the *Storage process[w]*, in *level 2* of Fig. 2, only if ≪*feasibility*≫ holds.

We say that the model system as a composite is correct and passes the computed property values to the *EPFL storage organization[w]* seen as a whole if and only if the ≪*feasibility*≫ expression is true.

B. Service Offering Level

In Fig. 2, the first level of the organizational hierarchy is the *EPFL teaching segment [c]*, seen as a composite. It is composed of the *EPFL storage organization [w]*, seen as a whole, and the customer - *Student[w]*, also seen as a whole. The *EPFL storage organization [w]* has the *Storage service[w]* and the *Student[w]* has the *Use storage[w]* service.

The *Storage service[w]* ≪*computed*≫ properties values are transferred from the *Storage process[w]* ≪*refinement_relation*≫ from the service implementation level explained in the previous subsection.

Similar to the *Storage process[w]*, the *Student storage process[w]* has an ≪*feasibility*≫ property boolean expression and ≪*refinement_relation*≫ properties that compute values.

The *Use storage[w]* service present in *Student[w]* is specified with $capacity >= 25GB \;\&\&\; capacity <= 40GB$ for the ≪*final*≫ property.

C. Computing and Transferring SEAM Quantitative Properties

As already seen, ≪*refinement_relation*≫ properties of a process compute the quantity (value) by using the properties values of services connected with the process (the computed values are then transferred to the upper level, to the service being implemented by the process, as shown in Fig. 3). The ≪*feasibility*≫ property boolean expression checks the correctness of the final or computed values connected to the process within one level (if the evaluation of the ≪*feasibility*≫ is false, then we say that the model is wrong).

The definition for the quantitative properties is inspired by [17].

D. Extraction of Properties From SEAM Models

In order to introduce formal semantics for SEAM properties, we follow these two steps:

1) Identify all quantitative properties by looking at services and processes in each system of the model. This gives the tuple $P = (p_1, p_2, \ldots, p_n)$. The tuple containing all the properties for our example in Fig. 2 is $P = (capacity, money, students)$.
2) For each service:
   - For a ≪*final*≫ property, find the specification formula F_S describing the service. The formula can be a constant value, a range, or a set (e.g. $F_{use} = capacity \geq 25GB \;\&\&\; capacity \leq 40GB$, seen in Fig. 2 *level 1* and Fig. 4).

- For a ≪*computed*≫ property, the process of the lower level is the one that computes and transfers the values to the properties of the service (e.g. the *Storage service[w]* with the ≪*computed*≫ *capacity* property depicted in Fig. 2 *level 1*, is implemented in *level 2* with the *Storage process[w]*, where one ≪*refinement_relation*≫ computes the *capacity* value).

3) For a process:
   - Find all services connected to the process (e.g. in *level 1* of Fig. 2, *Student storage process[w]* is connected to *Storage service[w]* and *Use storage[w]* service).
   - Find the ≪*feasibility*≫ boolean expression F_f (e.g. in *level 1* of Fig. 2, ≪*feasibility*≫ is $store.capacity >= use.capacity$).
   - Find the ≪*refinement_relation*≫ properties for the process F_{PS}. This property is a formula that computes process properties values and it depends on the properties of the services connected to it. If F_f evaluates to true, the computed values are transferred to the upper level as properties of the service being implemented (e.g. this formula transfers values from *Storage process[w]* in *level 2* to *Storage service[w]* in *level 1* of Fig. 2).

E. Formal Semantics for SEAM Quantitative Properties

We define the formal semantic for a service and a process in relation to the quantitative properties:

- We define a **service** S, with $S = (F_S, P_S) = F_S(P_S)$, where:
  - F_S is a specification formula describing the constraints of the service.
  - P_S is a properties tuple that satisfies the F_S given as an output of the service.

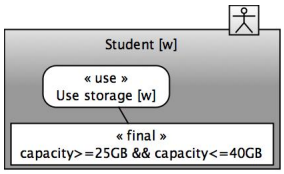

Fig. 4. *Use storage[w]* localized action, as depicted in Fig. 2 *level 1*

Our SEAM model example in Fig. 2 has three properties: capacity, money and students, so we define the tuple for properties as: $P_S = (capacity, money, students)$. The tuple for the *Use storage[w]* service in *level 1* is $P_{use} = (capacity, 0, 0)$[2], and

$$S_{use} = F_{use}(P_{use})$$

where $F_{use}(P_{use})$ is the following expression:

$$P_{use}.capacity \geq 25GB \wedge P_{use}.capacity \leq 40GB$$

The resulting value of the ≪*final*≫ *capacity* property for the student is any value between 25 and 40 GB.

- We define a **process** PS with:
  - $P_{S_{set}} = \{P_{S_1}, P_{S_2}, \ldots, P_{S_n}\}$ - a set of properties tuples from the services it uses.
  - P_{PS} - a properties tuple given as an output of the process, transferred to the corresponding service in the upper level with the ≪*computed*≫ stereotype.
  - F_{rr} - a formula, from all the ≪*refinement_relation*≫ process properties, dependent on $P_{S_{set}}$, that returns the output tuple P_{PS}. It expresses the specification and the logic of execution of all services in connected to the process.
  - F_f - a boolean expression depending on $P_{S_{set}}$ and P_{PS} corresponding to the process ≪*feasibility*≫ property.

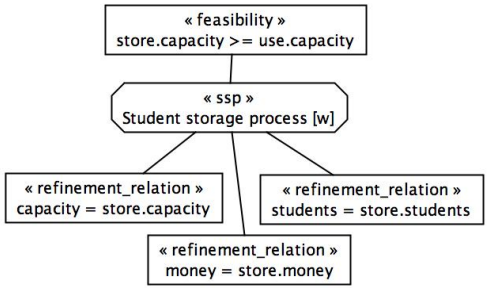

Fig. 5. *Student storage process[w]*, as depicted in Fig. 2 *level 1*

The process from Fig. 5 is formalized with PS_{ssp}, and we extract[3]:
- Properties used by the process: $P_{S_{ssp}} = \{P_{store}, P_{use}\}$
- Output properties tuple: $P_{ssp} = (capacity, money, students)$
- Refinement relationship formula with the computation of the output properties: $F_{rr_{ssp}} = ((P_{ssp}.capacity = P_{store}.capacity) \wedge (P_{ssp}.money = P_{store}.money) \wedge (P_{ssp}.students = P_{store}.students))$. This formula uses the values of ≪*computed*≫ properties only from the *Storage service[w]*.
- The ≪*feasibility*≫ constraint: $F_{f_{ssp}} = (P_{store}.p_{capacity} \geq P_{use}.capacity)$.

V. Automated Verification of Quantitative Properties in SEAM

Our goal is to verify the specified service and process properties specification formulas against the boolean expression in the ≪*feasibility*≫ property that describes the alignment. As previously mentioned, the verification tool we use is Leon [4], [19].

A. Leon

Leon is a verification system for a purely functional subset of the Scala [3] programming language [4], [19], [20], called

[2]Certain services are independent of some properties in the model, so any value can be used. In our example, we assign 0 to such properties.

[3]For readability, we use ssp to denote the *Student storage process[w]*.

Pure Scala. For each function written in Pure Scala, Leon generates a verification condition and tries to prove it [19]. The verification condition states that when the given precondition holds, after executing the function, the postcondition will hold as well. In Scala, the pre and postconditions are written with the *require* and *ensuring* methods respectively. Leon uses external automated theorem proving tools (SMT solvers: Z3 [21] and CVC4 [22]), and combines them with an internal algorithm to prove the generated verification condition. After running Leon, for each function the output can be:

- *valid* - Leon has proved that for any input for which the precondition is true, the postcondition will always hold.
- *invalid* - there is at least one counter-example, and Leon returns one, that violates the postcondition, but satisfies the precondition.
- *unknown* - usually returned after a timeout or an internal error.

For constraint solving, Leon implements the *epsilon operator* - ϵ, with the *choose* function. Choose is mainly used to find a contradiction in an interpretation of a formula [23], and we use it to specify constraints. For a given formula F and a variable x, then $\epsilon x.F$ returns a value that can be assigned to x such that F becomes true. If F is not realizable, then $\epsilon x.F$ can return any value [20]. With using choose in a function, Leon finds a counter-example that can be taken by epsilon to make the function's verification condition false. This helps in tracking the specification that gives a counter-example, and improve it.

B. Transformation of SEAM Models to Leon (Scala) Code

We first define the tuple for our example properties $P = (capacity, money, students)$ with a case class [3]:

```
case class P(capacity:Int, money:Int, students:Int)
```

We use this class to store, compute, compare and pass values for the properties in the model. Services and processes are treated differently:

- Every SEAM service that contains a property is mapped to a Scala val variable[4].
  For a service with a ≪*final*≫ property, the val calls the Leon *choose* function[5] with the F_S service specification formula, and returns the P_S service output properties tuple.
  This listing shows how the *Use storage[w]* service with a ≪*final*≫ property from Fig. 4 and from *level 1* in Fig. 2 is translated to Leon code:

  ```
  val s_use: P = P(choose((i: Int)
    => i >= 25 && i <= 40), 0, 0)
  ```

  For a service with a ≪*computed*≫ property, this val is set to the output of the Scala function with which we define the process. This function is described in the next bullet point, and the following listing shows the we assign a value to the ≪*store*≫ *Storage service[w]*, where this value is computed by the ≪*store*≫ *Storage process[w]*:

  ```
  val s_store: P = ps_store(s_budget, s_govern,
       s_help, s_infra)
  ```

- Each process is mapped to a Scala function. The properties from all services connected to the process in the $P_{S_{set}}$ are input parameters for this function. Since all the values of these $P_{S_{set}}$ properties come from other already specified services written as a Scala val, we need to write a precondition in the Scala *require* statement to match the values of the input $P_{S_{set}}$ with the values specified in the val variable for the corresponding services. Then, the output is computed based on the formulas from ≪*refinement_relation*≫ properties, as defined in the F_{rr} and are used in the Scala function's body for the computation of the output P_{PS}.
  Every property in the model is an invariant property, and every Scala function can be have both a precondition (with the Scala *require* statement), or a postcondition (with the Scala *ensuring* statement). Our choice is to use the *ensuring* statement for checking the F_f constraint written in the ≪*feasibility*≫ property.
  The following listing shows how the *Student storage process[w]* from Fig. 5 and *level 2* in Fig. 2 is translated to Leon code:

  ```
  def ps_ssp(use: P, store: P):P = {
   require(use == s_use && store == s_store)
   P(store.capacity, store.money, store.
       students)
  } ensuring(store.capacity >= use.capacity)
  ```

C. Leon Output

We develop an algorithm that transforms SEAM model's actions to Scala code according the mapping rules described here. This algorithm is not yet implemented, so the transition from the SEAM model to the Scala code is not automatic. In Fig. 6, the output of running the code for our example is shown, where Leon finds a counter-example.

Leon finds that *ps_ssp* and *ps_sp* processes are invalid, giving counter examples with the values for the properties of the services involved.

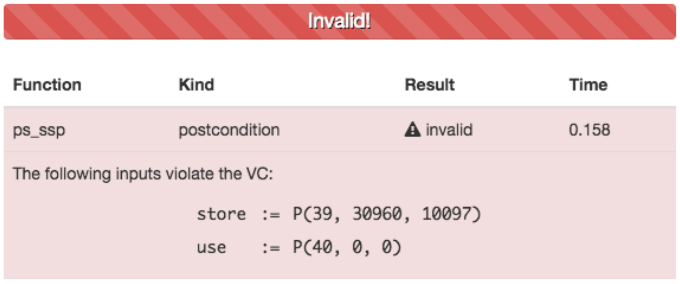

Fig. 6. Leon output giving a counter-example for not having enough storage capacity in the *Student storage process[w]* - *ps_ssp* from the service offering (level 1) in Fig. 2, when the 39 GB capacity offered by the ≪*store*≫Storage service[w] is smaller than the 40 GB capacity required by a student.

[4]"Scala has two kinds of variables, vals and vars. A val is similar to a final variable in Java. Once initialized, a val can never be reassigned."[3]

[5]As mentioned, *choose* is the Leon implementation of the ϵ operator.

ps_ssp – The specification is invalid (see Fig. 6) when the EPFL storage organization serves 10'097 students, with 39GB capacity per student ($store := P(39, 30960, 10097)$), when the student requires 40GB ($use := P(40, 0, 0)$).

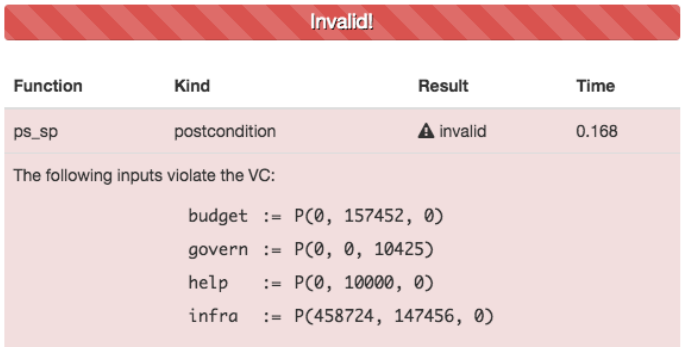

Fig. 7. Leon output giving a counter-example for not having enough money in the *Storage process[w]* - ps_sp from the service implementation (level 2) in Fig. 2, when the money allocated for the *Provide storage infrastructure[w]* service - $infra$ is 147'456 Swiss Francs, for the *Help and assistance[w]* service - $help$ is 10'000 Swiss Francs, and the global budget that should cover these two services from the *Budget decisions[w]* service - $budget$ is 157'452 Swiss Francs.

ps_sp – The specification is invalid (see Fig. 7) when the steering committee sets the annual budget to 157'452 Swiss Francs ($budget := P(0, 157452, 0)$), but assigns 147'456 Swiss Francs to EPFL IT infrastructure department for the operation of 458'724MB ($infra := P(458724, 147456, 0)$), and 10'000 Swiss Francs to EPFL help desk for the storage service support throughout an year ($help := P(0, 10000, 0)$).

The properties values for these two invalid functions are unrelated. In Fig. 7, the ps_sp postcondition does not hold for the properties values listed, they are a counter-example, so ps_sp can not be executed. The values in Fig. 6 for $storage$ depend on a valid ps_sp execution, so they are unrelated with the values from Fig. 7.

VI. Future Work

One part of the future work is the formalization and translation of functional properties and their behavior to Leon code. We also wish to automatically verify SEAM models built with the SeamCAD tool [24]. This tool saves SEAM models in XML format. The automatic generation of Leon code from XML (model to text) is another part of our future work.

The presented approach is applicable to qualitative properties by enumerating or refining them to a set of quantitative properties. For example, we can consider reliability as a quantitative property when we see a constraint that a system cannot be down for more than 15 hours per year. This part of the research is at a very early stage, and remains in future work.

The most important part of the future work is validating the application of our approach in practice.

Following, we propose improvements for modeling with SEAM and the Leon verification tool:

A. Modeling Improvements

SEAM uses the same link notation for connecting a service to a process, and a property to a service or a property to a process. Having different lines would reflect the semantics of the properties presented in this paper. The different notation would make the future automatic translation of SEAM model to Scala code easier.

In this paper we also use Scala syntax for the graphical notation for a SEAM property value computation and properties comparison. This reduces the effort to translate the model to Leon code, but requires basic knowledge of Scala programming while building the SEAM model.

B. Leon Improvements

Leon's support for only Pure Scala programs limits the expressiveness of service specifications. Pure Scala supports arithmetic operations over integer numbers only. The division operator $/$ and the modulo operator $\%$ should only be invoked with positive arguments, and they return integer values.

In our example we used integer division, which might lead to a confusion in the interpretation of the counter-example. In Fig. 6, the counter-example for the ps_ssp is $store := P(39, 30960, 10097)$, where 10'097 students with 39GB capacity per student make a total of 393'783 GB provided by the IT infrastructure department. This value does not satisfy the defined range in the $\ll$*final*$\gg$ capacity property (see Fig. 2), but any value above $39.616GB$ for the *Storage service[w]* would fall in the allowed range.

The integer division can be improved by defining a new division operator that uses $/$ and $\%$.

VII. Conclusions

In this paper we presented an approach to verify the alignment of quantitative invariant service specification properties in SEAM models. This alignment verification is simplified by modeling two levels of the service specification: a service offering and a service implementation, and checking the correctness of each level's feasibility boolean expression property. The SEAM model is then translated into Scala code and verified with a tool called Leon.

One limitation of our approach is the possible timeout before Leon finds a counter-example. Also, in some functions with a valid verification condition, Leon is not able to prove the correctness [19].

In [5] the authors verify the alignment based on formal refinement of pre and post conditions of a service. In this paper we focus on the alignment of the invariant properties of a service specification that at the same time can be quantified. These quantitative service properties influence the service design decisions, so the outcome of their verification helps the service designers.

References

[1] D. Cannon *et al.*, "ITIL Service Strategy 2011 Edition," *TSO The Stationery Office, London*, 2011.

[2] A. Wegmann, "On the Systemic Enterprise Architecture Methodology (SEAM)," in *ICEIS*, 2003.

[3] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Inc, 2008.

[4] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter, "An overview of the leon verification system: Verification by translation to recursive functions," in *Proceedings of the 4th Workshop on Scala*. ACM, 2013, p. 1.

[5] I. Rychkova, G. Regev, and A. Wegmann, "Declarative Specification and Alignment Verification of Services in ITIL," in *Enterprise Distributed Object Computing Conference Workshops, 2008 12th*. IEEE, 2008, pp. 127–134.

[6] Plataniotis, Georgios and De Kinderen, Sybren and Ma, Qin and Proper, Henderik, "Traceability and Modeling of Requirements in Enterprise Architecture from a Design Rationale Perspective," in *Ninth IEEE conference on Research Challenges in Information Systems (RCIS 2015)*, 2015.

[7] W. Engelsman, H. Jonkers, and D. Quartel, "Archimate® extension for modeling and managing motivation, principles, and requirements in togaf®," *White paper, The Open Group*, 2011.

[8] B. Ramesh and M. Jarke, "Towards Reference Models for Requirements Traceability," *Software Engineering, IEEE Transactions on*, vol. 27, no. 1, pp. 58–93, 2001.

[9] S. Hallerstede, M. Jastram, and L. Ladenberger, "A Method and Tool for Tracing Requirements into Specifications," *Science of Computer Programming*, vol. 82, pp. 2–21, 2014.

[10] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A Reference Model for Requirements and Specifications," in *Requirements engineering, 2000. Proceedings. 4th International Conference on*. IEEE, 2000, p. 189.

[11] J. P. A. Almeida, M.-E. Iacob, and P. Van Eck, "Requirements Traceability in Model-Driven Development: Applying Model and Transformation Conformance," *Information Systems Frontiers*, vol. 9, no. 4, pp. 327–342, 2007.

[12] J. Cabot, R. Clarisó, and D. Riera, "UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 547–548.

[13] A. Durán, A. Ruiz-Cortés, R. Corchuelo, and M. Toro, "Supporting Requirements Verification Using XSLT," in *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*. IEEE, 2002, pp. 165–172.

[14] A. Wegmann, G. Regev, I. Rychkova, L.-S. Lê, J. D. De La Cruz, and P. Julia, "Business and IT Alignment with SEAM for Enterprise Architecture," in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, 2007.

[15] I. Rychkova, G. Regev, L. Le, and A. Wegmann, "From Business To IT with SEAM: the J2EE Pet Store Example," in *Enterprise Distributed Object Computing Conference, 2007. EDOC 2007. 11th IEEE International*. IEEE, 2007, pp. 495–495.

[16] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," 2001.

[17] I. Rychkova, "Formal semantics for refinement verification of entreprise models," *Doctoral dissertation*, 2008.

[18] A. Wegmann, A. Kotsalainen, L. Matthey, G. Regev, and A. Giannattasio, "Augmenting the Zachman Enterprise Architecture Framework With a Systemic Conceptualization," in *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*. IEEE, 2008, pp. 3–13.

[19] Philippe Suter, Etienne Kneuss, Régis Blanc, Manos Koukoutos, Viktor Kuncak, "Leon documentation," http://leon.epfl.ch/doc/index.html.

[20] R. W. Blanc, "Verification of Imperative Programs in Scala," 2012.

[21] Microsoft Research, MIT licence, "Z3prover," https://github.com/Z3Prover/z3.

[22] Clark Barrett, Cesare Tinelli, "About CVC4," http://cvc4.cs.nyu.edu/web/.

[23] J. Avigad and R. Zach, "The epsilon calculus," 2002.

[24] Laboratory for Systemic modeling (LAMS), "SeamCAD," http://lams.epfl.ch/seamcad/.