Alignment Verification in the Early Stage of Service Design

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

Abstract—Verification is a costly task, sometimes burdensome and tedious, requiring strong formal background. To reduce the effort and cost invested in verification, we developed a model-driven approach for automatic verification of service properties, done in the early service design phase. Our approach is based on SEAM, a service modeling method, and it incorporates a verification system called Leon. With our approach service designers do not need substantial understanding of specific formal and verification languages, since the SEAM visual service model is both the input for and output of the alignment verification.

Index Terms—service modeling, service design, alignment verification, service design specification, SEAM, Scala, Leon

I. INTRODUCTION

The authors in [1] define service systems as “a value coproduction configuration of people, technology, other internal and external service systems, and shared information”. The Information Technology Infrastructure Library (ITIL) [2] has a stage that focuses on specifying the service design. In this stage, service designers identify and document requirements [3] while taking into account the constraints of all components involved in the value co-production. Then, they add the identified requirements and constraints in a service model that is seen as the service design specification. As indicated by [4], an early verification of the service design specification is needed to avoid costly consequences in the service lifecycle.

One aim in the service design stage is to check whether the models are under-specified and allow for scenarios that violate the specification. In the existing literature, such check is done by (1) transforming the modeled specification into a formal language [5, pp. 80], and then (2) using a model checker or an automated theorem prover to check whether the formal representation satisfies specified properties [6].

In industry modeling projects, modelers find verification a difficult task, or even a task that checks the modelers’ job performance [7]. The application of formal analysis brings great benefit in eliminating errors at the early stage of development [8], but not many people have strong formal background, or want to invest much time in verification.

To reduce the effort for formal analysis of service design specifications, we present an automatic verification of requirements and constraints specified in the SEAM modeling language [9] developed in our research laboratory. Our approach has the advantage of being independent of formal specification languages and is useful for service designers who do not have substantial knowledge of verification methods. Knowing only the SEAM modeling language and basic Scala programming expressions is sufficient.

SEAM started as an application of the systemic paradigm in the field of enterprise architecture [9], with the goal to seamlessly integrate the ‘business’ with IT. Since its inception, SEAM has expanded to incorporate tools and methods for service modeling, strategic thinking, business-IT alignment, and requirements engineering [10]. Our laboratory and our industrial partners mainly use SEAM in teaching and consulting.

Our model-driven approach has resulted from searching ways to explain alignment to students. We start by specifying the service design requirements and constraints, which are conceptualized as properties. In SEAM, a service is modeled at two different levels of abstraction: service offering and service implementation. The service offering level describes the stakeholders’ relationship with the service provider. The service implementation level shows the relationships among actors, components and resources used to deliver the service. In the first level, we specify the properties that describe the requirements. In the second level, we specify the refinement of the first level properties as a combination of the properties (constraints or requirements) set by the actors, components and resources involved in the service delivery. Consequently, we define the verification of the service design specification model as the alignment verification of the model’s properties within one level and among levels.

We have developed a tool for automatic verification of these service design specification models. The tool first translates the properties of the SEAM model into functional Scala code. The resulting Scala code is then passed to and checked with a verification system called Leon. Leon either:

• confirms that the specification is correct with respect to the specified properties, or
• provides a counterexample with an erroneous property value.

This automatic translation relies on globally unique identifiers that are used to preserve the structure of the model. If a counterexample is found, the tool uses these identifiers to annotate the model and mark the faulty property specification.

Our approach focuses on properties that can be quantified, such as performance, latency, storage size, budget and maintenance time. This work extends the refinement and verification
of behavior properties, expressed in terms of pre- and post-conditions on actions [11]. A description of this extension is in [12] and in this paper we present advancements made in:

- automating the translation of the SEAM service design models to Scala verifiable code, and
- showing the result back in the SEAM model.

The remainder of this paper is organized as follows. We first outline the related work. Then we describe the alignment verification on an example and we give details of the implementation of our automatic alignment verification with SEAM, Scala and Leon. This is followed by the limitations we face. Before concluding, we present our envisioned future work.

II. RELATED WORK

To the best of our knowledge, our approach differs from the existing literature in verification either in the target domain (service design in our case) or in the support for visually displaying the verification result.

In the context of software engineering, the adoption of Model-Driven Development (MDD) includes the use of tools to check the model correctness. There exist tools for the analysis of Unified Modeling Language (UML) diagrams annotated with Object Constraint Language (OCL) constraints [13], [14]. These approaches do not provide visual analysis result.

In the context of business process modeling (BPM), in [6] the author presents a survey of different verification approaches. A tool for automatic verification of BPMN choreographies is presented in [15]. Authors do not show the BPMN diagram in the tool, and unlike our approach, the verification output is displayed in the console. In addition, we find the design and the verification of BPMN choreographies come late in or after the service design stage.

In the context of early requirements, in [16] the authors present a framework and a tool for formal verification of early requirements specifications. Their framework combines early requirements engineering (i* models) with formal methods (model checking). As i* is static, the user should write formal specifications in the Formal Tropos language to describe the temporal dynamics of the model. Besides learning a formal language, the user does not get a visual feedback from the verification.

There exist many modeling approaches, such as the ones presented in [17], [18] and [19], that emphasize the importance of the alignment and traceability between the requirements and the design decisions. The impediment of these approaches is the lack of formal verification of the design decisions. Others approaches, such as [20] and [13], require users to learn additional formal languages, Event-B or the OCL. On the other hand, our approach offers a new perspective for visually doing alignment verification directly on the model, with only writing basic Scala expressions.

As other approaches, the previous work of Rychkova, [11] and [21], relies on manual mapping of SEAM constructs to Alloy\(^1\) verifiable code, and requires service designers to know Alloy and interpret the result from the Alloy Analyzer tool. In contrast, our work relies on writing basic Scala arithmetic and logic operations in the SEAM model, but does not capture behavioral properties.

III. MODELING EXAMPLE: MANAGE GAS LEAK

We illustrate our approach and we informally explain the semantics of our modeling language on a fictive example inspired by a real project conducted in a utility company. The utility company (UC) manages water, gas and electricity distribution.

In the example, we specify the design for a security service provided by UC for managing reported gas leaks. A regulation body sets the safety standards UC must respect concerning the time for securing a site where a gas leak is reported. The regulation is expressed as a service description: The UC must neutralize a gas leak reported by a witness, guaranteeing that a specialized UC team or a fire brigade arrives within 20 minutes and that the incident site is secured within 45 minutes from the time of the registration of the witness’ call.

A. SEAM Service Model Showing the Specification for Managing Gas Leaks

Fig. 1 depicts the SEAM model of the service offering specification level. We named this level Gas community and we specify three stakeholders with services they provide:

1) the Utility company offering the Manage gas leak service,
2) the Witness with the Report leak action (service), and
3) the Gas safety regulation body providing the Regulate safety of UC services.

This SEAM service model allows designers to conceptualize the utility company as a hierarchy of systems that provide services\(^2\). In this hierarchy, systems are conceptualized either as wholes, denoted with \([w]\) (black boxes), or as composites, denoted with \([c]\) (white boxes).

In a system as a whole, the system’s components are ignored and the focus is on the services offered by the system to its environment. A system as a composite shows the context and contains multiple systems as wholes whose services interact through a process. A process in a system as a composite gives the implementation of the corresponding service in the same system as a whole. In SEAM, processes allow for service collaboration and service exchange among systems. This is a direct application of the first foundational premise from service-dominant logic: “Service is the fundamental basis of exchange” [25].

In our example, all actors mentioned interact in the Manage site safety process and they belong to the Gas community \([c]\) system (see Fig. 1). The Gas community \([c]\) composing

\(^1\)Alloy [22], [23] is a language used to describe basic structures, as well as constraints and operations describing how structures change. It comes with an analyzer tool: a solver that graphically displays the structures modeled.

\(^2\)We use system to refer to an observed entity: an organization, a customer, an employee, an IT system, or an application [24]
systems as wholes are visible; the Utility company [w], the Witness [w] and the Gas safety regulatory body [w].

The main service of interest is the Manage gas leak, and Fig. 1 represents the specification of the service offering level. Fig. 2 illustrates how the UC is organized in providing the Manage gas leak service.

The composite view of the Utility company [c] depicted in Fig. 2 reveals the actors involved in the realization of the Manage gas leak service:

- the SAP application provides location coordinates for a given address,
- the ECS application monitors the actions of all agents within the UC and automatically contacts the fire brigade in certain conditions,
- the Dispatcher receives the witness call, enters the address of the reported leak in the SAP application, and dispatches the security and repair team to the location of the leak,
- the Security and repair team secures the gas leak, and
- the Fire brigade secures a site.

The Manage gas leak in the context of the Utility company [c] is a process that implements the corresponding service of the Utility company [w], therefore we call this a specification of the service implementation level.

B. Quantitative Properties in a SEAM Service Design Specification Model

The model from Fig. 1 and Fig. 2 is extended with properties (specifications of the requirements and constraints of systems’ actions). In our example, the Witness and the Gas safety regulation body are not concerned with the UCs internal operation and agreement, they only know:

- the time it takes for a specialist to arrive on site, and
- the time it takes to secure the site or the leak.

We need to refine these two times according to the values of UC actor’s properties. Let’s consider that the UC has the following agreement among its internal systems:

- The SAP application takes one minute to provide location coordinates for a given address.
- The Security and repair team, after receiving the call, needs (1) between 10 and 30 minutes to arrive on the site location, and (2) between 10 and 15 minutes to secure the gas leak.
- If the Security and repair team does not arrive on the site location after 15 minutes, the ECS application automatically contacts the Fire brigade.
- After receiving a call, the Fire brigade needs between 5 and 10 minutes to arrive on the site location, and firemen need between 10 and 20 minutes to secure a certain perimeter of the site.

It can be either the Security and repair team to arrive on the site, or the Fire brigade. The Security and repair team is dispatched by default after the first minute of receiving the call (the time it takes for SAP to retrieve the location coordinates of the leak). If the team does not arrive in the following 14 minutes to the site, the ECS application automatically signals the Fire brigade.

We already introduced an extension to the modeling language with a specification for four kinds of quantitative properties [12]. These properties have a stereotype and can only be connected to a service or a process.

Service Property Types: A <<final>> property specifies service requirements or constraints independent of other entities in the model. Properties defined in an Operational Level Agreement or an Underpinning Contract [3] are modeled with <<final>> properties. Fig. 3 depicts the final properties of the Fire brigade [w]. These properties are usually written as a Boolean expression over strictly one variable that gives a range, or even one value.

A <<computed>> property shows there is a corresponding service implementation process that specifies the computation. In the current context, the system as a whole, we do not know details about this computation. Fig. 4 depicts the computed properties of the Utility company [w].
Process Property Types: In our modeling language, the process shows a refinement of (1) a service offering to a service implementation, and (2) a system as a whole to a system as a composite. It connects and orchestrates services, so properties connected to a process expose the logic of the model specification.

A \( \text{\textless refinement\_relation\textgreater} \) property contains an expression that uses the values from the final properties of the connected services. The result of this expression is then transferred to the \( \text{\textless computed\textgreater} \) property of the corresponding service being implemented. Fig. 5 depicts the \textit{arrive\_on\_site} refinement relation property in the \textit{UC} \( \{c\} \), which is transferred to the computed property shown in Fig. 4.

A \( \text{\textless feasibility\textgreater} \) property gives the correctness of the level’s specification. The desired service outcome of a property, like a Service Level Agreement (SLA) used to specify “the metrics the client can use to monitor and verify the contract” (Spohrer, Maglio, Bailey, and Gruhl, 2007) is formalized as a Boolean expression and specified in a \( \text{\textless feasibility\textgreater} \) property. We use this property to define the alignment of a service as a conjunction of feasibilities: (1) from the level where the service is used in the system as a whole, and (2) from the level where the service is implemented in the system as a composite. Fig. 6 depicts the feasibility property of the \textit{Gas community} \( \{c\} \).

Fig. 7 depicts the complete SEAM model, specifying the service offering and the service implementation levels with all quantitative properties: \textit{arrive\_on\_site}, \textit{get\_geo\_info}, \textit{secure\_leak}, \textit{secure\_site} and \textit{site\_security}. The names of these properties and their complete expressions are used in the Scala code, hence the usage of underscores. In addition, we include an identifier in parenthesis next to the names of all services and processes, for example \textit{(ens)}, \textit{(sec)}, \textit{(sap)}, etc., with the aim to uniquely reference properties. This same identifier must be used in the service name and its corresponding process name to capture the references to properties from different services and to transfer values from refinement relationship to the computed properties.

IV. VERIFICATION OF SEAM SERVICE DESIGN SPECIFICATION MODELS WITH SCALA AND LEON

We first describe the concrete and abstract syntax of the SEAM modeling language, and then we give the mapping of the model elements to Scala code, used to:

1) generate a Scala object representing the model,
2) verify the Scala object with the Leon verification system,
3) and generate an output SEAM model to visualize the verification output.

A. Concrete Syntax

The concrete syntax of a visual modeling language is formed by the visual vocabulary (graphical symbols) and the visual grammar (compositional and well-formedness rules) [26]. The different graphical symbols of SEAM are partly described in [27] and are listed in Table I.

B. Abstract Syntax

Service designers who create models with our modeling language use the SeamCAD tool that is developed in our research laboratory [28]. SeamCAD uses an abstract syntax as a structure to save the models in an XML file. The abstract syntax is depicted in Fig. 8.

Fig. 8. The meta-model of the abstract syntax of our modeling language used in the SeamCAD tool.

All the systems, actions and properties visual elements inherit from the \textit{Node} class, and the link elements (lines) inherit from the \textit{Edge} class.

C. Leon and Scala

Scala [29], [30] is a functional programming language that implements “means of design-by-contract style specification of pre- and postconditions on functions” [31]. The \textit{require} clause is used to express a precondition, and ensuring a post-condition.

Leon is a system that does software verification, program synthesis and program repair for a subset of the Scala programming language [32], [33], called Pure Scala. Leon provides
to Scala programmers the convenience to use existing Scala clauses to write specification constructs in Pure Scala, without special training in formal logic. For each function written in Pure Scala, Leon generates a verification condition\(^3\) from the require precondition and ensuring post-condition clauses and tries to prove it [33]. To solve the generated verification condition, Leon combines an internal algorithm with external automated theorem proving tools [35] and CVC4 [36]. For each function, Leon’s output can be: (1) valid, (2) invalid if there is at least one counter-example, and (3) unknown. The unknown result is usually due to a timeout or an internal error.

For constraint solving, Leon introduces the choose construct. Choose is used to solve a constraint (a Boolean expression) for a given value [37]. The expression choose((res: B) => C(res)) evaluates to a value of type B satisfying the constraint C [37]. With using choose in a function’s pre- or post-condition, Leon can generate a counter-example that satisfies the constraint in choose, but violates the function’s verification condition.

We have developed a tool that maps the model properties to Scala functions with pre- and post-condition constraints. Afterwards, our tool runs Leon on the generated Scala functions to verify that for any input that satisfies the pre-condition, the post-condition is valid after the function execution.

D. Model-to-text (Scala Code) Mapping

The generation of the Scala code from the model starts with parsing the SeamCAD XML file and building data structures for the services and processes. Using these data structures, three main entities of the code are generated: Scala case classes for the properties, Scala values (val\(^4\) variables) for the services and Scala functions for the processes.

The names for the Scala values and functions are generated from the services’ and processes’ annotations in the SEAM model, which is the text in parenthesis. These annotations are used as identifiers that allow to map the values from the verification output back to the SEAM model.

Generation of Scala case classes for SEAM properties: Two Scala case classes, P and R, are generated as containers for the SEAM properties. They act as containers for the values that are used to find counter-examples for the <feasiability> properties. The P class contains all the variables that are defined in the <final> properties. The R class contains all the variables that are defined in the <refinement_relation>.

\(^3\) A verification condition is a statement of the form precondition action postcondition, which means that “if state \(x\) satisfies precondition and action transforms \(x\) to \(y\), then state \(y\) satisfies postcondition” [34].

\(^4\) 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.” [29].
TABLE I
MODELING LANGUAGE VISUAL VOCABULARY

<table>
<thead>
<tr>
<th>Systems</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utility company</td>
<td>A company, a department, an organization</td>
</tr>
<tr>
<td>SAP application</td>
<td>IT application, program module, IT component, IT platform</td>
</tr>
<tr>
<td>Witness</td>
<td>Human actor</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Actions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manage gas leak</td>
<td>An action of a system as a whole</td>
</tr>
<tr>
<td>Manage gas leak</td>
<td>An action of a system as a composite</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Properties</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;computed&gt;)\</td>
<td>A property of a service or a process, depending on the stereotype</td>
</tr>
<tr>
<td>time_to_secure</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Links</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A link between service-process, service-property and process-property</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Refinement link, from [w] to [c]</td>
</tr>
</tbody>
</table>

Properties. These two classes are implemented in Scala using case classes and we generate them by extracting information from the data structures obtained after parsing the XML file. The generated P and R classes for our example are:

```scala
case class P(arrive_on_site: Int, get_geo_info: Int, secure_leak: Int, secure_site: Int, site_security: Int)
case class R(arrive_on_site: Int, secure: Int, time_to_secure: Int)
```

**Generation of Scala values for SEAM services:** For each service present in the model, a Scala value (val) is generated from the service identifier in parenthesis. If the service has \(<final>\) properties, then the service value is defined by an instance of P. Non-defined properties are set to zero in the instance of P. Otherwise, the property value is computed with the `choose` operator explained in Subsection IV-C. In our example, the Secure gas leak (sec) service in the Security and repair [w] system is mapped to:

```scala
val s_sec = P(choose((arrive_on_site: Int) => arrive_on_site > 10 && arrive_on_site <= 30), 0, 0, choose((secure_leak: Int) => secure_leak > 10 && secure_leak <= 15), 0)
```

On the other hand, if the service has a corresponding process, i.e. service properties are \(<computed>\), then the service value is defined by a call to the function that maps that process (see next paragraph), returning an instance of the R case class. In our example, the Manage gas leak (mng) service in the Utility company [w] system is mapped to:

```scala
val s_mag = ps_mng(s_add, s_call, s_disp, s_fb, s_fb_sec, s_sap, s_sec)
```

where the `ps_mng` function is generated based on the mapping of the process in the model.

**Generation of Scala functions for the processes in the model:** Each process in the model is mapped to a Scala function. The input for this function represents the services connected to the process and the output is an instance of the R case class. Non-defined properties are set to zero in the instance of R. Otherwise the corresponding property is mapped to the expression present in the corresponding \(<refinement_relation>\).

The pre-conditions are generated to guarantee that the input properties of the process function match the connected service values.

The post-condition applies to the output instance of R and is generated directly from the \(<feasibility>\) property of the mapped process.

The generated function for the Manage site safety (safe) process from our example is:

```scala
  require(\(add == s_add && call == s_call && disp == s_disp && ccs == s_ccs && sap == s_sap && sec == s_sec\))
  R(0, secure_leak + sec.arrive_on_site + sap.get_geo_info)
} ensuring(res => res.time_to_secure <= 45)
```

**E. Scala Code Verification with Leon**

Our tool passes the generated Scala code of the model to Leon. Leon statically verifies it by checking the feasibility properties that are mapped to post-conditions. Counter-examples are generated, if any exist. Fig. 9 illustrates Leon’s output in the terminal of the verification pre- and post-conditions for our example. There exists one invalid post-condition in our SEAM service design model: the process annotated with \(<safe>\).

![Leon output in terminal.](image)

**F. SEAM Model of the Alignment Verification Output**

It is difficult for the user to interpret the output from the terminal and to locate the problem in the model (see Fig. 9). Therefore, our tool automatically re-transforms the Leon output to a SeamCAD file. We parse the Leon output and
with the help of the annotations in parenthesis in the service and process names, we mark the faulty process with red color as in Fig. 10. The concrete values of counter-examples found are placed on the link between the process and the service. By doing this it is possible to graphically visualize the problem and identify the exact source of the misalignment. Fig. 10 shows the verified SEAM model of our example. It is clear that the feasibility property of the Manage site safety process in the Gas community [c] system is violated.

The cause of this violation is the arrive_on_site time requirement. The time set by the Gas safety regulation body [w] system concerning the arrival on the site is used in the feasibility property in the service offering level, but this constraint is not transferred to the service implementation level, i.e. in the Utility company [c]. Service designers can use this visual verification output to further refine their specification in a way that their design choices are aligned among different specification levels. For example, service designers of the Utility company should

1) include the restriction from the regulation body in the feasibility of the service implementation level, and
2) negotiate the operational level agreement with the Security and repair team, or the underpinning contract with the Fire brigade.

V. LIMITATIONS

The current implementation of the SeamCAD tool, and our automatic mapping and verification tool do not check for model’s well-formedness. In case of a modeling mistake, the model-to-text code could not generate the necessary data structures used in the Scala code generation. In addition, for successful automatic verification of SEAM service specifications, the model properties must be basic Scala Boolean expressions. Our tool directly maps all properties’ values to Scala code. In these cases, the service designer is left to find the problem based on the console output.

Service designers must also be attentive on the measurement units of quantities put in the properties. In our example we use minutes, but we do not specify in the model that all quantities are minutes. Consequently, every occurrence of a property variable must have the same unit of measurement and must be expressed in the same order of magnitude. Our tool is not able to detect or perform automatic conversion of units of measurement. It is however possible to define multiple properties in the same model to capture different units.

Finally, Leon’s support for only Pure Scala programs limits the expressiveness of service specifications in the service design model as well. Currently it is impossible to use external libraries that deal with dynamic units of measurement conversions and static type checking.

VI. FUTURE WORK

We plan to apply our approach in a real project. Many projects we encounter deal with non-functional requirements, such as security and quality, so we are researching ways of representing and quantifying such properties. In addition, we are working on including the previous work [21] on verifying behavior properties in SEAM models.

The example we show is a simple one, as our goal is to present our approach in details and to formally describe what we mean by alignment. With SEAM, service designers are not bound to specify only two levels of a service. Every service in a system as a whole can be refined, and the specification of its implementation can be modeled. In this recursive manner, SEAM is used to show specifications from the service strategy level up until the lowest IT level. We still need to conduct an evaluation (e.g. user study) of the usefulness of the automated verification with SEAM. Our current expectation is that in combination with Leon our approach has the potential to become a powerful verification tool in the field of MDD.

VII. CONCLUSIONS

We present a model-driven approach for automatic alignment verification of quantitative properties modeled in SEAM service design specification. We demonstrate service modeling on a simple example: services modeled at two levels. A feasibility property is specified to capture the desired behavior of services in each level. Feasibility properties of all levels are used check the alignment correctness of each level and among levels. We also show refinement of properties from the service offering (higher) level to the service implementation (lower) level. The tool for this automated verification:

1) takes a SEAM model as an input,
2) translates the model to Scala code,
3) verifies the code with Leon, and
4) displays the result in a new annotated SEAM model.

The novelty of our approach is using software verification tools in the early stage of service design. The practical contribution emerges from using Leon; service designers do not need to learn a formal language, but they only need to know SEAM modeling and basic Scala expressions to capture and verify their design choices. The model-to-text translation and the verification are running in the background, so there is no overhead in learning and manually transforming the service specification to a formal model. The visualization of the verification result in a SEAM model gives feedback about the cause of the misalignment and allows to refine the service specification.

SEAM recursively uses the same notation, so service designers are free to model any number of levels. Our approach is applicable to the complete organizational hierarchy, starting from the business down to the IT level. Therefore, because it is a visual approach, a wider audience is able to benefit from designing correctly aligned service specifications among different levels of the organizations and IT systems (such as students, service designers, software architects and developers).

REFERENCES
