Integrating Executable Requirements in Prototyping

Yann Le Goff[1,2,*], Pierre Laborde[2], Éric Le Pors[2], Mickaël Kerboeuf[1] and Alain Plantec[1]

[1]Univ. Brest, Lab-STICC, CNRS, UMR 6285, F-29200 Brest, France
[2]THALES Defense Mission Systems France - Etablissement de Brest, 10 Avenue de la 1'ere DFL, 29200 Brest, France

Abstract

At Thales DMS, the application creation process involves several teams. The Systems engineering team produces a set of requirements that the final application must meet. Implementing an application in accordance with these requirements can be very tedious and time-consuming. To facilitate the development of the final application, a prototype can be implemented as an intermediate representation of the final product. One of the main difficulties is linking the requirements to the elements of the prototype to ensure that they are taken into account and are meaningful.

In this article, we present our meta-model for specifying requirements and how we use requirements models to verify the prototype during its implementation through the execution of *Tracing Points*. We implemented Tracing Points in Pharo. We explain how Tracing Points are integrated and executed within the prototype code to verify the corresponding requirements.

Keywords

Requirements, System Engineering, UX, Prototype, Executable model, Pharo

1. Introduction

At *Thales Defense Mission Systems* or *Thales DMS*, the application creation process involves several engineering disciplines and can last several years. Among the disciplines are Systems Engineering and User Experience (UX) , each with a dedicated team. Each team produces multiple requirements, in different languages, with different goals. A requirement describes what is needed without defining specifics of implementation [1].

The Systems Engineering team is responsible for capturing user needs and transforming them into a set of requirements. Other teams use these requirements to design and develop the final application. The UX team is in charge of formalizing Human-Machine Interface (HMI) requirements, often in the form of a prototype. The prototype is an executable software that is representative of the HMI graphics and ergonomics of the final product, and can be used as a requirement model of the HMI. The Software Engineering team designs and develops the software based on the requirements of the Systems Engineering team and the designs and ergonomics proposals of the UX team.

Currently, the traceability of requirements between the different teams is not fully supported by tools . This lack of proper tooling can generate errors and synchronization issues as the requirements change during the several years of creation and sometimes are no longer coherent in between them. The HMI requirements of the UX team are defined by taking into account the System requirements. The absence of traceability between the two makes justifying the HMI requirements difficult to all stakeholders. Furthermore, better traceability would help to forecast the impact of changes in System requirements on HMI functionalities.

We propose tracing the requirements produced by the Systems Engineering team in the prototype developed by the UX team. We create an executable requirement meta-model that is linked to the prototype source code. The executable requirements are characterized by assertions injected into the prototype code that will be evaluated at runtime while the prototype is being executed.

IWST 2025 — International Workshop on Smalltalk Technologies Gdansk, Poland; July 1st to 4th, 2025 Co-located with ESUG 2025
*Corresponding author.
yann.le-goff@thalesgroup.com (Y. Le Goff); pierre.laborde@thalesgroup.com (P. Laborde); eric.lepors@thalesgroup.com (: Le Pors); kerboeuf@univ-brest.fr (M. Kerboeuf); alain.plantec@univ-brest.fr (A. Plantec)

In this article, we present the context of Thales DMS and define the different models produced by the Systems Engineering team and the UX team (Section 2). We define the executable requirement meta-model (Section 3). We present our implementation of our model in the Pharo language with an example (Section 4). We discuss the limitation of our proposition and our current implementation (Section 5). We discuss the related works (Section 6). Finally, we conclude and present our future works (Section 7).

2. Context

In this section we will present different models created by the System Engineering team and the UX team, and how this models are used by the software Engineering teams. The models are complex, define in different formalism, uses specific methods, techniques and tools [2]. The models will be presented throughout an authentication example.

2.1. System Engineering

The System Engineering team is responsible for capturing user needs and transforming them into a set of requirements. Other teams use these requirements to design and develop the final application. These requirements may change throughout the development process [3, 4].

Arcadia [5] is a System engineering method used to define the needs of the users and to design the solution's architecture. Arcadia is used to describe with precision what the system needs to accomplish, how it will accomplish it, how it will be built. Capella is a software used by the system engineers to apply the Arcadia method. Capella produces a series of models of different form with different goals, each model element and their associations represent a part of the system requirements. For instance, you will find in Capella model the following elements:

- **Actors** - The different persons that will interact with the system.
- **Capabilities** - The expected capabilities of the system.
- **Functions** - The different functions that resolve this capabilities.
- **Components** - The different components of the system, each component has functions.
- **Functional Exchanges** - The exchanges in-between functions.
- **Data model** - The different information that are transmitted in a functional exchanges.
- ...

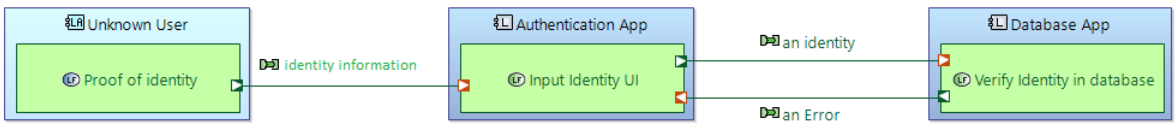

Figure 1: Example of a Capella file edited in the Capella software.

The figure 1 shows a Logical Architecture Breakdown diagram (LAB) of the example. The LAB can be used to describe the different components and actors of the system (the *Unknown User*, the *Authentication App* and the *Database App*), the associated functions (*Proof of identity*, *Input identity UI* and *Verify identity in database*) and the different functional exchanges (the line between functions on the figure).

On the left, we have an actor named *Unknown User* that wants to prove its identity to the system. To prove its identity, the unknown user will provide *an Identity* to the *Authentication App* through the functional exchange between the functions: *Proof of Identity* and *Input Identity UI*.

The Authentication App transmits the identity to the *Database App* that will verify if the identity is correct or not. In the case of an incorrect identity, the Database App sends to the Authentication App an *Error* in order to inform the user of the incorrect nature of the provided identity.

Other aspects of the system are not represented in the figure 1, for example, what happens if the provided information is correct. In the rest of the article, we will only focus on the Error aspect of this example.

2.2. User Experience

The UX team is in charge of formalizing HMI requirements, their activities involve several stages: assisting in the capture of user needs, creating static mockups, developing prototypes [4]. The prototypes are used as a support to discuss HMI issues and obtain early feedback with all project stakeholders including engineers, clients, and users [4, 6, 7, 8, 9, 10].

The users of the application participate in the design and evaluation of the prototypes [4, 8]. The prototypes highlight design issues present in the Systems Engineering requirements [6, 8] and help to anticipate problems and architectural needs before the industrialization of the system. They work either by connecting to existing system functions or by simulating the expected behavior of the final application functions. The prototypes can then be used as model specifications for the UX for Software Engineering, in charge of developing the final application's HMIs [7, 9, 10].

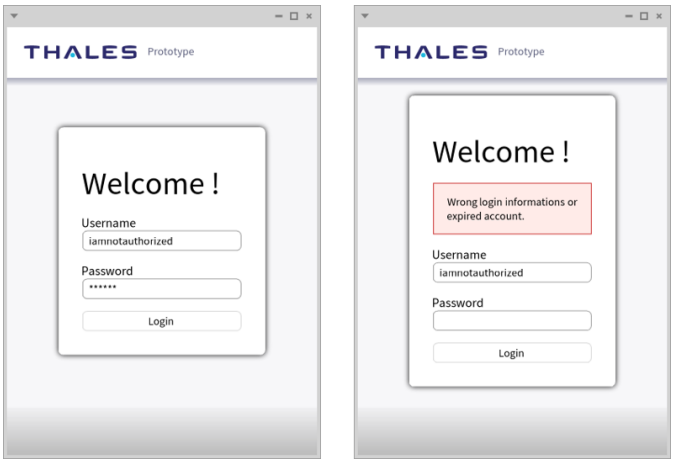

Figure 2: User Interface of a login panel for authentication.

The figure 2 shows the views of the User Interface (UI) of the authentication example. On the left, the UI of the prototype before receiving an Error. On the right the same UI after receiving an Error, it is receive when the provided information (username and password) are incorrect. The UI is composed of a panel, the panel features an error box in red (not visible by default), two text fields for entering the username and password, and a *Login* button to send the authentication message to a simulated database app.

3. Proposal

We propose an executable requirements meta-model integrated into the prototype (figure 3). Instances of this meta-model allows us to create links between the requirements provided by the System engineering team and the code of the prototype. The system requirements are not precise enough to explicitly describe what the code must do. It is necessary for the UX designer to make design choices for the

application (directly in the prototype code) and then justify these choices by linking them with the system requirements (using our model). Our model can be used:

1. To ensure that the System requirement can be linked to the prototype, and therefor are meaningful enough to design and develop the final application's HMIs,
2. To ensure that the System requirements are taken into account in the code and during the execution of the prototype.

Each link can be associated with preconditions and postconditions that assert that execution of the prototype satisfies the requirement. The user of the model have the responsibility to write the precondition and the postcondition to evaluate the requirements.

- **Precondition** - *An assertion attached to a routine, which must be guaranteed by every client prior to any call to the routine* [11].
- **Postcondition** - *An assertion attached to a routine, which must be guaranteed by the routine's body on return from any call to the routine if the precondition was satisfied on entry* [11].

In the *Floyd-Hoare Logic* [12], the assertions are used to prove the correctness of a routine. *"If the assertion P* [the precondition] *is true before initiation of a program Q* [the routine]*, then the assertion R* [the postcondition] *will be true on its completion".* A routine in this context correspond to *a software unit that other units may call to execute a certain algorithm, using certain inputs, producing certain outputs* [11].

In the *Eiffel* language [13], assertions are applied to a routine and serve to define a contract that explains how the routine must be called and what are its outputs. The contract can be used to verify if the routine is called correctly and to anticipate all possible outputs of the method.

Certain requirement can be interpreted by the UX team as a state transition within the HMI. In our model, the precondition and postcondition are employed to verify the state of the prototype HMI. To verify the presence of a requirement in the prototype, its HMI must be in a specific state. The precondition is used to confirm this state, and unlike in Eiffel or Floyd-Hoare Logic we expect sometimes the precondition to be unverified if the HMI state is not meet. The requirements can also modify the state of the HMI, and the postcondition serves to ensure that the state has been modified. If a precondition is true, we expect the requirement to be verified and the postcondition to be also true like in Eiffel or Floyd-Hoare Logic.

In our model, preconditions and postconditions are employed to verify the state of the prototype's HMI. Certain requirements can be interpreted by the UX team as state transitions within the HMI. To verify the presence of a requirement in the prototype, the HMI of the prototype must be in a specific state, which is confirmed using the precondition. Unlike in Eiffel or Floyd-Hoare logic, the precondition may sometimes remain unverified if the HMI state is not met. Requirements may also modify the state of the HMI, we use the postcondition to verify it. When the precondition is true, the requirement is expected to be satisfied, and the postcondition should also be true, like in Eiffel and Floyd-Hoare logic principles.

The model is composed of two parts: - a static part that defines the requirements, the assertions - and a dynamic part that manage the execution of the assertions and generates reports on the results of this assertions.

3.1. Static model

The requirements of our meta-model is based on the abstract requirement defined by Pohl [3]. Our requirement is composed of a title, a description and a source document. We do not address yet the other aspects of the abstract requirement at this time.

We extended the requirement with verifications, each verification is composed of a collection of ordered steps. Each step contains a reference to the code of the prototype (named *Node* in figure 3) and can include a precondition and/or a postcondition (named *Assertion* in figure 3) that will be executed with the current context of execution.

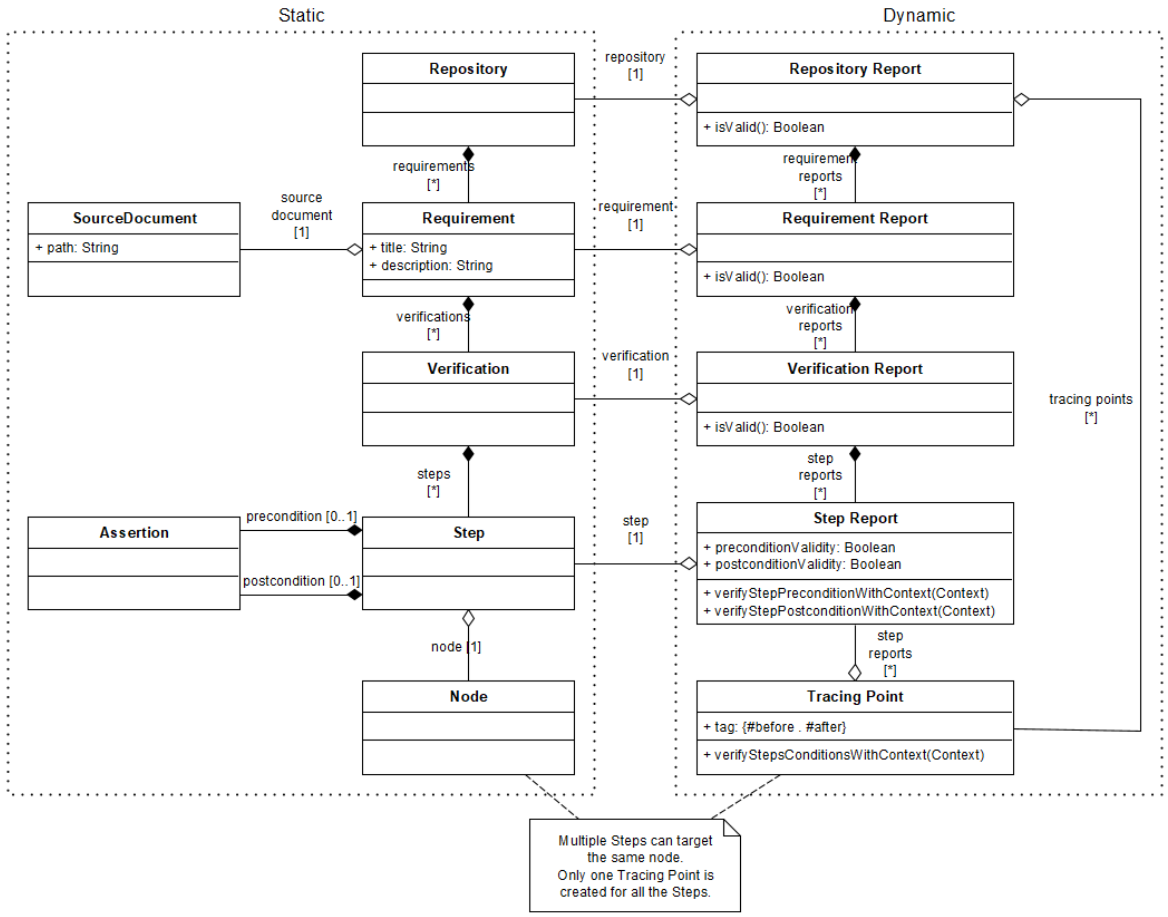

Figure 3: The class diagram of the executable requirement meta-model.

3.2. Dynamic model

The dynamic model transforms all the different steps into *tracing points* and integrates those tracing points into the source code of the prototype. It also produces reports that are updated by the tracing point during the execution of the prototype. The reports can then be used to detect the presence and the absence of the requirements in the execution of the prototype.

Tracing point We call *nodes* the Abstract Syntax Tree (AST) nodes of the source code of the prototype. We define a tracing point as an annotation of a node that can be executed just before or just after the node. After adding a the tracing point to the node, the language environment will be able to execute the different preconditions and postconditions with the different contexts of execution. The contexts of execution includes: the different variables, the senders and receiver of the addressed AST node.

For example in the code in figure 4, the targeted AST node is the one highlighted at line 9. When the code at line 9 is executed, the precondition and the postcondition can access the value of the variable named *simulationResult*, they can access the code of the sender and the receiver of the message *refuseLogin*. In the context of figure 4, the receiver and the sender are the same object: *self*.

The tracing point when executed evaluate all preconditions and postconditions associated to it and update the associated reports. The precondition are basic to evaluate, before the execution of the node, we evaluate the precondition using the current context of the executed code.

The postcondition is challenging. After the node is evaluated we evaluate the assertion using the current context. But, in the case of an early return or an exception: the code stop its execution before being able to evaluate the postcondition. We need to take into account these special cases. Our current implementation only works in the nominal case and with the early return, if the code signals an

exception the postcondition will not be evaluated. We discuss the exception case in the section 5.2.

```
loginButtonAction

   | simulationResult |
   simulationResult := simulation
      canAuthenticateUsername: self username
      password: self password.
   simulationResult isError
      ifTrue: [ self validateLogin ]
      ifFalse: [ self refuseLogin ]
```

Figure 4: Example of a targeted node in the Pharo language.

Reports Each step is transformed as a report, the goal of the report is to inform the user of the model that the requirement is present inside the prototype and the associated code, precondition and postcondition are indeed evaluate during the execution. We use the targeted source code of each step to create the different tracing points, we create a step report for each step and link this report to the associated tracing point.

To know if a requirement is valid, we create the different reports and tracing points from the steps of the requirement. We then execute the prototype, and at runtime the tracing points will be evaluated and the report will be updated. A requirement in the execution of the prototype can be verified when all associated reports are valid. A reports is valid when the precondition and postcondition are valid in the given execution context of the associated tracing point. A postcondition is evaluated only if the precondition is valid. A precondition is evaluated only if the previous step report is valid. If either the precondition or the postcondition is invalid, we consider the report invalid and both assertions will be evaluated again next time the targeted AST node is evaluated.

We can verify that the requirements are sufficiently accurate to design and develop the HMIs because we can identify the specifications of these requirements within the prototype code. We can ensure that the System requirements are properly considered because all the requirements that concern the HMIs should be linked to code parts of the prototype. This meta-model allows us to compose UX requirements present in the prototype with other engineering requirements and help us to justify UX design choices to the other engineering teams.

4. Validation

Our prototypes are developed in the Pharo [10, 14, 15] programming language. Therefore, we implemented the executable requirements meta-model in this language. We extended the *Debug Point* system to create what we call *Tracing Points* using inheritence. The Debug Point serves as the breakpoint system of the Pharo debugger. To illustrate the application of our model, we developed a simple authentication page, employing Capella for the system requirements and Bloc [1] / Toplo [2] for the prototype development. This example demonstrates how the meta-model operates in a practical scenario.

4.1. Implementing the meta-model into Pharo

We implemented the meta-model in the Pharo language and environment because it is the same language used to create prototypes. Having the model in the same language allows us to use a single syntax for developing the prototype, as well as for writing the preconditions and postconditions of the requirements.

[1]Bloc is a UI library for Pharo; Github repository of the Bloc library: https://github.com/pharo-graphics/Bloc
[2]Toplo is a Widgets library for Pharo; Github repository of the Toplo library: https://github.com/pharo-graphics/Toplo

4.1.1. Implementation of the model

```
1  capella_requirement_3
2
3     <ExReqToploExample>
4     ^ ExReqRequirement new
5           title: 'database shall send "error" to Authentification app';
6           description: 'The component named "database"...';
7           sourceDocument: (ExReqCapellaDocument new
8                  id: '141c2297-10ac-4e90-82c4-83fc9c7b3b14';
9                  name: 'login_system';
10                 yourself);
11          addVerification: [ :verif |
12             verif
13                addStepOnAST:
14                   ((ExReqToploLoginApplication >> #loginButtonAction) ast allChildren at: 21)
15                withPrecondition: [ :obj | obj card isLoginStatusContainerVisible not ]
16                withPostcondition: [ :obj | obj card isLoginStatusContainerVisible ] ];
17          yourself
18
```

Figure 5: Example of a Requirement description with a verification and a step in the Pharo language.

The static and dynamic parts of the model are straightforwardly created in the Pharo language. A method annotated with a specific pragma [3] returns a requirement. All methods with the same pragma form a requirement repository. This organization enables us to put all the requirements of a project within one package, while placing the project itself in others, allowing for the delivery of the software without the associated requirements. In the figure 5: the pragma is visible at line 3; the title of the requirement at line 5; the description at line 6 and the source document at lines 7 to 10.

The requirement can be automatically generated from a source document, for example from a Capella file like the one figure 1. We only have to add manually the different verifications. In the figure 5 we added a single verification at lines 11 to 16. This verification has a single step and this step has a precondition at line 15 and a postcondition at line 16. The figure 6 shows an instance diagram of this requirement and its associated reports and Tracing points.

Links between our model and the code The code at line 14 in figure 5 is used to targeted the specific node is obtain using a small tool that allows us to select any node in a code browser, transform it into the targeting code and copy it to the clipboard. We only have to paste it inside the verification code. This step is not an automated process, if the method that contains the targeted node changes, we would need to manually update the requirement.

About the preconditions and postconditions We used *Block Closure* or a block to define our precondition and postcondition. A block is a lambda function [4]. The blocks return a boolean that represent the validity of the assertion. They can have 3 arguments: the context receiver (the object that execute the targeted node); the context arguments (all arguments of the parent method node); the associated requirement.

In the language *Eiffel* [13], the unary operator "*old*" represent the value of the context before the execution of the node. It is used only in the postconditions, to assert the modifications of a variable during the execution of the node. In our meta-model we do not save the context state, therefor we cannot have a similar mechanism as "old".

[3] Pharo documentation on pragmas: https://github.com/pharo-open-documentation/pharo-wiki/blob/master/General/Pragmas.md

[4] A Pharo blog post on block closure: https://thepharo.dev/2020/04/27/about-blocks-basics-101/

4.1.2. Implementation of the Tracing Point

The Debug Point system allows us to extend any node of the AST for execution control, sending a break signal to the Pharo environment. The Pharo environment will then stop the execution of the code and open a debugger, which facilitates immediate analysis and correction of the code execution.

To create the Tracing Point, we extended the Debug Point system of Pharo. In order to manage early returns inside an AST node, we added a *safe postcondition tracing point*. In total, we use three tracing points for each step:

- **The precondition tracing point:** This tracing point is installed just before the AST node. It evaluates the preconditions of the steps. This tracing point is visible in the figure 6, named: *for precondition.*
- **The postcondition tracing point:** This tracing point is installed on the AST node with the **after** control tag. This tag ensures that the tracing point is evaluated after the AST node. This tracing point evaluates the postconditions of the steps. Visible in the figure 6, named: *for postcondition.*
- **A safe postcondition tracing point:** This tracing point is installed on the AST node of the parent method node of the targeted AST node with an **after** tag. This tracing point is necessary in the case of an early return inside the targeted AST node. It ensures that the postcondition is still evaluated in the case of an early return, which could otherwise prevent proper validation. Visible in the figure 6, named: *for safe postcondition.*

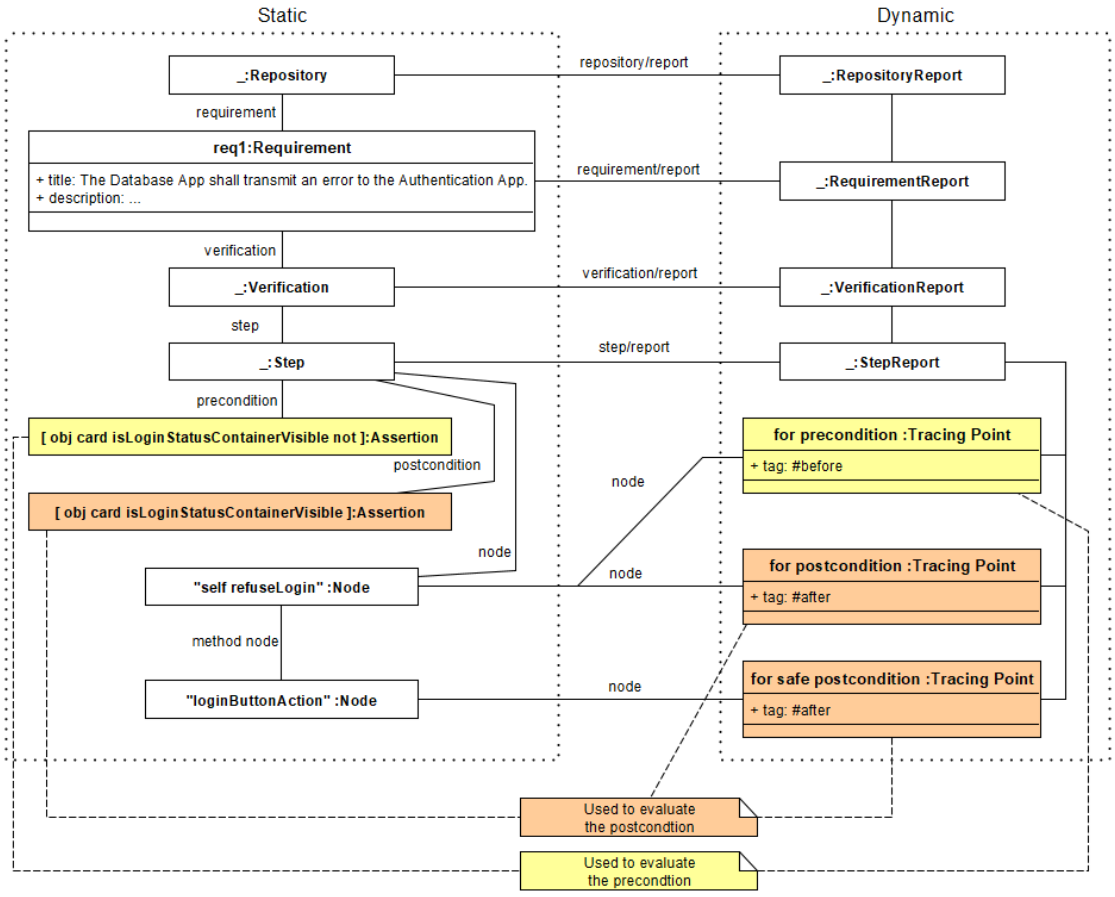

Figure 6: Instances of the meta-model with the different Tracing Points.

4.2. Creating an Example as Proof of Concept

In this section, we present a Capella model produced using the Arcadia method and the Capella software. We then translate it into requirements. We will link these requirements to a prototype. Finally, we will execute the requirements model and the associated prototype to validate the presence of the requirement in the prototype.

Context of the example We present an authentication example. We use the models describe in section 2:

- **System Engineering models:** we use the Capella model of the figure 1. In the figure the Database App can send an Error to the Authentication App. We except the system, and therefore the associated HMI, to inform the users if they have provided wrong identity information.
- **UX models:** we use the prototype presented in the figure 2. We except the HMI to be able to display an Error message if the users provide wrong identity information. We added to the prototype *Login* button a controller, its code is presented in the figure 4. The controller sends the username and the password provided by the users to a simulation (lines 4 to 6). It then verifies the result provided by the simulation (lines 7 to 9). If the result is an Error, it executes the method *refuseLogin* (line 9), in this method it will modify the HMI to display the Error message.

Translation of requirements We created a process to import the elements of Capella models into the Pharo environment. The different steps are:

1. We create a model using the Capella software.
2. We used the standard XML library of Pharo [5] to import the contents of the Capella file into the Pharo environment. (The Capella software saves its models as XML files).
3. We then translated this xml elements into a simplified version of the Capella model created in Pharo. This simplified version is used to remove lots of complexity of the capella elements. It still have the notion of components, actors, functions, and functional exchanges.
4. From this simplified model we apply a selection function to select specific elements from the model. In the context of prototyping, we will select all elements that represent or interact with the User Interface we want to describe.
5. We then transform this selection into requirement instances of our model.
6. We used an instances to code serializer [6] to persist the requirements model in the Pharo source code and to be able to manually to add the missing verifications.

The translation of the simplified Capella model to requirement will generated a title, a description and the source document for each requirement. We only need to add the verification steps, preconditions, and postconditions by hand.

We created a simple example with Capella and we imported it in Pharo to create the different requirements needed for our example. In our authentication example, we select the functional exchange corresponding to the Error send by the Database to the UI. We then translate it into the following requirement: *The Database App shall transmit an error to the Authentication App.*

[5] GitHub repository of the standard XML library in Pharo: https://github.com/pharo-contributions/XML-XMLParser
[6] GitHub repository of Stash (serializer instances to code): https://github.com/Nyan11/Stash

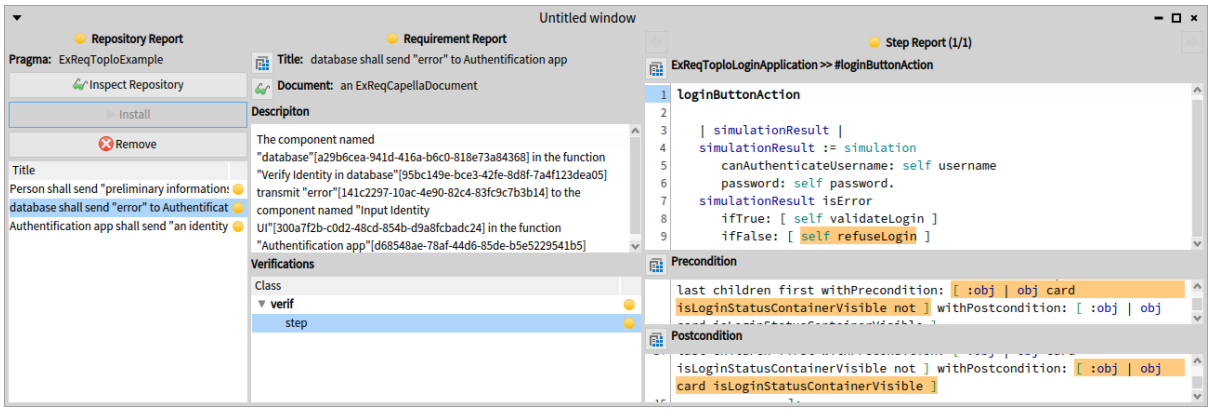

Figure 7: UI created in Spec2 to install and remove the tracing point and to display the status of the reports.

Adding the verification The Error is sent by the database to the authentication app when the information provided by the users does not allow to authenticate them. Thanks to the tracing point system, we can target any node inside an AST of any method. We will target the highlighted node in the figure 4. This node correspond to the reception of an Error message from the simulated Database to the Authentication UI. We will add one precondition and one postcondition to this node:

- **Precondition:** we must ensure that the error box is not visible. The code of the block closure is visible in figure 5 at line 15.
- **Postcondition:** we must ensure that the error box is visible. The code of the block closure is visible in figure 5 at line 16.

Execution of the Requirements Model and the Prototype In order to test if the requirements is indeed present in the prototype, we install the different tracing points inside the code of the prototype and execute the prototype. The user will manually enter a wrong username and password, then click on the login button. The login button will call the method *loginButtonAction* (figure 4). It will receive an Error form the simulated Database and execute the code in the conditional branch of line 9, where we installed the verification.

Before the execution of the highlighted node, the prototype evaluates the precondition to ensure that the error box is not visible. Then the prototype executes the highlighted node, in the method *refuseLogin* the prototype will display the Error box. Finally, the prototype executes the postcondition and verify that the error box is present. Because both precondition and postcondition are verified, we can conclude that our prototype indeed implements the requirement: *The Database App shall transmit an error to the Authentication App.*

We created a User Interface in Spec2 [16] (see figure 7)that allows us to easily see and install the tracing points in the prototype. This simple interface presents which step reports are successful and which are not. This view helps us to identify the different requirements that still need to be integrated into the prototype.

5. Limitations

5.1. Requirement version management

As discuss before, the requirements evolve during the design and development phase of the system. This evolution can cause troubles, for instance, a HMI design to specifics sets of requirements can be invalid for a different set. Our current model allow us to *fix* a certain version of the requirements, and to help us understand its impact on the global HMI. But we do not have a mecanism that allow us to keep track of the different versions and the evolution of the requirements.

Possible Improvements Adding a versioning system to the requirements would allow us to see the evolution and help us to justify the different evolution of the HMI according to the corresponding requirements.

5.2. Exceptions

When a node is evaluated, it can have one of the following behaviors: execute the next node in the code, execute an early return or trigger an exception. If a node trigger an exception, the execution is interrupted and the associated postconditions are not evaluated. Therefor, we cannot instrument a node that trigger an exception with the tracing point. And we cannot verify a requirement that needs to test if a node trigger an exception.

Possible Improvements In order to catch the exception to evaluate the postcondition, we need to intercept the context of the method that sends the exception. We attempted to do it with MethodProxies [17], but the uninstrumentation created bugs between the DebugPoint system and the method proxies system.

6. Discussion and Related Works

We have choose to extend the Debug Point system to implement our executable requirements model in Pharo. Other solutions could have been used.

The advantage of the DebugPoint The use of the debug point system allows our tracing points to be installed on any AST node of the source code thanks to the debug point. The debug point also allows to target a specific instance in the case of object centric debug point. We could in the future implement the principle of object centric requirement traces that only target a specific instance of a class of objects. The difficulty of selecting the correct instance to install an object centric tracing point is similar with object centric debug point. Also we still have to find if such functionality present any interest in the industrial process. We can imagine a scenario where we select one specific instance of a button in the UI and not target the associated button class.

Not Selected Solution - Use of MetaLinks *MetaLinks* are used to annotate AST nodes. An annotated AST is expanded, compiled and executed on the fly. The Debug Points create and use MetaLinks to implement their functionalities.

We choose to extend the DebugPoints in order to have a simplified API to work with and to have access to all the installation and removal methods for the associated MetaLinks. In our solution we change the default MetaLink created by the DebugPoint to add the **after** control tag for the postconditions.

The MetaLink offer a lot of options we did not try yet in our model, for instance the tag **optionWeakAfter** that should allow us to manage the exceptions.

Not Selected Solution - Use of Method Proxies We could have used *Method Proxies* [17] to implements the tracing points. But Method Proxies [7] can only target a method and not a specific AST node. For example, our solution with the extension of the debug point can target a specific node in a condition (as seen in figure 4). Targeting a specific node is necessary in the UI domain where methods can be very long and we only focus on a small part. For example, a theme is a set of rules that define the look and feel of the different element of the UI. A single method of a theme can do more than 400 lines of code and contains more than 10 different rules. Using a Method Proxy to target a specific rule in the 400 lines of code will be complicated, our solution can target the associated node of the rule.

Method Proxies manage to intercept exceptions and could have managed easily the "postcondition". The exceptions are sometimes used by the HMI developers to manage the different states of the UI.

[7] GitHub repository of Proxy Methods: https://github.com/pharo-contributions/MethodProxies

We try to implement the tracing point by extending the debug point and to manage the exceptions using the Method proxies but we did not managed the installation and the removal of both the method proxy and the debug point at the same time. Using Method Proxies break the link between the method and its AST, stopping the installation of the debug point. Removing the debug point, force the recreation of the proxified method, duplicating the method.

Not Selected Solution - Directly Add Annotation in the Code We could have directly added traces in the code of the prototype as a form of annotation. For example, in the Eiffel language [13] they added a *"require"* and an *"ensure"* clauses to the routine that describe the precondition and the postcondition. Pragmas could be used for this.

However, we chose to extend the Debug Point system because it allows us to decide to add the traces in the code or not. This approach removes the complexity of managing links outside the functional code of the prototype (at the expense of increasing the complexity of the requirements model). Because there are no references to requirements in the prototype code, we can deploy the prototype without the requirements, facilitating a cleaner separation between functional implementation and requirement tracing.

7. Conclusion and Future Works

There is a lack of tooling for traceability of requirements between the System engineering team and the UX teams. We propose a requirement model that will link the system requirements to the code of the prototype created by the UX team. This model allows us to add assertions and to verify the presence or the absence of the requirements during the execution of the prototype. We develop a simple example of an authentication prototype to test our model and we manage to verify the presence of the associated system requirement when using the prototype. We plan to deploy this model on larger scale through other prototypes.

Declaration on Generative AI

During the preparation of this work, the author(s) used GPT-4o-mini in order to: Grammar and spelling check, Text Translation, and Improve writing style. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] P. J. Schubert, L. Vitkin, F. Winters, Executable specs: What makes one, and how are they used?, SAE Transactions (2006) 705–714.

[2] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, J. Bezivin, Inter-DSL coordination support by combining megamodeling and model weaving, in: Proceedings of the 2010 ACM Symposium on Applied Computing, ACM, Sierre Switzerland, 2010, pp. 2011–2018. URL: https://dl.acm.org/doi/10.1145/1774088.1774511. doi:10.1145/1774088.1774511.

[3] K. Pohl, Requirements engineering: An overview, RWTH, Fachgruppe Informatik Aachen, 1996.

[4] P. Hsia, A. Davis, D. Kung, Status report: requirements engineering, IEEE Software 10 (1993) 75–79. doi:10.1109/52.241974.

[5] J.-L. Voirin, Conception architecturale des systèmes basée sur les modèles avec la méthode Arcadia, volume 3, ISTE Group, 2018.

[6] W. Lidwell, K. Holden, J. Butler, Universal principles of design, revised and updated: 125 ways to enhance usability, influence perception, increase appeal, make better design decisions, and teach through design, Rockport Pub, 2010.

[7] M. J. Escalona, L. García-Borgoñón, N. Koch, Don't throw your software prototypes away. reuse them!, in: E. Insfran, F. González, S. Abrahão, M. Fernández, C. Barry, H. Linger, M. Lang, C. Schneider (Eds.), Information Systems Development: Crossing Boundaries between Development and Operations (DevOps) in Information Systems (ISD2021 Proceedings), Universitat Politècnica de València, Spain, 2021.
[8] A. Sánchez-Villarín, A. Santos-Montaño, N. Koch, D. L. Casas, Prototypes as starting point in mde: Proof of concept., in: WEBIST, 2020, pp. 365–372.
[9] Iso/iec/ieee international standard - systems and software engineering–vocabulary, ISO/IEC/IEEE 24765:2017(E) (2017) 1–541. doi:10.1109/IEEESTD.2017.8016712.
[10] P. Laborde, S. Costiou, É. Le Pors, A. Plantec, Reuse in component-based prototyping: an industrial experience report from 15 years of reuse, Innovations in Systems and Software Engineering 18 (2022) 155–169.
[11] B. Meyer, Object-oriented software construction, volume 2, Prentice hall Englewood Cliffs, 1997.
[12] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (1969) 576–580.
[13] B. Meyer, Design by contract, Prentice Hall Upper Saddle River, 2002.
[14] A. P. Black, O. Nierstrasz, S. Ducasse, D. Pollet, Pharo by example, Lulu. com, 2010.
[15] P. Laborde, Y. Le Goff, É. Le Pors, A. Plantec, S. Costiou, Live application programming in the defense industry with the molecule component framework, Journal of Computer Languages 80 (2024) 101286.
[16] J. Fabry, S. Ducasse, The Spec UI framework, Square Bracket Associates, [Kehrsatz] Switzerland, 2017. OCLC: 1021885954.
[17] S. J. Montaño, J. P. S. Alcocer, G. Polito, S. Ducasse, P. Tesone, Methodproxies: A safe and fast message-passing control library, in: IWST 2024: International Workshop on Smalltalk Technologies, July 8-11, 2024, Lille, France, 2024.