DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Validating Syntactic and Semantic Errors of a Domain-Specific Language for Defining Clinical Pathways in Integrated Care Platforms

Selin Erdem

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Validating Syntactic and Semantic Errors of a Domain-Specific Language for Defining Clinical Pathways in Integrated Care Platforms

Validierung von syntaktischen und semantischen Fehlern einer domänenspezifischen Sprache zur Definition klinischer Pfade in integrierten Pflegeplattformen

Author:	Selin Erdem
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Tri Huynh, M.Sc.
Submission Date:	07.04.2022

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 07.04.2022 Selin Erdem

Acknowledgments

First and foremost, I would like to thank my advisor Tri Huynh for his support, patience and guidance throughout this thesis.

I also would like to thank Prof. Dr. Florian Matthes for giving me the opportunity to pursue this thesis at his chair.

In addition, I would like to thank all of the participants of the user studies for taking their time and joining the evaluation phase.

Last but not least, I would like to thank my friends and family, who have always been my biggest supporters and especially my parents for their endless love and support, without whom I would never have enjoyed so many opportunities.

Abstract

Text-based modeling of Clinical Pathways is complicated and error-prone due to the complexity and extent of the treatment process. For medical experts, the error rate is significantly more probable because of their unfamiliarity with modeling or programming practices. Therefore, to solve this problem, our study develops a constraint validator to detect syntactic and semantic errors in a Domain-Specific Language specialized in modeling clinical pathways. The output of our validator is non-technical, elaborated error messages to assist medical experts in locating the error source and executing the corresponding solution. To evaluate the usability and accuracy of the validator, we will interview medical experts across various departments to collect quantitative and qualitative feedback. The evaluation focuses on the comprehension of the generated error messages, error correction, and the overall user experience of the medical experts.

Contents

1. Introduction

This chapter describes the motivation, objectives, research questions, and lastly the structure of this thesis.

1.1. Motivation

CONNECARE, "Personalised Connected Care for Complex Chronic Patients", is an H2020 Research and Innovation Project that aims to explore digital tools to provide a smart and adaptive integrated care system for chronic care management for patients with multi-morbid conditions. It supports collaborative work among various stakeholders, such as patients, their families, as well as health and social care workers [1].

In the current CONNECARE system, the clinical pathways are modeled using the semi-structured language, Extensible Markup Language (XML). XML allows for flexible usage of data from existing systems or applications. However, the models defined in XML are usually significantly lengthy and involve a lot of code repetitions. For this reason, a new domain-specific language, Acadela, a modeling language for clinical pathways, is developed as an alternative to XML. It is developed using textX, a meta-language for domain-specific language specification. Acadela aims to increase the usability of CONNECARE with a more readable and user-friendly syntax in comparison to XML, as well as increase the learnability of the system with a far more intuitive grammar that can be used by modelers with varying degrees of technical proficiency as well as novice modelers.

However, despite Acadela's aim to reduce the implementation effort of modelers with its intuitive syntax, the built-in error handler of textX is not user-friendly and comprehensive enough to reduce the learning and debugging effort of modelers. The reason why textX's built-in error handler falls short for Acadela and its primary objective of user-friendliness is that the syntax error messages produced by textX are highly non-intuitive, ambiguous, and use a relatively technical jargon. They also lack in providing context to the modelers as they do not provide the cause of the error and suggestions to fix it. Moreover, it is not possible to detect semantic errors with textX's built-in error handler, which is critical for creating logically correct and coherent models. Hence, this thesis aims to develop a user-friendly and accurate syntax and semantic error validator, which will eventually reduce the learning and debugging effort of Acadela.

1.2. Objectives and Research Questions

The main objective of this thesis is to implement a syntax and semantic error validator for the Acadela language, which produces user-friendly and accurate error messages.

The preliminary part of this research focuses on identifying the syntax and semantic error types that can be validated by the error validator which also involves identifying the semantic rules of the Acadela language. Therefore, the first research question is:

Research Question 1: *What syntactic or semantic errors can be validated by the DSL during the modeling of clinical pathways?*

After the preliminary part, the focus of the work is to implement a user-friendly error validator following the constraints identified in the preliminary part.

Research Question 2: *How to identify and interpret the errors to modelers in an accurate, user-friendly manner?*

Once the technical implementation of the error validator is complete and it is integrated into the existing system, the usability and user-friendliness of the new error validator have to be evaluated by the potential users of the language:

Research Question 3: *What is the opinion of modelers regarding the usability and accuracy of the error validator?*

1.3. Thesis Structure

The content of the upcoming chapters is briefly described as follows:

Chapter 2 describes the relevant concepts and technologies as background knowledge for the readers to better understand the following chapters.

Chapter 3 describes the previous research and works that focus on syntactic and semantic constraint validation of Domain-Specific Languages as well as research on assessing the usability and the readibility of error messages as related work.

Chapter 4 describes the current CONNECARE system as a case study for readers to better understand the system and the integration of Acadela into it.

Chapter 5 describes the identified constraints for the error validator including the identified semantic rules of Acadela and constraints for increasing the user-friendless of the language through error validation.

Chapter 6 explains the system architecture of the error validator, as well as the Acadela compiler and interpreter, and the integration of the error validator into the current architecture.

Chapter 7 provides answers to research question 2 with a thorough description of the implementation of the error validator as well as its integration into the existing Acadela system.

Chapter 8 provides answers to research question 3 by presenting the evaluation approach and the results of the user studies conducted as part of the evaluation of the implemented error validator and the Acadela language.

Chapter 9 describes and discusses the limitations of the implemented system and its validity.

Chapter 10 concludes the thesis by summarizing the findings and providing suggestions for future work.

2. Background Knowledge

This chapter describes several important concepts and technologies which are valuable for the readers to understand the following chapters.

2.1. Spell-checker

A spelling checker or a spell-checker is a tool or application that aims to detect word errors. Spell-checkers are widely used for detecting word errors and handling these errors. They identify the misspelled or incorrect words in a text and either suggest the best possible combination of correct words to the user or replace them if it also offers auto-correction [2].

Levenshtein distance algorithm, coined by Vladimir Levenshtein, is an algorithm that is widely used in many applications of computer science, especially in spell-checker implementations as a metric to measure text similarity. The Levenshtein Distance is defined as the minimum number of character edits that needs to be performed to transform a given string into another. An edit refers to one of the following edit operations: insertion, deletion, and substitution [2]. For instance, the Levenshtein distance between the words "kelo" and "hello" is 2 as the letter "k" needs to be substituted with "h" and an additional "l" needs to be inserted.

2.2. Clinical Pathway Modeling

The term clinical pathway (CP) can be defined as a method for providing patient-care management of a specific group of patients in a well-defined period of time. The development and utilization of CPs is a multidisciplinary process that involves several stakeholders such as clinicians, nurses, physiotherapists, and case managers. CPs effectively and efficiently standardize treatment progressions for many conditions and they mitigate the problems that decision-makers need to address [3, 4]. What the term CP encompasses can significantly differ from one CP to another as there is a variety of ways of formulating, approaching, and modeling CPs. However, a criterion is developed for what constitutes a CP which involves the criteria that *"the intervention was a structured multidisciplinary plan of care"* that must be satisfied. Other criteria involve but are not limited to: *"the intervention detailed the steps in a course of treatment or care in a plan"*, *"the intervention had time-frames or criteria-based progression"* and *"the intervention aimed to standardize care for a specific clinical problem of healthcare in a specific population"* [4]. As traditional text-based approaches are not effective in terms of maintenance and analysis

of CPs, some generic modeling languages such as UML, EPC, and BPMN and in some cases domain-specific process modeling languages are now being used for modeling CPs [3].

2.3. Domain Specific Language

A Domain-specific language (DSL) is a language that is built specifically for a specific application domain, in contrast to general-purpose programming languages (GPLs) which are applicable across many application domains [5]. DSLs in general, offer considerably more expressiveness and usability in their domain of application compared to GPLs. They usually have a simpler notation and constructs than GPLs as they are tailored exclusively toward their specific application domains [6]. DSLs also reduce the amount of domain and programming expertise needed and consequently they can be used by people with less or no programming experience but accustomed to the application domain. However, developing DSLs is considered a challenging and lengthy process as it requires both extensive knowledges of the domain as well as language development expertise. Some of the DSLs that are widely known and commonly used are HTML, Latex, SQL, and MATLAB [5].

2.4. Compiler-Compiler

A compiler-compiler is also known as a compiler generator, is a tool that creates a parser, interpreter, and/or a compiler from a formal definition of a language. There are several types and variants of compiler-compilers. The first type is a parser generator which only handles the parsing or in other words syntactic analysis of a given language. Lex/Yacc and ANTLR are the most commonly known and used parser generator tools. Typically, a parser generator takes a grammar that defines the language's syntax and outputs the source code of the parser of the language. The parser then checks the syntax of the source code, and creates an abstract syntax tree (AST) or a parse tree from the input source code. However, parser generators do not handle the semantic analysis of the language [7].

A meta compiler, on the other hand, is defined as a tool that takes again the syntax description of a language, plus the description of its semantics. The language in which such language descriptions are defined is called a metalanguage [8]. Hence, meta-compilers usually automate the type checking and the semantic implementation and analysis of a language in addition to generating parsers [9].

2.5. Metamodel

To understand what a metamodel is, first the concept of a model should be defined. A model has many definitions and its definition widely varies according to the context in which it is being used. However, it can be defined as *"a system that helps to define and to give answers to the system under study without the need to consider it directly"*. In other words, a model is an abstraction of a system, which represents a system in a more partial or simplified way. Even though metamodel also has many definitions, it can simply be described as *"a model of models"* or *"a model that defines the language for expressing a model"* [10].

Modeling languages can be understood with respect to models and metamodels as *"a modeling language is defined by a metamodel and is a set of all possible models that are conformant with its respective metamodel."* [10]. Metamodel, in this context, refers to the abstract syntax of a given meta-language which is usually defined in a form of grammar. The relationship between a model, metamodel and modeling language relative to a system can be further understood through the Figure 2.1

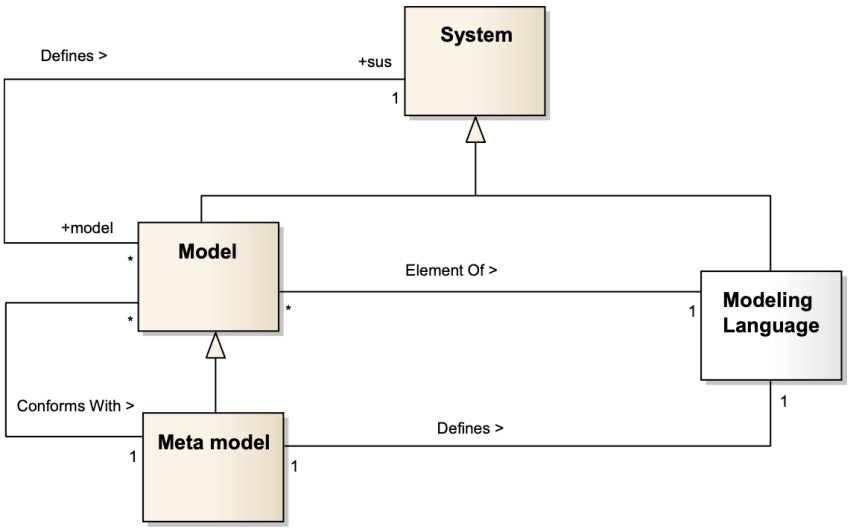

Figure 2.1.: The relationship between the concepts: System, model, modelling language and metamodel [10]

2.6. TextX

textX is a free and open-source meta-language and a tool for the fast development of DSLs in the programming language Python. The Acadela language is built using textX. textX provides many powerful features for language developers such as building a meta-model, in other words, abstract syntax, and constructing a parser of the language automatically from the grammar file of the DSL in run-time. The meta-model built by

textX contains a set of Python classes inferred from the language grammar. Furthermore, the parser parses programs written in the new language and constructs a model that is a Python object graph, which is in a sense, an Abstract Syntax Tree (AST). The basic workflow and architecture of textX are visualized in Figure 2.2. As the first step (step a in Figure 2.2), textX parses the grammar of the DSL and produces a meta-model and a parser from it. This parser from the first step is then used for parsing programs and models created using the DSL. As the second step (step b in Figure 2.2), the parser parses programs or models and creates a graph of objects where each object is an instance of a corresponding class of the meta-model. The generated in-memory model can later be used for procedures such as interpretation or source code generation(step c in Figure 2.2) [11].

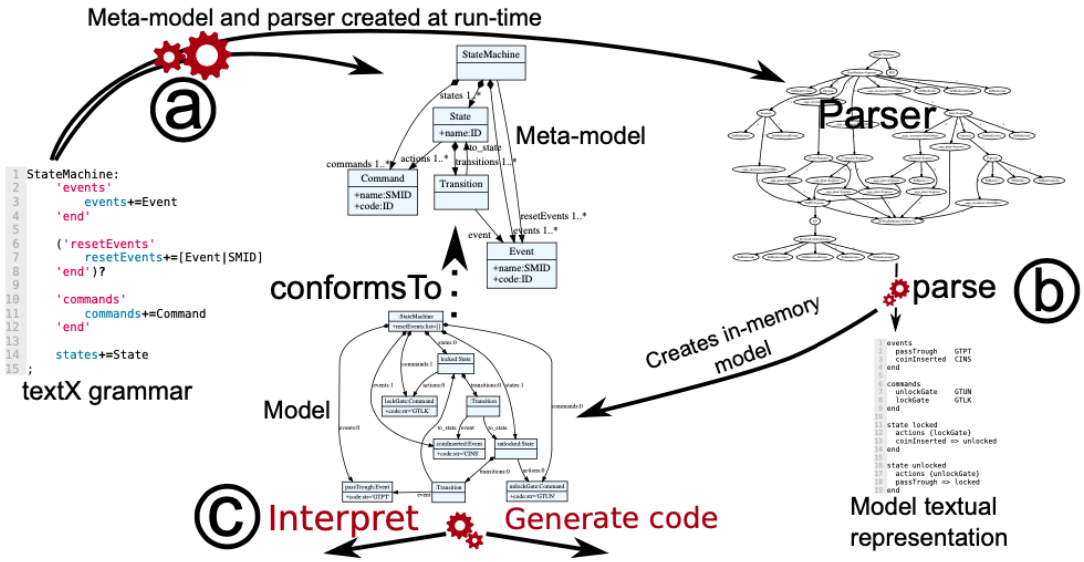

Figure 2.2.: textX workflow and architecture [11]

3. Related Work

This chapter describes some of the previous research and works that focus on syntactic and semantic analysis and error handling of Domain-specific languages as well as research about constructing user-friendly and readable error messages.

Thomas Baar (2016)

In his work, *"Verification Support for a State-Transition-DSL Defined with Xtext"*, Thomas Baar tackles the lack of semantic verification support for DSLs defined with Xtext[1], which is an Eclipse-framework for textual DSL development with full tooling support [12]. The metalanguage textX[2] which is used for defining Acadela is inspired by Xtext so that their implementations and usage are very similar to each other [11]. In his paper, Baar claims that even though Xtext provides adequate support for checking a given DSL's syntax rules, support for specifying the semantics of the DSL and semantic verification is neglected in Xtext similar to textX. Therefore in his work, he proposes an approach for adding semantic analysis support for textual DSLs created with Xtext, and his approach is illustrated on a simple State-Transition-DSL called SMINV, implemented in Xtext. The editor of SMINV in the end verifies the semantics of models on the fly. For the implementation of the semantic validator, the semantic model properties of the DSL are formulated as proof obligations which are first-order formulas. These proof obligations are then implemented by using the standard validator classes of Xtext-framework which are used for syntax validations as well. Finally, the implemented proof obligations are passed to an automatic theorem prover called PRINCESS and validated. In conclusion, his implementation provides a systematic semantic validator for Xtext in which the semantic checks are realized almost analogously to syntax checks [12].

The semantic validator proposed in this work, SMINV, provides on-fly semantic verification which can be useful for DSLs with more formulated and arithmetical semantic rules. However, for Acadela, a solution similar to this would fall short as most of the semantic constraints cannot be formulated as arithmetical and logical proof obligations.

[1] https://www.eclipse.org/Xtext
[2] https://textx.github.io/textX/3.0/model/

Dejanovic et. al (2015)

Dejanovic et. al in their research, describe a DSL called PyFlies for designing experiments in the field of psychology. PyFlies is also a textX based textual language like Acadela. The syntactical validation of PyFlies is solely left to the parser generated by textX based on the language grammar. The parser produces an in-memory graph representation, in other words, a model, which is later used for checking the semantic conformance of models. For defining semantic constraints, the paper proposes a pragmatic approach where the models developed in PyFlies are mapped to the target platform with already defined semantics using source code generators. Similar to Acadela, Dejanovic et. al, list syntax and semantic checks as a limitation of their language. Based on the results of the case study that they have conducted, they suggest that the error messages should include an explanation of what the user is doing wrong and suggestions to solve the errors [13].

Even though this work does not provide further information on syntactic and especially semantic analysis of textX based DSLs compared to the official documentation of textX, it shows the shortcomings of textX in terms of validation and discusses ways to improve them. It is also a relevant and noteworthy work as the language PyFlies is very similar to Acadela in terms of its objectives and implementation.

Simon Graband (2020)

In his Master's Thesis with the title *"Model Validation in Graphical Cloud-Based Editors"*, Simon Graband, covers a similar scope to this thesis, with a focus on graphical modeling languages. It covers the existing solutions for constraint validation in graphical modeling languages, and how they can be applied to cloud-based tools. This research is also relevant to our research as the implemented constraint validator is integrated into Case Management Model and Notation(CMMN) editor, a graphical modeling language commonly used in the CONNECARE system for modeling case studies as well as other health domains. The thesis includes a literature review of model validation frameworks available for graphical modeling languages and proposes a solution to be used for model validation in graphical cloud-based editors. The thesis also evaluates the usability of error messages generated by the proposed solution [14].

This work is useful for our research as it provides some insight into the error validation of modeling languages and tools, even though the modeling languages that it covers are graphical modeling languages in contrast to Acadela which is a text-based modeling language. The evaluation part of this thesis is also applicable to our work as it offers some idea about error messages of modeling languages and their perceived usability by end-users with different technical backgrounds.

Denny et. al (2021)

As mentioned earlier, the scope of this thesis is not only to define and validate the syntax and semantic rules of Acadela but also to provide user-friendly and accurate error messages to reduce the learning and debugging effort of the language. Therefore, is it crucial to consider the literature on factors affecting the readability and understandability of error messages used in programming languages.

Denny et. al in their research, studied the readability and its constituent factors to design error messages for novice programmers. The motivation of their research is to investigate the factors that influence error message readability, especially for novice programmers through three related experiments and as a result to define some requirements for error messages that can improve the learning experience of novice programmers. According to the results of these experiments, the 10 error messages rated as most readable and 10 least readable appeared to be alike and shared certain qualities. For instance, two qualities which most likely to play a role in the perceived readability are the length of the messages and the density of technical jargon and acronyms. Regardless of their technical competence, the participants found shorter messages more readable. It is important to note that, according to this work, even though adding more explanation to error messages to further aid the novice users sometimes works, such error messages are generally found to not contribute to the understanding of the users. So the research claims that the addition of words does not consistently improve novice comprehension. In conclusion, the research identified removing technical jargon, using complete sentences and simpler vocabulary, and providing shorter messages are effective ways to improve the readability of error messages [15].

The findings of Denny et. al, can provide a good baseline for how the default syntax error messages of textX can be customized and how the new semantic error messages can be structured for better user-friendliness and readability.

4. Case Study

This chapter further describes the CONNECARE project and its architecture as a case study and gives an overview of the current system. It also explains how Acadela System is going to be integrated into the existing CONNECARE architecture.

4.1. CONNECARE Project Overview

In Europe, 35-40 percent of the elderly population which refers to the people who are over the age of 65, is reported to have chronic or longstanding health problems and 25% of those receive long-term medical treatment. For instance in Catalonia, according to recent data, 30% of the general population has one or more chronic disorders. This increasingly large group of the general population needs to visit emergency rooms frequently and suffer from sudden hospital admissions.

A Complex Chronic Patient (CCP) is a person who has at least one or more chronic diseases or comorbidities. CCPs are mostly elderly people and because of their conditions, they consume a very high level of healthcare resources. For this reason, the European research project CONNECARE aims to *"[...] co-design, develop, deploy, and evaluate a novel integrated care services model supported by a smart and adaptive case management system for better care coordination and self-management of CCPs."* CONNECARE enables collaboration and communication among stakeholders such as healthcare professionals, patients, and their caregivers through integrated technological solutions [16]. A conceptual representation of the potential stakeholders of integrated adapted case management and their communication and synchronization pattern can be observed in Figure 4.1. In short, the CONNECARE project's goal is to integrate clinical treatment, primary care, and home hospitalizations of CCPs to save costs, spare health resources, and increase their life quality.

Figure 4.1.: High-level project vision of the Smart Adaptive Case Management with stakeholders [17]

4.2. Clinical Pathway Elements

The models of CPs in CONNECARE consist of many elements. The conceptual meta-model that shows these elements and their relationships can be seen in Figure 4.2. As most of these elements are also modeled in Acadela, it is important to explain some of the essential elements to better understand this thesis.

Case: `Case` is the parent element or the root data structure of the model which refers to the CP that is being modeled.

Stage: `Stage` elements denote the steps in the treatment. Each `Stage` element consists of one or more `Task` elements that need to be executed.

Task: `Task` elements represent the work that needs to be conducted in a given `Stage` of treatment. It can be a `HumanTask` which is completed by one person such as a clinician, an `AutomatedTask` which is typically executed by a third-party system, or a `DualTask` which refers to tasks that are tasks which a `HumanTask` is followed by an `AutomatedTask`.

InputField: A field element to collect necessary medical inputs or documents.

OutputField: A field element that describes the patient's medical state based on the values collected via `InputFields`.

Hooks: `Hook` elements orchestrate internal and external communication on state change events.

Summary Section: Summary Section elements which are denoted with the keyword `Section` under `SummaryPanel` element, define the relevant medical information to be displayed on a Case Summary page in SACM.

Workspace: A `Workspace` element is a container for model definitions and their instances. A `Workspace` can refer to a clinic where CPs are modeled.

Entity: Entities are the `Case` `Task` and `Stage` object definitions.

Attribute: Attributes are the properties of the entities defined in a CP.

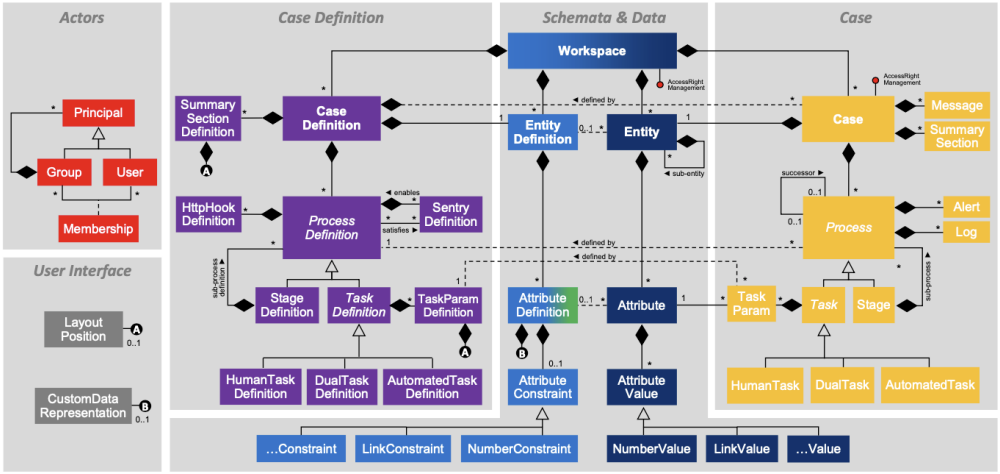

Figure 4.2.: Conceptual meta-model of clinical pathways [17]

4.3. CONNECARE System Architecture

The CONNECARE system consists of three major subsystems, namely the Smart Adaptive Case Management System (SACM), the Self Management System (SMS), and the User Identity Management (UIM) as can be seen in Figure 4.3. The SACM is the central and most integral component of the CONNECARE system that is responsible for case management and holds the business logic. The UIM is responsible for user management, roles, and active sessions in the system. It in a way ensures the security of the system and personal health data. The SMS ensures the patients' access to the CONNECARE system. It is a mobile application that provides patient notification and communication functionalities as well as the integration of medical devices [18]. As UIM and SMS are not relevant to the scope of this thesis, this chapter will only further elaborate on SACM.

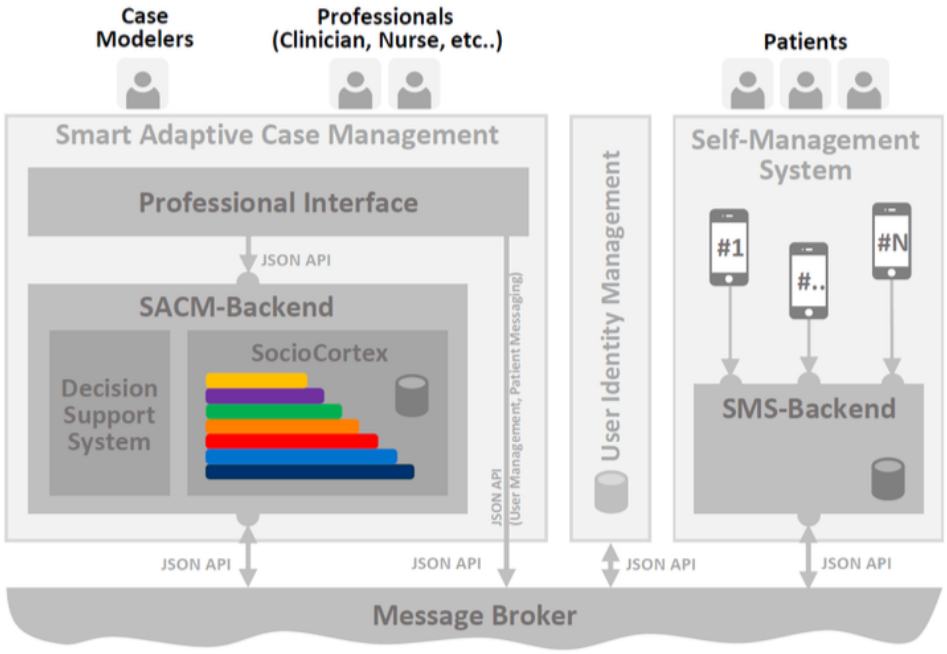

Figure 4.3.: Architecture of the CONNECARE System [17]

4.3.1. Smart Adaptive Case Management System (SACM)

SACM is the most important and central component of the CONNECARE system. It provides the system's adaptive case management functionality and it is the primary persistent storage of the whole system. It also provides a user interface for modelers and medical professionals [18].

SACM system consists of two subsystems, the SACM-Backend and the Professional Interface which is the frontend of the SACM-Backend. These two components of SACM communicate via a RESTful and JSON-based API [18].

SACM is built upon SocioCortex, an information modeling platform developed by the Technical University of Munich. It is essentially a generic full-stack modeling engine. To be used in the healthcare domain, specifically for CONNECARE, it is adapted to meet the specific requirement and extended to support case management. SocioCortex supports the basic case modeling in SACM [17]. It is adapted from a data modeling system into a fully integrated process modeling platform to be used by the SACM [19]. SocioCortex supports the model elements described in section 4.2 such as but not limited to, workspace, group, entity and attribute, case, stage, and task definitions. These elements are further described in detail by Michel in his work [17]. SocioCortex is responsible for persisting these elements of the CP models.
The SACM backend is very flexible in terms of case models that it can support as it is built upon SocioCortex which is a generic case engine. The whole SACM system and its functionalities are configured through case models so modeling CPs is very critical for

providing adequate and correct care. SACM currently supports an XML-based DSL to model cases [18].

The Professional User Interface is a Single Page Application (SPA) which is used by clinicians and modelers to access, interact and manage the CPs that are stored in the SACM system. It includes views for core user interface features of SACM such as a dashboard for an overview of possible actions, pending tasks, the cases of the user, a case summary view, and a case workflow view [19]. An example view of the Professional User Interface can be seen in Figure 4.4 which is the Summary view of a patient. It extracts the stages and tasks, and the input and output fields of a selected case. The output fields contain the visualization logic as an attribute based on the data from the input fields. The visualization logic can be defined as a conditional expression or a color function that determines the color used in the visualization conditionally based on the input field values. The modeler can also include an HTML expression for the visualization of the patient's medical status.

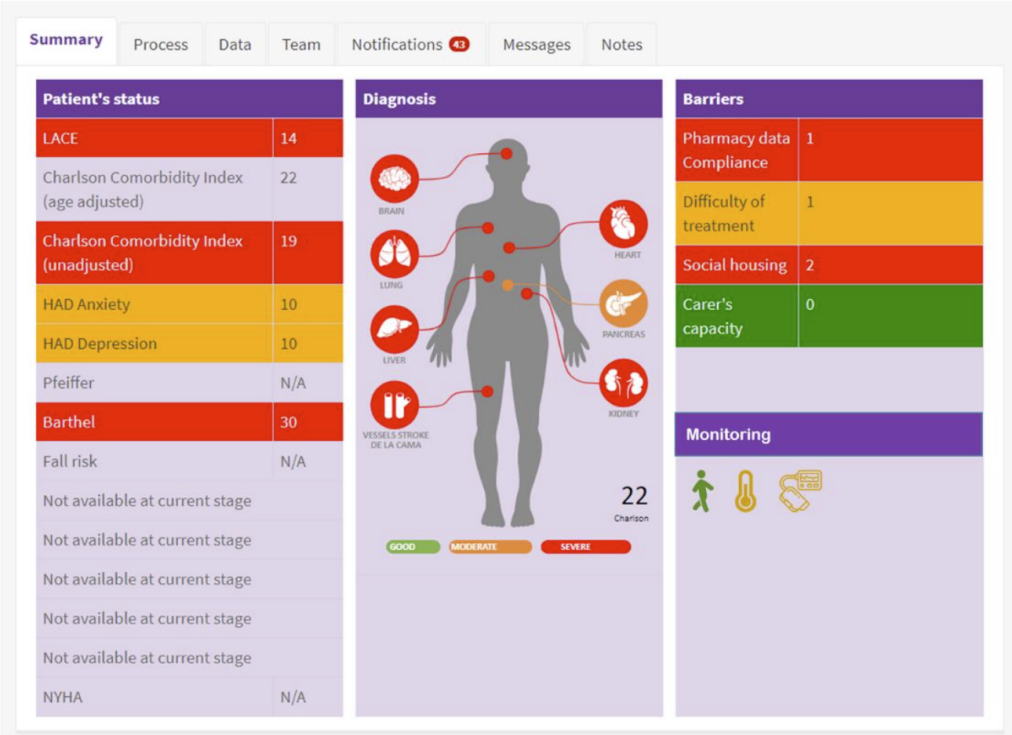

Figure 4.4.: A screenshot of Professional Interface that displays the summary panel page of a patient [20]

4.4. Acadela

Adaptive CAse DEfinition LAnguage (Acadela) is a textX based DSL that is designed to be used for modeling CPs. The goal of Acadela is to simplify the modeling of clinical pathways in CONNECARE while preserving the flexibility and adaptability of SACM. The elements which are modeled as part of CPs in CONNECARE are also definable in the Acadela language. Acadela reduces the complexity of modeling and eases the modeling process with three features: simpler grammar, shorter models, and Integrated Development Environment (IDE) support [20]. A comparison of the current XML-based modeling and Acadela based modeling can be seen in Figure 4.5. As it can be observed from the example code segments in this figure, the Acadela code (4.5b) is significantly shorter, more readable, especially for novice users, and observably has a much simpler syntax compared to the XML-based model (4.5a). Acadela also comes with IDE support, which offers auto-complete and code highlighting features.

```xml
<StageDefinition
  id="MCS2_CaseIdentification"
  description="Case Identification"
  isMandatory="true"
  repeatable="ONCE"
  entityDefinitionId="MCS2_Identification"
  entityAttachPath="MCS2_Identification">

  <!-- place <HumanTaskDefinition> here--->
  <!-- place <AutomatedTaskDefinition> here--->
  <!-- place <DualTaskDefinition> here--->

</StageDefinition>
```

```xml
<EntityDefinition id="MCS2_Identification" description="Identification">
  <AttributeDefinition id="MCS2_Charlson" description="Charlson Comorbidity Index"
    EntityDefinition.MCS2_Charlson" multiplicity="any" />
  <AttributeDefinition id="MCS2_Barthel" description="Barthel" type="Link.EntityDef
    multiplicity="any" />
  <!-- Place additional <AttributeDefinition> to extend evaluation schema here -->
</EntityDefinition>
```

(a) Current XML based modeling for a Stage

```
Stage Identification
    #mandatory
    #noRepeat
    owner = 'Setting.CaseOwner'
    label = 'Identification'
```

(b) Acadela based modeling of the same Stage

Figure 4.5.: The modelling of a Stage element in (a)XML and (b)Acadela

4.5. Acadela Integration

Acadela interpreter functions as a transpiler, a source-to-source compiler that translates a source code in a language to another language. It translates the source language which is the Acadela DSL itself into JSON. After the Acadela source code is parsed, an AST-like Python object graph is constructed by the Acadela compiler. Acadela interpreter processes this graph and exports the elements of the model in a JSON format that is readable by SACM. The SACM can process the CPs in JSON format and send a request to SocioCortex to create the elements of the CP and store them. Acadela can easily

replace XML as the modeling language as they both get transpiled into JSON, which is recognizable by the SACM.

After the integration of Acadela into the existing system, the workflow of CP modeling is designed to be as follows:

First, the modeler develops and submits their models through Acadela IDE and Acadela IDE sends a request with the Acadela code submitted by the modeler to an endpoint in Acadela Backend.

Then, the Acadela Compiler located within the Acadela Backend compiles the source code of the model and performs the syntactic and semantic validation.

If the source code is found to be syntactically and semantically valid, the Acadela interpreter exports the elements of the model in JSON format and sends it to the SACM. The case is then created in SocioCortex as described before and a response with a success message is returned to Acadela Backend.

Thereafter, Acadela Backend forwards this response to Acadela IDE. Acadela IDE is designed to display this message to inform the modeler that their model is successfully created and stored in the system.

If the source code is found to be syntactically or semantically invalid, Acadela Backend returns a response to the Acadela IDE with an error message. The error message is displayed to the modeler to help them fix the error.

5. Constraint Identification

This chapter presents the identified constraints and requirements for the error validator and explains the error types that need to be validated. It provides answers to the first research question: *What syntactic or semantic errors can be validated by the DSL during the modeling of clinical pathways?*

5.1. Usability Contraints for Error Messages

As mentioned earlier, the main objective of this work is to design and implement a user-friendly and accurate error handler that will reduce the learning and debugging effort of Acadela. Thus, several constraints and requirements are defined to achieve this objective in the preliminary stage of this work. These constraints and requirements include some specific to each error type and at the same time, more general ones concerning both error types. The constraints and requirements specific to each error type are discussed in section 5.3 and section 5.2 of this chapter.

The quality of a given programming language's error messages is exceedingly critical for the usability of the language, particularly for novice users. Error messages can often be perceived as overwhelming by the users, especially if they are not accustomed to the language yet or do not have familiarity with technical concepts, acronyms, and jargon. Therefore it is decided that the language of the error messages should be as non-technical as possible. For the same reason, the error messages should also be straightforward, precise, and concise.

The readability and understandability of the error messages mostly depend on the delivery of the error messages. However, the content of error messages is also crucial as a readable delivery alone cannot decrease the debugging effort of the target users alone without actually providing valuable information within the error messages. Therefore it is decided that the syntax and semantic error messages should include:

1. Location of the error
2. Cause of the error
3. Suggestions to solve the error if possible

The exact location of the error, in other words, the line number and column number of the position that the error is located should be specified in the error messages as this information is the most critical information for a user to be able to locate and fix an error.

Secondly, stating the cause of the error in simple and understanding terminology is crucial as the target users need to understand the cause of the error in order to successfully and rather easily solve them. Therefore, it is decided to include the cause of the error in error messages as accurately as possible in an understandable language.

Lastly, it is decided to include suggestions to solve errors in the error messages to mainly reduce the debugging time and effort.

The requirements and constraints described so far are defined for both semantic and syntax error validators of the language. Therefore, these constraints are taken into consideration for the implementation of the handlers of both error types. The rest of this chapter focuses on the constraints and requirements specific to each validation type.

5.2. Constraints for Syntax Validation

The syntactic constraints of Acadela are already defined by the textX grammar with a set of textX rules. The rules that define the language syntax must be satisfied by the Acadela models during compilation and if one or more syntax rules are violated, a syntax error should be produced. textX checks these rules with the parser that it generates and provides a built-in syntax error handler that produces and throws these syntax errors. However, the default syntax error messages of textX are non-intuitive, technical-oriented, and lack context. As the CP modelers can have varying degrees of technical proficiency, making these error messages as understandable and concise as possible is critical. Therefore, it is decided that the new syntax error handler will use the default error messages of textX as a baseline for the new error messages and extend the error message by the constraints and requirements defined in this chapter.

5.2.1. Unexpected Elements

In this work, unexpected element errors are defined as a subset of syntax errors in which the statement or more specifically an element is defined at a place where the compiler was not "expecting" it to be defined. In other words, the syntax error type "unexpected elements" refers to the attributes and elements that are not placed under the correct parent element or the commands that include elements that are "foreign" to the Acadela grammar and consequently are not recognized by the parser.

The default syntax error handler of textX successfully catches these errors and produces error messages in the same format for every possible syntax error that falls under this type. The format of these error messages can be roughly understood from the following error message:

```
None:171:9: error: Expected 'owner' or 'client' or 'dynamicDescriptionRef'
or 'externalId' or 'additionalDescription' or PreconditionTerm or
HumanTaskTerm or Ref or AutoTaskTerm or DualTaskTerm at position
/Users/macbookpro/dsl-connecare/acadela/model_placeholder:
(171, 9) => '          *group = 'S'.
```

As it can be seen from the example above, the default error messages of textX list the potential keywords or elements expected in place of the unexpected element. This list of potential keywords provides a good basis for the error messages however the error messages still lack context and do not include any valuable explanation for the cause of the error. Therefore these error messages should be processed and extended to assist the users to solve the error.

To further assist the user via the new error messages, this error type can be divided into two categories where:

1. **Misplaced element or attribute:** the keyword is in the language definition however it is "misplaced" according to the grammar
2. **Unrecognized (foreign) element:** The keyword that caused the error is unrecognized by the language. It can also be a potential typographical error which is explained in the next section in detail.

To enhance the error messages which fall into the "unexpected elements" subcategory of syntax errors, it is decided to check if the unexpected element is a misplaced element or an unrecognized element and display an error message accordingly to further clarify the cause of the error.

The location of the error is already provided by textX. As for suggestions to solve the error, the list of expected elements will be included in the new error messages in a relatively clearer and more readable way. Furthermore, the cause of the error will be further analyzed by integrating a spell-checker and the readability of the error message will be enhanced by explaining or replacing the technical keywords and acronyms included in the error messages as described in the next sections.

5.2.2. Typos

Given Acadela is a DSL, it includes a lot of reserved keywords in its grammar that refer to the elements of the clinical-pathway domain, for instance, `Stage`, `HumanTask`, `Task`, `InputField`, or `Form`. As misspelling keywords is a commonly made syntax error by programmers regardless of the language being used, it is decided to use a spell-checker to look for and catch potential typographical errors (typos) and suggest keywords that can replace the "unexpected element" to further assist the user with solution suggestions. For instance, in case a user misspells the keyword `owner`, and types `owneer` instead, then the error handler should detect this potential typo and include a statement in the error

message such as *"No keyword ownneer. Did you mean owner?"*. Such a statement will warn the user about a potential typo and will improve the debugging experience.

For the sake of implementing this feature, it is required to find a spell-checker library available for Python, that suits this purpose. It should build a dynamic dictionary from keywords within Acadela and look for potentially misspelled statements around the location of the error and suggest keyword replacements with one or two edit distances if there are candidates.

5.2.3. Explaining Technical Terminology

The error messages of textX which are used as a baseline for the new error messages include technical terms and acronyms such as `INT`, `STRING`, `EQ`, `FLOAT` which are mostly base types of textX. An error message that includes such a term can be seen below:

```
TextXSyntaxError: None:175:25: error: Expected STRING at position
/xxx/xxx/dsl-connecare/acadela/model_placeholder:(175, 25)
=> 'ndition = *Evaluation'.
```

It is thought that novice modelers or modelers with no programming experience might not be familiar with this technical jargon. Hence, to make the language of error messages less technical, these terms need to be replaced with relatively more understandable alternatives. For instance, the term `STRING` should be replaced with `Text with quotation marks` or the term `EQ` should be replaced with `equal sign (=)`.

The planned implementation of this constraint is somewhat straightforward and trivial as it involves identifying the terms that need to be explained or replaced and replacing the occurrences of each of these terms with a less technical equivalent.

5.2.4. Validating String Patterns

Within the Acadela language, values of several attributes follow certain string patterns. However, in the original Acadela grammar, the values for these attributes have the type `STRING` and the syntax rules of these string patterns are not yet defined in the grammar. Therefore the values of these attributes are not yet being syntactically validated during compilation even though they need to be semantically and syntactically valid as they are either conditional expressions or functions which are used for visualizing elements in the SACM UI. After considering several options including the use of regular expressions, it is decided to validate these string patterns using textX, by creating separate grammars for each string pattern. The reasoning behind this decision is discussed in the chapter 9 as a limitation.

5.3. Constraints for Semantic Validation

Unlike syntactic constraints of Acadela which are already defined within the Acadela grammar and validated via textX during compile-time, the semantic constraints of Acadela should be defined separately from scratch and validated after compilation using the AST generated by textX during parsing. Therefore during the preliminary stage of this thesis, semantic constraints or rules of Acadela that are needed to be validated for ensuring meaningful and logically coherent CPs are identified. These constraints include constraints specific to the Acadela language, which provide meaning to its syntax structure as well as constraints of the CONNECARE System.

It is important to note that, type checking, which is an integral part of semantic analysis of languages is already implemented so it is not included in the constraints explained in this section, even though it is, in fact, a semantic constraint that requires validation.

5.3.1. References and Paths

In Acadela, elements such as `Tasks`, `Stages`, `InputFiels` and `OutputFields` can be referenced from other elements using a dot notation, where each element is separated by a dot and an element on the right-hand side of a dot is a child of the element on the left-hand side.

The access pattern of such paths should be correct in terms of their ordering. For instance, to access a `InputField` or `OutputField` of a `Form` from another `Stage` or `Task` element, the reference path should follow the following access pattern:
`<Stage>.<Task>.<Field>`

Secondly, the elements within a reference should be already defined in the model and should be referenced via the correct parent element. Therefore the checks involved in this rule are similar to uninitialized or undefined variable errors from GPLs.

5.3.2. Unique Element Identifiers

As a constraint of the CONNECARE System, some elements of CPs modeled in the system have to have unique identifiers. In the current system, this constraint is satisfied with unique XML identifiers within the case templates and the automatically generated database identifiers which are unique across all case templates [17]. This constraint also needs to be enforced in Acadela models. Hence uniqueness of the identifiers of elements is included as a semantic rule of Acadela that needs to be validated. The unique element identifiers constraint has to be checked by the validator and enforced via throwing semantic errors during the semantic analysis. For this purpose, within the semantic validator, a module for checking this constraint needs to be implemented.

5.3.3. Trusted URLs

In the Connecare system, the concept of hooks allows declaring notifications on stage change events of `Stage` and `Task` elements of CPs. These hooks enable process orchestration across system boundaries. These hooks are also defined in the Acadela language. To ensure the correct execution of the hooks and data security, the URLs triggered via HTTP methods defined within the `Hook` elements should be validated. A `Hook` element is considered semantically valid in Acadela if and only if the following conditions are satisfied:

1. the URL of the `Hook` is among the list of trusted URLs of the workspace that the CP belongs
2. the HTTP method declared in the `Hook` is among the allowed methods of this URL.

Thus, a module for semantic validation of `Hook` elements by validating these two conditions should be implemented as part of the semantic error validator.

6. System Architecture

This chapter gives an overview of the architecture of the existing Acadela System and describes the architecture of the implemented error handler.

6.1. System Architecture Overview

In this section, the architecture and the functionality of the existing Acadela System, which is responsible for the compilation and interpretation of the CPs modeled with the Acadela language will be described as the implemented error validator is integrated into this system. The term Acadela source code is used interchangeably with "the CPs modeled with the Acadela language" for simplicity.

The Acadela System consists of three main components namely the error handler, the interpreter, and the compiler script. The compiler script can be considered as the entry point of the system as it is utilized for accessing and using the compiler through outside systems. It is also responsible for compiling and parsing the Acadela source code that it receives as an argument. If the compiler detects a syntax error, the error handler component processes this error and produces and returns a user-friendly error message. The interpreter component is responsible for interpreting the source code, processing the AST generated by the compiler, and semantically validating the source code using the error handler component. It also functions as a transpiler as it transforms Acadela source code into JSON format. The architecture of the Acadela System can be seen in the Figure 6.1.

6.2. Acadela Compiler

This section describes how the Acadela Compiler component works. As mentioned before, the compiler can be accessed and executed through a Python script. This compiler script expects two arguments, one that corresponds to the input Acadela source code, and the mode value which determines whether to run network operations or not.

As a first step, the meta-model and the parser of Acadela are constructed from the Acadela grammar file defined in textX metalanguage using the `metamodel_from_file` function of the textX library. After the parser and the meta-model is created, the source code passed as an argument is parsed and a model of the source code is created using the generated meta-model.
At this point, there are two possible outcomes depending on the result of the syntactical analysis of the source code:

- If there is a syntax error in the source code, in other words, the source code does not conform to the Acadela grammar, textX will automatically throw an error as a Python exception. This exception will be processed by the syntax error handler module of the error handler component and an error message will be returned. This will be further described in the Acadela Error Handler section of this chapter.
- If the source code is syntactically valid, then the interpreter will be initialized and the model will be interpreted and semantically analyzed. The interpretation and semantic analysis of the models will be discussed in the next section.

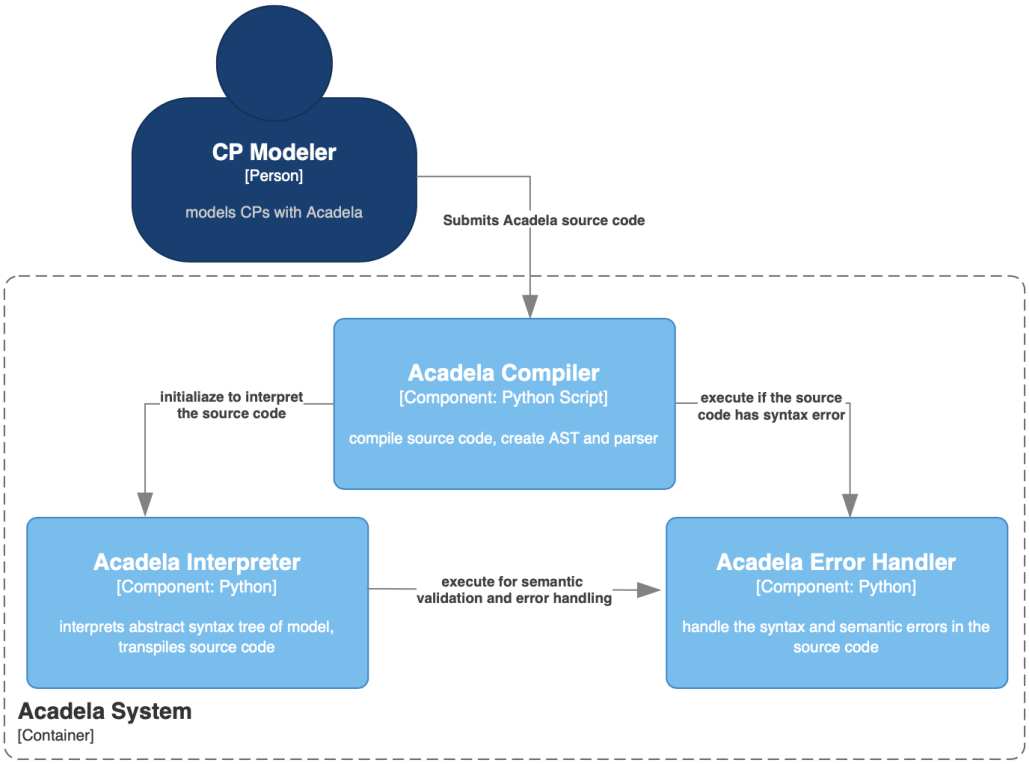

Figure 6.1.: A container diagram of Acadela System that shows the system architecture

6.3. Acadela Interpreter

This section will simultaneously describe the architecture of the interpreter component as well as its functionality.

The Acadela interpreter component consists of a parent interpreter class `CaseInterpreter` and child interpreter classes for each element type that is being modeled in Acadela. Furthermore, to process and structure the attributes and child elements of these elements, each element has a corresponding object class defined in the Acadela Interpreter component. The child interpreters interpret each corresponding element stored in the

AST-like graph object created by the parser and create an object for each one of them using the element class definitions.

To interpret an Acadela source code, first, a `CaseInterpreter` class object should be initialized with the meta-model, model objects created by the compiler(the AST), and the input source code as its attributes. Then, the `interpret` method of this object should be called with the variable `runNetworkOp` as a parameter. The parameter `runNetworkOp` takes a boolean value derived from the `mode` argument of the compiler script. This value declares if the network operations such as sending the model to the SACM should be executed or not.

The `interpret` method of `CaseInterpreter` objects, processes the model object which is an AST-like graph of Python objects where each object is an instance of a class from the meta-model. These classes are created on the fly by textX from the grammar rules [11]. More specifically, textX automatically generates a Python class for each element type modeled in Acadela using the grammar rules.

As mentioned before, the `CaseInterpreter` class objects also has child interpreters defined and initialized for each element type as a class attribute. The child elements of the `Case` element are hierarchically interpreted (top-down graph order) by these individual child interpreters.

In the end, the objects of the graph are interpreted into a Python dictionary of objects, `CaseObjectTree`, to make the upcoming semantic validation process and transpiling easier. The constructed `CaseObjectTree` is used to store the elements of the case definition, their corresponding child elements, and attributes with the following keys: `workspace`, `groups`, `users`, `entities`, `stages`, `tasks`, `case`, `attributes` and `settings`. The values of each of these keys are lists of objects of the corresponding model elements.

Thereafter, the Acadela interpreter calls the Semantic Error Handler module for the semantic validation of the model. After the source code is semantically analyzed using the `CaseObjectTree` and the source code is found to be semantically valid by the validator, the source code is transpiled into JSON format with the `compile_for_connecare` method of the `CaseInterpreter` class. If the value of the parameter `runNetworkOp` is true, then the CP which is transpiled into JSON format is sent to an endpoint of the SACM within an HTTP request.

6.4. Acadela Error Handler

This section describes the implemented system in the scope of this thesis which is the error validator. The error validator will be referred to as the error handler throughout this section to be consistent with the names of the components in the implemented system.

The Acadela Error Handler component consists of two sub-components namely, the Syntax Error Handler and the Semantic Error Handler. The architecture of the Acadela Error Handler on the component level and its interactions within the Acadela System can be seen in the Figure 6.2.

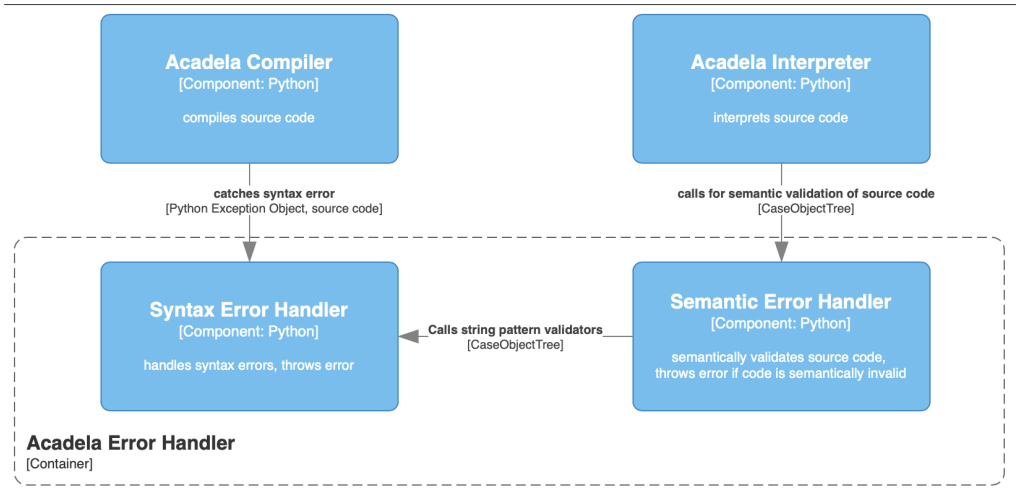

Figure 6.2.: A container diagram of Acadela Error Handler and the interactions of it within the Acadela System

6.4.1. Syntax Error Handler

The Syntax Error Handler component consists of one handler module and two sub-modules, Typo Handler and Keyword Handler, as well as one sub-component which is the String Pattern Validator. The handler module `SyntaxErrorHandler` is called from the compiler script to process the caught syntax errors by textX.

The responsibility of the two sub-modules within this component is to process the default error messages of textX to make them more readable and understandable. The main module, `SyntaxErrorHandler` is responsible for extracting necessary information needed within this process and calling these two sub-modules, the typo, and keyword handlers.

The String Pattern Validator has a completely different functionality within the system and it is only called during the semantic validation phase even though it syntactically validates some string patterns that exist in the CP models. This anti-pattern is caused by some limitations of the Acadela grammar and the textX tool which will be discussed in depth both in the Implementation and Discussion chapters.

The structure and the functionality of the sub-modules and the string pattern validator will be described in the next subsections.

6.4.1.1. Typo Handler

The typo handler module is responsible for processing the syntax errors caused by unexpected elements and checking if the unexpected element detected by textX is a potential typo or not. It consists of two functions `generate_dictionary` which generates a dictionary for the spell-checker using the extracted keywords from the Acadela grammar and the `typo_handler` function which assesses the unexpected element and produces a segment of the error message. The resulting segment is then processed by the keyword handler module.

6.4.1.2. Keyword Handler

The keyword handler module consists of a function, `keyword_handler` that replaces the technical keywords and acronyms in the error messages after the typo handler module processes them. The `keyword_handler` function iterates through a list of tuples which include the keywords and acronyms that need to be replaced and the replacements of them. The function replaces each of the occurrences of these keywords in the error messages. This function also cleans up the unnecessary white spaces and characters from the error messages to reduce their lengths and increase their readability.

6.4.1.3. String Pattern Validator

The String Pattern Validator component is responsible for validating the attributes of elements that need to conform to specific string patterns. The component includes the textX grammars that define the syntax rules of these specific string patterns. Furthermore, the component includes three modules, a separate syntax error handler called `StringPatternSyntaxErrorHandler`, a module for processing and validating the field expressions, and a module for processing and validating the values of the `UIRef` attribute of the field elements.

The module for validating the field expressions, consists of two functions, `validate_expression` and `expression_handler`. The `expression_handler` function traverses the tasks and their fields stored in the `CaseObjectTree` and checks if each field has a `expression` attribute to be validated or not. If a field element has an expression attribute then this function retrieves the exact number of the line that the `expression` is defined in the source code and calls the `validate_expression` function.

The `validate_expression` function then creates a meta-model and parser from the defined grammar and validates the value of the `expression` attribute. If the parser detects a syntax error, it calls the `StringPatternSyntaxErrorHandler` which generates an error message by processing the error message generated by textX using the keyword handler module and includes the line number that the `expression` is located into the message.

The module for validating the `UIRef` attribute of field elements works in almost the same way as the module for validating the field expressions. It includes two functions, `expression_handler` to retrieve the `UIRef` attributes of fields and the exact line number that they are defined and calls the `validate_ui_ref` function to validate these attributes with the corresponding grammar.

6.4.2. Semantic Error Handler

The Semantic Error Handler component consists of three sub-modules for each identified semantic constraint of the Acadela Language. Furthermore, it accesses and calls the String Pattern Validator component of the Syntax Error Handler as the syntactic validation of some string patterns in Acadela takes place after compilation. As mentioned earlier, the reason for this anti-pattern is discussed in the Implementation and Discussion chapters as a limitation of the system.

Unlike the Syntax Error Handler, the execution of these modules are mutually exclusive as each of them validates exclusively one constraint and the result of these validations are independent. For an input source code to be semantically validated, each module is called from the `SemanticErrorHandler` component in a sequentially and executed one at a time. An exception with an error message is thrown in case any one of these three modules catches a semantic error and the execution of the `SemanticErrorHandler` module is terminated.

The functionality and structure of the sub-modules of the Semantic Error Handler are

discussed briefly in the next subsections and their implementations are discussed in detail in chapter 7.

6.4.2.1. ID Uniqueness Checker

The ID uniqueness checker module is responsible for validating the ID uniqueness constraint of the CONNECARE system and consists of two functions, namely `check_id_uniqueness` and `find_duplicate`.

The `check_id_uniqueness` function traverses the lists of each element type stored in the `CaseObjectTree` and extract a list of identifiers/names for each element type. Then it calls the `find_duplicate` function which checks each of these lists for duplicate values and throws an exception with an error message if it finds one.

6.4.2.2. Path Validity Checker

The path validity checker module is responsible for validating that the paths defined for referencing elements such as `Tasks`, `Stages`, `InputFields` and `OutputFields` from other elements which belong to different parent elements are semantically correct.

The element paths are defined using a dot notation, where each element is separated by a dot and an element on the right-hand side of a given dot is a child of the element on the left-hand side.

The path validity checker module consists of three functions, `check_path_validity`, `parse_field_expression` and `parse_precondition`. The main function of this module, `check_path_validity` is called initially by the Semantic Error Handler. This function traverses the `CaseObjectTree` and for each element that contains a path as an attribute for referencing, checks the validity of these paths and throws an exception with a custom error message if the paths are found to be invalid. The validation of some of these paths is relatively straightforward and does not involve specific constraints.

However for more complex paths with more complex constraints, specifically the paths defined as part of `expression` attributes of the `OutputField` elements and the `previousStep` and `condition` attributes of the `Precondition` elements, the `parse_field_expression` and `parse_precondition` functions are called respectively.

The `parse_field_expression` function extracts the element identifiers defined in the `expression` attributes using a regular expression and checks the existence of each of these identifiers in the case definition.

The `parse_precondition` function validates the `previousStep` and `condition` attributes of the `Precondition` elements according to the constraints defined specifically for the paths defined in these attributes.

6.4.2.3. URL Validator

The URL Validator module which consists of a single function, `url_validator`, validates the trusted URL constraint of the CONNECARE system. The `url_validator` function iterates through the `Hook` elements defined in a case definition, using the `CaseObjectTree` and checks if the attributes of this `Hook` elements satisfy the trusted URL conditions which are defined in the Constraint Identification chapter.

7. Implementation

This chapter describes the implementation of the syntax and semantic error handlers of the Acadela language as well as how it is integrated into the Acadela System's other components such as the Acadela IDE and the Acadela Backend. It also provides answers to the second research question, *"How to identify and interpret the errors to modelers in an accurate, user-friendly manner?"*.

7.1. Syntax Error Handler

As mentioned earlier, textX has a simple, built-in syntax error handler that was already in use. However, even though the built-in syntax error handler is capable of detecting syntax errors to some extent, the produced error messages tend to have very technical language and do not include the exact cause of the error clearly. This section describes the implementation details of the new syntax error handler.

During the parsing of source code or in other words the models, textX raises an exception of type `TextXSyntaxError` if a syntax error is detected. The raised exception has three attributes, namely `message`, `line`, and `col`. The `message` attribute contains the default error message generated by textX, the `line` attributes contain the line number and the `col` attribute contains the column where the error was found [21]. The newly implemented syntax error handler component uses these attributes as a baseline for the new error messages and extends these error messages to increase the readability and the understandability of it by the user.

In the new syntax error handler, after a syntax error has been detected by textX during the parsing of the model, the main function of the new syntax error handler, `handleSyntaxError`, is being called. In the handler, the logic for processing errors consists of the following steps:

1. Checking which file the error is located and retrieving the file content
2. Extracting keywords of elements and attributes from the Acadela grammar
3. Handling potential typos and unexpected elements
4. Handling technical keywords, removing unnecessary parts of the default error message, and constructing the new error message
5. Raising a new exception with the constructed error message

It is crucial to note that, unlike the semantic error handler which is described in detail in section 7.2, the syntax error handler does not handle the error types and constraints which are identified in chapter 5 separately. Instead, all error types are processed with the same logic and therefore with the same steps listed above. The reason for this is that all of the syntax errors are in fact can be identified as "unexpected element" errors. Furthermore, even though the string pattern validations are covered in this section as they are categorically syntactic validations, they are being detected after the meta-model construction and compilation, more specifically during the interpretation phase together with the semantic validations. This design decision is discussed further in subsection 7.1.2. As a result, string pattern validations are not a part of the steps listed above.

The first step which is checking which file the error is located is a crucial step to display correct error messages to the user. As the new error handler displays more detailed error messages with suggestions for the user, having the full content of the line that an error has occurred is required. However, the `message` attribute of `TextXSyntaxError` only includes the first few characters of the line starting from the column that error starts which can be seen in the example error message below:

```
error: Expected Hash or 'label' or 'Question' or 'CustomFieldValue' or
'uiRef' or 'externalId' at position /xxxx/xxxxx/dsl-connecare/acadela/
model_placeholder:(112, 21) => '          *labl = 'Sy'.
```

For this reason, the new error handler extracts the full line from the source code of the model using the `line` attribute of the exception object. For errors that occur in the main in other words the parent source-code file, this process is quite trivial. However, in case an error occurs in an external file imported into the main model, that file has to be located and its content should be processed instead of the parent model to produce accurate suggestions for the user.

The second step mentioned above, extracting attributes from the grammar, is also a "pre-processing" step for the following steps. In this step, the keywords, in other words, the reserved words in the language are extracted from the Acadela grammar. The dictionary which is required for the spell-checker used in the typo handler (step 3) is constructed from these extracted keywords. The implementation of this step is further detailed in the "Unexpected elements and Typos" section.

The third step, handling potential typos and unexpected elements, is among the syntax error types and constraints identified during the planning phase of this thesis. As described before, "typo detection" is included in the system mainly to guide the users to the error more clearly and further guide the user to fix the error. The implementation of this step is thoroughly described in subsection 7.1.1 as well.

The fourth step, which is also described in detail in subsection 7.1.1, is a step to make the error messages more understandable and accurate for the users. It involves replacing the technical terms and acronyms with less complex ones as well as removing

unnecessary parts of the original error message which may confuse and mislead the user as they do not contribute any useful information.

Last but not least **the fifth step** involves raising a `Python exception` with the error message that is generated in the previous steps.

7.1.1. Unexpected Elements and Typos

As mentioned in the previous chapters, one of the main goals of the customized error handler is to increase the usability of the Acadela language and increase the ease of debugging models of CPs.

The syntax error type "unexpected elements" refers to the attributes and elements that are not placed in the correct parent element and the commands that include elements that do not exist in the Acadela grammar and are consequently not recognized by the parser of the language generated by textX.

The default syntax error handler of textX produces error messages in the same format for every possible syntax error which falls under the category of unexpected elements. The format of these error messages can be roughly understood from the example error message below:

```
Expected 'mandatory' or 'notmandatory' or 'readOnly' or 'notReadOnly' or
'link' or 'notype' or 'text' or 'longtext' or 'boolean' or 'number' or
'singlechoice' or 'date.after(TODAY)' or '(date)' or 'json' or 'custom'
at position /Users/macbookpro/dsl-connecare/acadela/model_placeholder:
(117,28) => '   #left #*exactlyOne'.
```

This error message was produced by the following code segment:

```
OutputField SystolicAnalysis
    #left #exactlyOne
    label = 'Systolic Assessment:'
    uiRef = use rgu.redGreenUiRef
```

The directory `exactlyOne` which is defined under the rule `Multiplicity` in the grammar is an unexpected element for an `OutputField` as the rule for `OutputField` elements only expects the directives from the `Mandatory`, `ReadOnly`, `Position` and `Type` rules as viable directives. Hence, the error message produced by textX includes the directories that belong to these rules which are:

1. **Directives defined in the Mandatory rule:** `mandatory`, `notmandatory`
2. **Directives defined in the ReadOnly rule:** `readOnly`, `notReadOnly`
3. **Directives defined in the Type rule:** `link`, `notype`, `text`, `longtext`, `boolean`, `number`, `singlechoice`, `date.after(TODAY)`, `(date)`, `json`, `custom`

4. **Directives defined in the Position rule:** No directives listed as the directive `left` from this rule is already in the code.

From this example, it can be concluded that the textX's default error messages correctly identify the potential elements that can replace the unexpected element in the error. However, the messages lack context and do not offer any suggestions to solve the error. Therefore it is already decided that the new custom error handler would use the potential elements and attributes identified by textX and extend the error messages to give more context to the user, consequently increasing their readability and understandability.
To increase the usability and user-friendliness of the new error messages for the user, it is decided to integrate a spell-checker to the new error handler to detect potential typos that the user made that cause syntax errors.

It is decided that using a pre-existing spell-checker is a better option compared to implementing a spell-checker from scratch due to time limitations and reliability concerns. Hence, research is conducted to find a spell-checker that suits the requirements of the system and after considering several spell-checker libraries available in Python, it is decided to use the library `pyspellchecker`[1].

The main motivation behind this decision is that, `pyspellchecker` allows creating custom dictionaries, unlike other spell-checker libraries which only allow using their built-in English dictionaries. The aim of using a spell-checker is to find potentially misspelled keywords from the Acadela language and considering all potential candidates from the English language can lead to misleading and incorrect error messages, so it is crucial to keep the candidates limited to the keywords that exist in the Acadela language. Furthermore, there are some keywords in the Acadela language which does not exist in a regular dictionary such as `HumanTask` or `notmandatory` that are required to be in the dictionary of the spell-checker for more accurate and complete suggestions.

The Python library `pyspellchecker`, uses a Levenshtein distance algorithm to find permutations within an edit distance of one or two from the original word and returns the set of possible candidate words for the misspelled word [22]. As described before, Levenshtein distance is a metric for measuring the "distance" between two words where the distance refers to the minimum required number of single-character edits such as insertions, deletions, and substitutions to change one word to another [2].

After the decision is made on which spell-checker library to use, an algorithm to detect and process typos is constructed. The steps included in this algorithm are as follows:

1. Extract and store the keywords from the Acadela grammar for constructing the dictionary
2. Initialize and prepare `pyspellchecker`
3. Detect the potentially misspelled word

[1] pyspellchecker documentation

4. Use the `pyspellchecker` library to identify the potential candidates

5. Check if the keyword is a directive and if so check if the directive sign, # is missing or not

6. Check if any keywords in the dictionary include the "misspelled" keyword as a substring

To use the spell-checker, a dictionary in the required `JSON` format is generated from the Acadela grammar file. To extract the keywords from the grammar of the Acadela language, several regular expressions are used. The two possible string patterns that keywords that needs to be extracted from the grammar are as follows:

- Simple string values with quotation marks e.g. `staticId`, `name`
- String values accompanied with white space notation e.g. `/(SummaryPanel)\s/`

The regular expression `\/\([a-zA-Z]+\)\\s\/|\[a-zA-Z]+'` is used to find all the matching keywords in the grammar and extract them. Then the non-alphabetic characters are removed from each extracted keyword. Finally, the same arbitrary value of 10 is assigned as the weight for all of the extracted keywords as it is assumed that all keywords have the same probability of occurrence and there is no need for tie-breakers. Finally, the dictionary file is saved and the `PySpellChecker` instance is initialized with this custom dictionary and the edit distance of 2.

Additionally, the directive keywords are extracted separately to be used in **step 6** as it is thought that the user might potentially forget the directive sign, # and this should be covered as a specific case in the error messages.

In the **step 3**, the potentially misspelled word which is, in fact, the "unexpected element" that caused textX to throw a syntax error is extracted from the original line of the error using the `line` and `column` values from the original exception object.

Then, in the **step 4**, the candidate keywords for the potentially misspelled word are found using the spell-checker. The `candidates` function of the `PySpellChecker` instance returns the most likely word found in the dictionary with the edit distance provided and if there are no matching words, it returns the misspelled word itself. If the candidate keyword is different from the misspelled word then the algorithm is designed to return a segment of the new error message that suggests this candidate keyword.

At **step 5**, the algorithm checks if the "unexpected element" matches any of the directive keywords in the grammar and if this is the case whether the keyword preceded by the directive sign, #. If this is the case, an error message segment that implies the user might have forgotten the # character is returned.

In addition to spell-checking, at **step 6**, the algorithm checks whether the "unexpected element" has any of the keywords from the grammar as a substring or not. This step is added to cover the cases where a whitespace is forgotten after a keyword. If there is no candidate keywords found from **step 4** and **step 5** and a matching keyword is found in

this step, the algorithm returns an error message segment that suggests this matching keyword.

Last but not least, if all of the checks from **steps 4, 5, and 6** fails it is assumed that there are two remaining possible causes for the caught exception:

- **If the unexpected keyword is among the keywords extracted from the grammar:** The keyword is misplaced i.e. does not belong to the parent element as an attribute or child element
- **If the keyword does not match any keyword from the grammar in any way:** The keyword is unrecognized and the user made a syntax error that is not foreseen

In these six steps, the typo handler produces a segment of the error message which gives suggestions for suspected typos and/or explains the cause of the error in a more user-friendly manner. Even though the main focus of these steps is detecting potential typos, further assessment of "unexpected elements" is also completed in this process simultaneously.

After the typo handling process is done, the remaining steps include reducing the complexity and increasing the readability and understandability of the error messages as a whole. These steps are:

- **Replacing keywords with less technical terms:** This step involves replacing technical keywords and acronyms with more understandable alternatives which require less technical knowledge i.e. replacing `INT` with "*number*", `STRING` with "*text with quotation marks*" ...
- **Removing the path of the file that the error is located:** As the error messages are being displayed directly in the Acadela IDE, the file path is irrelevant and misleading for the user. To remove the file path, the following regular expression which matches all possible path patterns for all operating systems is constructed:

  ```
  ((((?<!\w)[A-Z,a-z]:)|(\.{1,2}\\))([^\b%\/\|:\n\"]*))|("\2([^%\/\|:
  \"]*)")|((?<!\w)(\.{1,2})?(?<!\/)(\/((\\\b)|[^ \b%\|:\n\"\\\/])+)+\/?)\:
  ```

- **Removing unnecessary white-spaces and characters:** In the original textX error messages which are used as the baseline for the new error messages include redundant white-spaces and characters which need to be removed to increase the readability and reduce the length of the error messages.

Finally, a new exception is thrown with the new error message and consequently the execution of the Compiler script is terminated.

The final version of error messages are illustrated below with example code segments with syntax errors and their corresponding error messages:

- An example code segment with a syntax error and its corresponding error message for a case where the keyword that caused the error is a typo with an edit distance of 1 so that the spell-checker successfully detects it and returns a candidate keyword:

```
ownneer = 'Setting.Nurse'
```

```
Syntax Error! Unrecognized command at line 179 and column 13!
No keyword ownneer. Did you mean: owner?
Expected one of:
1. Hash sign (#)
2. 'label'
3. 'owner'
4. 'dueDateRef'
5. 'externalId'
6. 'additionalDescription'
7. 'dynamicDescriptionRef'
8. Precondition at position (179, 13) => '*ownneer = '.
```

- An example code segment with a syntax error and its corresponding error message for a case where there is a white space related syntax error (detected by the **step 6** of the typo handling algorithm):

```
GroupUmcgPhysicians name = 'Umcg Physician'
```

```
Syntax Error! Unrecognized command at line 14 and column 9!
No keyword GroupUmcgPhysicians. Did you mean: Group?
Expected one of:
1. Group
2. User
3. Setting
4. Ref
5. 'Trigger'
6. SummaryPanel
7. Text with quotation marks ("") at position (14, 9) => '*groupUmcgP'.
```

- An example code segment with a syntax error and its corresponding error message for a case where the keyword that caused the error is in fact a keyword for an attribute that exists in the language but does not belong to the specified element i.e. is not an attribute of the parent element:

```
Precondition
            dueDateRef = 'Setting.WorkplanDueDate'
            previousStep = 'Evaluation'
```

```
Syntax Error! Unrecognized command at line 174 and column 13!
Unexpected keyword: dueDateRef
Expected one of:
1. 'previousStep'
2. 'condition'
3. Precondition
4. 'client'
5. 'dynamicDescriptionRef'
6. 'externalId'
7. 'additionalDescription'
8. HumanTask
9. Ref
10. AutoTask
11. DualTask at position (174, 13) => '*dueDateRef'.
```

- An example code segment and its corresponding error message for a case where the keyword that caused the error is not a known keyword and there are no suggestions available:

```
Precondition
            nextStep = 'Evaluation'
            condition = 'Evaluation.RequestMedicalTest.CholesterolTest = 0'
```

```
Syntax Error! Unrecognized command at line 195 and column 13!
Unrecognized keyword: nextStep
Expected one of:
1. 'previousStep'
2. 'condition'
3. Precondition
4. 'client'
5. 'dynamicDescriptionRef'
6. 'externalId'
7. 'additionalDescription'
8. HumanTask
9. Ref
10. AutoTask
11. DualTask at position (195, 13) => '*nextStep ='.
```

- An example code segment and its corresponding error message for a case where the user forgets inserting the directive sign (#) before the directive keyword (detected by the **step 5**)

```
OutputField OverallAssessment
   #left custom
```

```
Syntax Error! Unrecognized command at line 138 and column 27!
The keyword custom is a directive value. Did you mean #custom?
Expected one of:
1. Hash sign (#)
2. 'label'
3. 'additionalDescription'
4. 'uiRef'
5. 'CustomFieldValue'
6. 'expression'
7. 'externalId' at position (138, 27) => '    #left *custom    '.
```

7.1.2. String Pattern Validations

Within the Acadela language, several fields are expected to follow a string pattern. However, in the original Acadela grammar, these fields have the type `STRING` so the content of these fields was not being syntactically validated. As the content of these fields is being used to visualize and assess CPs in the SACM, these fields have to be validated before the system sends the CP model to SACM.

Initially, these fields are planned to be validated via regular expression. However, using only regular expressions to validate these fields failed not because they cannot be validated but the validation only provided information on whether the fields match the regular expressions or not. As a consequence, the location and source of the error were missing in the error messages. For this reason, it is decided to use textX for these fields as well, by creating separate grammars for each pattern and customizing the error messages afterward. The syntax errors that are produced by these grammars did not require much customization except for simplifying the keywords in the error messages in the same way that is described in the previous section. It is important to note that, creating separate mini grammars for these fields was a design decision which is made in order not to further complicate the original Acadela grammar as well as due to several limitations of textX and the original Acadela grammar. These limitations are discussed in detail in the Discussion chapter.

Due to this design decision, the syntactic validations of these fields take place after the compilation, and before the semantic validation. As there are a very limited number of short keywords in these new grammars, a spell-checker is not used and only the steps from the previous "Unexpected Elements and Typos" section that include reducing the complexity and increasing the readability and understandability of the error messages are applied to the error messages generated by textX.

7.1.2.1. Round Expression

Round expression is one of the two potential string patterns that the value of the `expression` attribute of the `OutputFields` should follow along with conditional expression. The structure of the `round` expression can be observed from the following example:

```
expression = 'round(Weight / (Height * Height))'
```

As its name suggests, the expression is used for rounding the results of arithmetic operations which can involve `INT` values, `Field` identifiers, and `number` functions which convert the values of `InputFields` or `OutputFields` with the type `STRING` to number values i.e. `number(Weight, 2)`.

The implemented grammar for `round` function includes a parent rule `RoundFunction` which defines the `FunctionName`, round, and the `ArithmeticOperation` rule which is positioned between the outermost function parentheses. The `ArithmeticOperation` rule is both left and right recursive rule which is "`ArithmeticTerm ArithmeticOperator`

ArithmeticTerm" where the ArithmeticTerm can be a STRING, INT, number function or again an ArithmeticOperation. The ArithmeticOperator rule simply includes simple arithmetic operators: +, -, *, /

The final grammar for the round function which also includes the grammar for number function is as follows:

```
//round function grammar
RoundFunction:
FunctionName '('(ArithmeticOperation)')'
;
FunctionName:
    'round'
;

ArithmeticOperation:
    ArithmeticTerm ArithmeticOperator ArithmeticTerm
;

ArithmeticTerm:
    numberFunction
    | TextNoQuote
    | "(" ArithmeticOperation")"
    | INT
;

numberFunction:
    numberFunctionName '('TextNoQuote ',' num=INT ')'
;

numberFunctionName:
    'number'
;
TextNoQuote:
    /([a-zA-Z0-9-_.]*)/
;
ArithmeticOperator:
    '+'
    | '-'
    | '*'
    |'/'
;
```

Example error messages generated via the new grammar are as follows:

- Missing right parentheses of the `round` function

```
OutputField DiastolicCalculation
       label = 'Diastolic calculation for the field'
       expression = 'round(Diastolic + number(Age, 2)'
```

```
Syntax Error! Invalid expression at line 131:
 Expected ')' at position  => 'Age, 2)*'.
```

- Invalid arithmetic operator

```
OutputField DiastolicCalculation
       expression = 'round(Diastolic % 4)'
```

```
Syntax Error! Invalid expression at line 131:
 Expected one of:
1. '+'
2. '-'
3. '*'
4. '/' at position => 'Diastolic *% 4)'.
```

7.1.2.2. Conditional Expression

Conditional expression, more specifically if-else expression, is one of the string patterns that the value of `expression` attribute of the `OutputField` can follow together with `round` expression from the previous section. The `expression` attribute states how to conditionally output a value based on input(s). The structure of the conditional expression in Acadela, which is essentially an `if-else` statement is as follows:

```
if (Predicate1)
    then Output1
else if (Predicate2)
    then Output2
else Output3
```

Where a predicate consists of a logical expression in the form of:

```
(Variable Operator Variable and/or Variable Operator Variable)
```

The variables in the predicates can be a constant or a path to an `InputField` or `OutputField`. The type validation of this field as well as in the case of a path value, the validation of the reference (i.e. whether it points to a valid task and field) are done separately in part of the semantic validation. A predicate can consist of multiple predicates

connected via an `and/or` operator. An example value of an expression attribute that follows this string pattern can be seen below:

```
expression =  if (Diastolic < 80 and Systolic < 120)
                then ‘‘Normal"
              else if (Systolic < 130)
                then "Elevated"
              else "High"’
```

To validate the `if-else` statements syntactically, the following new grammar is implemented using textX.

```
IfElseStatement:
    ifStatement elseIfStatement* elseStatement
;
ifStatement:
    /(if)\s/’(’ (conditionStatement) ’)’ thenStatement
;
elseIfStatement:
    /(else\sif)\s/ ’(’(conditionStatement) ’)’ thenStatement
;
elseStatement:
    /(else)\s/ STRING
;
conditionStatement:
    TextNoQuote Comparator NUMBER (andOr TextNoQuote Comparator NUMBER)*
;
Comparator:
    ’=’
    | ’<>’
    | ’<=’
    |’>=’
    | ’<’
    | ’>’
;
TextNoQuote:
    /([a-zA-Z])*/
;
andOr:
    /(and)\s/ | /(or)\s/
;
thenStatement:
    "then" STRING
;
```

The `if-else` statements in Acadela are required to have an `if` and an `else` statement specified. `Else-if` statements are optional. Therefore, in the textX grammar implemented, the `IfElseStatement` parent rule has a combination of one `ifElseStatement`, followed by zero or more `elseIfStatement`, followed by exactly one `elseStatement`. In Acadela, the statements(outputs) which follow `then` or `else` keywords can be only of type `STRING`. This constraint can be observed from the `elseStatement` and the `conditionStatement` rules. As mentioned before, a predicate can consist of multiple predicates connected via the `and` and `or` keywords. Hence, the `conditionStatement` rule is designed as a right recursive rule. Lastly, the white-space character `\s` is explicitly inserted before and after keywords in each rule to detect white-space-related syntax errors.
Example error messages generated via the new grammar are as follows:

- A code segment of an expression with a missing parentheses after the first predicate. The error message also suggests `and` and `or` keywords as a predicate can be combined with another predicate.

```
expression = 'if (Diastolic < 80 then "Normal"
                else "High"'
```

```
Syntax Error! Invalid expression at line 138:
Expected one of:
1. 'and'
2. 'or'
3. ')' at position  => 'olic < 80 *then "Norm'.
```

- Missing white space between `else` and `if`

```
expression = ' if (Diastolic < 80) then "Normal"
               elseif (Diastolic <= 89) then "Elevated"
               else "High"'
```

```
Syntax Error! Invalid expression at line 138:
Expected one of:
1. 'else if'
2. 'else' at position  => ' "Normal" *elseif (Di'.
```

- Missing comparator between the `InputField` identifier and the value being compared

```
expression = ' if (Diastolic < 80) then "Normal"
                else if (Diastolic  89) then "Elevated"
                else "High"'
```

```
Syntax Error! Invalid expression at line 138:
 Expected one of:
1. '='
2. '<>'
3. '<='
4. '>='
5. '<'
6. '>' at position  => 'Diastolic *89) then "'.
```

7.1.2.3. User Interface Reference

The value of the user interface reference attribute of `OutputField`, defined with the keyword `UIRef`, is used for visualizing the value of a given field in the SACM. The value of this attribute can be one of the following options:

- `colors` function i.e. `uiRef = 'colors(5<orange<=18<green<=25<red<100)'`
- `privatelink`
- `hidden`
- A segment of `JavaScript` or `HTML` script in quotation marks

To validate this field syntactically a new textX grammar is constructed. However, as syntactically validating JavaScript and HTML code segments is very complex and an open-source grammar or a parser is not readily available for this purpose, this case is left out from the syntax error handler on purpose. Hence, the new grammar only validates the first three options.

Defining the constant string options, `privatelink` and `hidden` is trivial so that in the new grammar they are covered directly in the parent rule `UiRef` with ordered choice operator `|`.

The `Function` rule of the grammar includes the `ValidFunctionName` rule which includes the function name `colors`, an open parenthesis, the function body defined by the rule `CompareExpression` and the close parenthesis.

The `CompareExpression` rule is a right recursive rule in which the expression always starts and is followed by at least one or more expressions with the pattern:

```
(Comparator ColorName Comparator NUMBER)
```

Hence the body of the `color` function also always ends with a number. The `ColorName` rule includes the available colors which can be displayed in the UI and the `Comparator` rule includes the available comparators in the grammar.

The final grammar defined for the User Interface Reference attribute, `UIRef` is as follows:

```
//grammar for uiRef

UiRef:
    Function
    | 'privatelink'
    | 'hidden'
;
Function:
    ValidFunctionName'('(CompareExpression)')'
;
CompareExpression:
    NUMBER (Comparator ColorName Comparator NUMBER)+
;
ColorName:
    'red'
    | 'blue'
    | 'green'
    | 'orange'
    | 'yellow'
;
ValidFunctionName:
     'colors'
;
Comparator:
    '='
    | '<>'
    | '<='
    |'>='
    | '<'
    | '>'
;
```

The error messages generated via the new grammar can be understood by the following examples:

- Missing parentheses after the function name `colors`

  `uiRef = 'colors0<green<20<yellow<=139<red<=300)'`

```
Syntax Error! Invalid expression at line 113:
 Expected '(' at position  => 'colors*0<green<20'.
```

- Misspelled the function name: color instead of colors

```
uiRef = 'color(0<green<20<yellow<=139<red<=300)'
```

```
Syntax Error! Invalid expression at line 113:
Expected one of:
1. 'colors'
2. 'privatelink'
3. 'hidden' at position  => '*color(0<gr'.
```

- Missing comparator between the color and the number value

```
uiRef = 'colors(0<green20<yellow<=139<red<=300)'
```

```
Syntax Error! Invalid expression at line 113:
Expected one of:
1. '='
2. '<>'
3. '<='
4. '>='
5. '<'
6. '>' at position  => 'rs(0<green*20<yellow<'.
```

7.2. Semantic Error Handler

As mentioned earlier, unlike syntax errors, textX does not have a built-in semantic error handler that can be readily used. This is because the syntax of the Acadela language is solely determined by its grammar defined in textX similar to the other languages and consequently the compiler can recognize syntax errors during compilation. However, as semantics refer to the meaning associated with statements in a language, the semantic rules need to be defined separately from the grammar. The semantic, in other words logical, validations cannot be possibly done by the grammar alone. Furthermore, a given model has to be validated semantically after compilation (more specifically during parsing) and during the interpretation phase as the AST which is required to perform semantic analysis can only be generated during compilation by textX.

textX library automatically builds a meta-model and parser for the language from the language grammar description. The built parser then parses the source code and again automatically builds a model, which is a graph of Python objects, corresponding to the meta-model. For each grammar rule, a Python class is dynamically created. These Python classes are instantiated during the parsing of the input, in the case of Acadela, the models of CPs, to create a structure similar to an AST, which is a graph of python objects. Each object is an instance of a class of the meta-model [21].

After the model is generated from the Acadela grammar and the input CP model is parsed, the AST-like graph is generated. After this stage, this graph is interpreted into a Python dictionary for convenience and used for semantic validation. The constructed Python dictionary, which will be from now on referred to as `CaseObjectTree` throughout this chapter is being used to store the elements, their corresponding child elements, and attributes with the following keys: `workspace`, `groups`, `users`, `entities`, `stages`, `tasks`, `case`, `attributes` and `settings`. The values of each of these keys are lists of objects of the corresponding elements.

In chapter 5, the semantic rules which need to be validated during semantic analysis and the corresponding semantic errors that need to be detected to prevent erroneous logic that might produce wrong results are explained in detail. These semantic error types are validated using the `CaseObjectTree` during the interpretation phase in the following order as a part of the semantic validation:

1. ID uniqueness validation
2. Reference and path validation
3. Trusted URL validation

For an input model to be semantically validated, each validation is performed one at a time and an exception with an error message is raised in case any one of the validator modules listed above catches a corresponding semantic error. The implementations of these modules are further explained with the example error messages produced, in the rest of this chapter.

7.2.1. ID Uniqueness Validation

As mentioned in previous chapters, the elements of the CPs modeled in the CONNECARE system have to have unique identifiers as a constraint. For this reason, within the semantic error validator, a module for checking the uniqueness of the identifiers of specific elements is implemented. The elements that do require unique identifiers(ids) are:

- **Groups:** Each `Group` has to have a unique name which is the identifying attribute of the `Group` elements
- **Users:** Each `User` element has to have a unique name which is the identifying attribute of the `User` elements
- **Stages:** Each `Stage` element has to have a unique ID
- **Attributes in Settings:** Each `Attribute` listed under the `Setting` element has to have a unique ID
- **Tasks:** The IDs of `Task` elements of all `Stage` elements have to be unique
- **Form Fields:** The IDs of the `InputFields` and `OutputFields` of all `Form` elements have to be unique.

Validating the uniqueness of element IDs is accomplished via a relatively simple process that involves the following steps:

1. Get the lists of each element type listed above from the `CaseObjectTree`
2. Extract the list of names or IDs from each list of elements
3. Check the lists from the previous step for duplicate values
4. If there are duplicate values, get the `lineNumber` attribute of the first occurrence of the duplicate element
5. Throw an exception with an error message that includes the element type, the non-unique ID, line, and column numbers.

The same steps listed above are repeated for all the elements which have the uniqueness constraint. The only exception is that due to the edge constraint regarding IDs which requires the IDs of `Attributes` in treatment `Settings` to be mutually exclusive from `Stage` IDs in addition to the uniqueness constraints that each element type already has. For this reason, a set difference check for each element types' identifiers and names is done.
An example error message for this category of semantic errors is as follows where two `Field` elements have the same identifier:

```
Error: Field IDs should be unique! SystolicAnalysis at line 124 and column
17 is a duplicate. Please verify that the IDs are unique for each field.
```

7.2.2. Reference and Path Validation

In the Acadela language, elements can be referenced from other elements using a path with a dot notation. For instance, an `InputField` named `CholesterolTest` of a `Form` named `CgiForm` from a `HumanTask`, `RequestMedicalTest` can be referenced from another `Task` using the path using a dot notation as follows:

```
Evaluation.RequestMedicalTest.CholesterolTest = 1
```

In Acadela, there are several attributes of elements that can include references using paths. However, in the grammar of the language, these attributes have the type `STRING` so these references and paths are not part of the grammar itself and need to be validated after compilation. Furthermore, the `CaseObjectTree` which is constructed after compilation, is needed in order to perform the necessary checks for whether the paths that are used for references point to both valid and existing elements.

The elements and their attributes which can include a reference listed below with an example for each:

- owner attribute of `Task` elements

  ```
  owner = 'Setting.Clinician'
  ```

- due date attribute of `Task` elements, `dueDateRef`

  ```
  dueDateRef = 'Setting.WorkplanDueDate'
  ```

- `Task Precondition` where both `previousStep` and `condition` attributes can include paths

  ```
  Precondition
         previousStep = 'MeasureBloodPressure'
         condition = 'Setting.BloodPressureCondition = "High"'
  ```

- `CustomFieldValue` attribute of an `OutputField` or an `InputField` of a `Form`

  ```
     InputField SelectDoctor
              #custom
              CustomFieldValue = "Setting.Clinician"
  ```

- `expression` attribute of a `DynamicField`

  ```
  expression = 'if (Diastolic < 80 and Systolic < 120) then "Normal"
                else "High"'
  ```

- the value of an `InfoPath` element of a `SummaryPanel Section`

  ```
  SummaryPanel
          Section BloodPressureMeasurement #left
              InfoPath Evaluation.MeasureBloodPressure.Systolic
  ```

- `owner` attribute of `Stage` elements

  ```
          owner = 'Setting.CaseOwner'
  ```

- `Stage Preconditions` where both `previousStep` and `condition` attributes can include paths

```
Precondition
    previousStep = 'Evaluation'
    condition = 'Evaluation.RequestMedicalTest.CholesterolTest = 1'
```

All of the attributes with references listed above are semantically validated following the same simple logic where the path is split by the dot notation (.) and the reference is validated from left to right by checking if each element name is valid. The requirement for a reference to be valid is that its path points to an existing element as well as it follows the correct order of elements. However, there are also several case-specific constraints for the references listed above.

- **Task and Stage owner**
  The value of the owner of a given task or stage should be defined under case settings and it should be referred from the owner attribute in one of the following formats:
  - `Setting.CasePatient`: `CasePatient` is a keyword which refers to the list of patients in the e-health system
  - `Setting.CaseOwner`: `CaseOwner` is a keyword which refers to the responsible person or group that executes the CP
  - `Setting.<AttributeName>`: Attributes are global variables

  If the owner is either one of the `CasePatient` or `CaseOwner` attributes, then the name of these attributes should be among the defined `groups` of the case definition. Furthermore, if the owner is not either one of `CasePatient` or `CaseOwner` and an `Attribute`, then the owner should be linked to a `group` using the function `#Link.Users(<groupName>)` as in the following example `Setting` definition:

```
Setting
    CaseOwner UmcgProfessionals #exactlyOne
        label = 'UMCG Professionals'

    CasePatient UmcgPatients #exactlyOne
        label = 'Patient'

    Attribute Clinician
        #exactlyOne #Link.Users(UmcgClinicians)
        label = 'Clinician'
```

  where each of "UmcgProfessionals", "UmcgPatients" and "UmcgClinicians" refer to a `group` element. The algorithm to semantically validate the `owner` attribute of `Task` and `Stage` elements, therefore, involves looping through the task and stage lists from the `CaseObjectTree` and checking the constraints described above by applying the following steps for each element:

1. Split the dot-separated path by using dot as a separator into a list
2. Check if the value of the left-most element(first element of the list) is `Setting` and throw an error if it is not.
3. Check if the right-hand side of the dot notation (second element of the list) is `CaseOwner` or `CasePatient`. If this is the case, check if a group with the same name as the `CaseOwner` or `CasePatient` exists and if not throw an error.
4. If the owner is neither the `CaseOwner` nor the `CasePatient`, check if there is an `Attribute` with the same name as the right-hand side of the dot notation. If there is, proceed to the next step, otherwise throw an error.
5. Check if the value passed to the `Link` function of the attribute from the previous step matches a `group` element and throw an error if it does not.

If there is no error thrown in any of these steps, the path defined in the `owner` attribute is assumed to be semantically correct.

Example semantic error messages for semantically invalid statements:

- No `Attribute` with the name "Doctor" in `Setting`

```
owner= 'Setting.Doctor'
```

```
Semantic Error at line 105! Owner 'Doctor' not found in settings.
```

- An `Attribute` with name "Clinician" exists but the linked `group` "UmcgClinician" does not exist

```
Attribute Clinician
  #exactlyOne #Link.Users(UmcgClinician)
  ........
  .......
  owner= 'Setting.Doctor'
```

```
Semantic Error at line 38! User 'UmcgClinician' is not found in
groups.
```

- **Task and Stage due date**
  The value of the due date attribute of `Task` and `Stage` elements, denoted by the keyword `dueDateRef`, should refer to a due date attribute defined in the `Setting` element of the case. The referenced due date attribute should also have the type `date` as in the following example:

```
Attribute WorkplanDueDate
   #exactlyOne #date.after(TODAY)
   label = 'Workplan Due Date'
```

  The algorithm to semantically validate the `dueDateRef` attribute of `Task` and `Stage` elements involves looping through the task and stage lists of the `CaseObjectTree` and checking these constraints by applying the following steps to each element:

  1. Split the dot-separated path by using dot as a separator into a list
  2. Check if the value of the left-most element(first element of the list) is `Setting` and throw an error if it is not.
  3. Check whether the right-hand side of the dot notation, the attribute name, exists in the list of attributes of `Setting` element and if it does not, throw an error, otherwise continue to step 4.
  4. Check if the matched attribute has the type `date` and throw an error if it has a different type.

  Again, if there is no error thrown in any of these steps, the owner attribute is assumed to be semantically correct.

  Example semantic error messages for semantically invalid `dueDateRef` attributes are as follows:

  - No due date attribute with the name "WorkplanDate" in the case `Setting`

```
dueDateRef = 'Setting.WorkplanDate'
```

```
Semantic Error at line 106! Due date "WorkplanDate" not found in
Settings
```

  - An attribute with the name "WorkplanDueDate" exists in case `Setting` element but its type is `number`, not `date`

```
Attribute WorkplanDueDate
    #exactlyOne #number
```

```
Semantic Error at line 41! Due date value is not in date format.
```

- **Task and Stage preconditions**
  A `Precondition` element's `previousStep` attribute can refer to another `Task` or `Stage` element by its name. Each `Task` and `Stage` element can have one or more `previousStep` attributes. The `condition` attribute on the other hand, which is a non-mandatory attribute, can include a conditional expression based on another element and might include a reference to another element with a path in either one of the following formats:

  - `<Stage>.<Task>.<Field>`

  - `Setting.<AttributeName>`

  To semantically validate these two attributes, the following steps are repeated for each `Precondition` element that belongs to a `Stage` or `Task`:

  1. For each `previousStep` of the `Precondition` element, check if its value refers to a matching `Stage` or `Task` element. If it does not, throw an error, otherwise, continue to the next step
  2. Check if the `Precondition` element has a `condition` attribute and if it does proceed to the following step
  3. If the left-most element of the path is not `Setting` proceed to step 4, otherwise, check if there is an `Attribute` element with the same name as the second element of the path in the case settings. If there is a matching attribute, the value is semantically valid. Otherwise, throw an error.
  4. Check whether the left-most element is found in the list of stages of the case and throw an error if it is not found, else continue to step 5
  5. If the `Task` identifier, the second element in the list, is not found in the list of `Tasks` of the `Stage` element found in step 2, then throw an error, else continue to step 6
  6. If the `Field` identifier, the third element in the list, is not found in the list of `OutputFields` and `InputFields` of the `Task` element found in step 5, then throw an error

  The values of `previousStep` and `condition` are assumed to be semantically valid, if there are no errors thrown in any of these steps listed above. Example semantic error messages for semantically invalid `previousStep` and `condition` statements:

  - A non-existing element "Identifications" is referenced as the value of `previousStep` attribute

    ```
    Precondition
        previousStep = 'Identifications'
    ```

    ```
    Semantic Error at line 100! Stage 'Identifications' does not exist!
    ```

- The path in the value of a `condition` attribute does not point to an existing element as the task name is missing.

```
Precondition
     previousStep = 'Evaluation'
     condition = 'Evaluation.RequestMedicalTest = 1'
```

```
Semantic Error at line 175! Invalid precondition path
'Evaluation.RequestMedicalTest=1'. The path does not point to an
existing element. Make sure your path follows one of these rules:
 1. <StageName>.<TaskName>.<FieldName>
 2. Setting.<AttributeName>
```

- **OutputField and InputField attributes**
  As mentioned before, the value of the `expression` attribute of the `OutputFields` can be a `round` expression or a conditional `if-else` expression. The value of the `expression` attribute is syntactically validated using individual grammars. However, the variables within these expressions also have to be semantically validated by checking if they point to a valid and existing element. For instance, for the expression `'round(Weight / (Height * Height))'` to be semantically valid, the "Weight" and "Height" should be the names of existing `OutputField` or `InputFields`. For this reason, the `expression` attribute of every `OutputField` in the case definition is validated with the following steps:

  1. Extract all of the potential field names using a regular expression from the expression value and store them in a list
  2. For each of the extracted field names in the list, check if there is a matching field with the same name. If there is a matching field, continue to the next field name in the list. Otherwise, throw an error.

  An example semantic error and its corresponding error message is as follows:

```
expression = 'if (Diastolic < 80 and Systollic < 120) then "Normal"
```

```
Semantic Error at line 141! Invalid field Systollic found in
the expression of OutputField "OverallAssessment". The field
does not exist.
```

  Furthermore, the value of the `CustomFieldValue` attribute of an `OutputField` or an `InputField` is expected to contain the path of the element storing the value of its output, for instance an `Attribute` element from the case `Settings`. Therefore, this attribute is also semantically validated using the same algorithm that is used for semantic validation of `Precondition` attributes. An example semantic error and its corresponding error message is as follows:

```
OutputField OverallAssessment
            #left #custom
            CustomFieldValue = "Setting.PressureCondition"
```

```
Semantic Error at line 139! 'PressureCondition' not found in
the Case Settings. Invalid reference.
```

- **Summary Panel InfoPath**
  The `InfoPath` attribute of a `SummaryPanel` element defines a path to an `InputField` or `OutputField`. The algorithm that validates the `InfoPath` elements of `SummaryPanels` follows the same logic mentioned before except for a minor difference due to the constraint where the path needs to include exactly three elements with the following pattern: `<Stage>.<Task>.<Field>`

  The following steps are repeated for each `InfoPath` element that belongs to the `SummaryPanel` of the case definition:

  1. Split the dot-separated path by using dot as a separator into a list and check the length of the list. If the length is less than 3 then throw an error, else continue to step 2
  2. Check the left-most element, which is the first element of the list and if a matching `Stage` element is not found in the list of stages of the case, then throw an error, else continue to step 3
  3. If the task identifier which is the second element of the list, is not found in the list of `Tasks` of the `Stage` element found in step 2, then throw an error, else continue to step 4
  4. If the field identifier, the third element of the list, is not found in the list of `OutputFields` and `InputFields` of the `Task` element found in step 3, then throw an error

  Example error messages:

  - A task with the name, "MeasureBlooPressure" does not exist:

    ```
    InfoPath Evaluation.MeasureBlooPressure.Systolic
    ```

    ```
    Semantic Error at line 56! 'MeasureBlooPressure' not found in the
    tasks of Stage Evaluation. Invalid info path.
    ```

  - The Field name is missing from the path:

    ```
    InfoPath Evaluation.MeasureBloodPressure
    ```

    ```
    Semantic Error at line 58! Invalid info path
    'Evaluation.MeasureBloodPressure'.The path does not point to an
    existing element. Make sure your path follows the following rule:
       <StageName>.<TaskName>.<FieldName>
    ```

7.2.3. Trusted URL Validation

In the Acadela language, one of the elements that are being used to model CPs is the `Hook` element that enables process orchestration across system boundaries after state change events, such as activate, enable, complete, terminate. A `Task` element or a `Case` element can have hook definitions. To define a `Task hook`, a task state change event, a URL, and an HTTP method, such as POST, GET, PUT, or DELETE must be specified. Optionally, a failure message can be declared. For a `Case` hook, a URL and a state change event must be specified. An example `Task hook` is defined as follows:

```
Trigger
    On complete
    invoke 'https://server1.com/api2'
    method POST
    with failureMessage 'Cannot complete the data creation!'
```

The URLs should be among the predefined list of "trusted" URLs of the workspace of a case definition to have semantically correct hooks. Furthermore, the HTTP method should be among the allowed methods of the specified trusted URL. Therefore a module for semantically validating hooks is implemented as part of the semantic error validator. It is crucial to note that, how and where to store the list of trusted URLs and corresponding allowed HTTP methods is not yet decided for the new system. For this reason for the sake of this thesis it is assumed that this information is stored in a CSV file in the format displayed in Table 7.1:

Table 7.1.: An example trusted URLs file

Workspace Name	URL	Methods
Umcg	http://127.0.0.1:3001/connecare	POST
Umcg	https://server1.com/api2	POST , GET
Umcg	http://integration-producer:8081/v1/activate	
Umcg	localhost:3001/connecare	

To validate the hooks, as initial steps, the `Task` and `Case` elements are accessed via the `CaseObjectTree` and the CSV file where the list of trusted URLs are stored is opened and the rows are read using a CSV reader library.

As mentioned before, the semantic validation of `hook` elements involves:

- Check whether the specified URL by the modeler is among the trusted URLs of the workspace that the case definition belongs to
- If the URL is a trusted URL, then check whether the specified HTTP method is in the list of allowed methods of the URL provided.

To check these two constraints an algorithm is implemented with the following steps:

1. For each `Task` object in the `tasks` list from the `CaseObjectTree`, check if the `Task` object has a `hookList` attribute with a length greater than 0

2. If the `Task` has a `hookList`, for each `hook` object in the list, iterate through the rows of the CSV file until a row with matching `workspaceName` and URL is found.

3. If a matching row is not found, throw an error as the URL is not trusted. Else, check if the HTTP method of the `hook` is in the list of allowed methods of the matching row.

4. If the HTTP method is in the allowed methods list, move on to the next `hook` element in the `hookList`. Else, throw an error, as the trusted URL does not accept the HTTP method of the hook object

The algorithm described above is also applied to the `Case` hooks with slight modifications such as the HTTP method check is excluded as `Case` hooks does not have specified HTTP methods.

An example error message for a case where the URL defined by the user is not found in the list of trusted URLs:

```
Trigger
        On complete invoke 'https://server3.com/api2'
        method POST
        with failureMessage 'Cannot complete the data creation!'
```

```
The URL https://server3.com/api2 at line 146 and column 21 is not in the
list of trusted sources for workspace Umcg. Please check the trusted
sources list for the permitted URLs
```

An example error message for a case where the URL defined by the user is found in the list of trusted URLs but the HTTP method defined by the user is not found within the allowed methods of this URL:

```
Trigger
    On complete invoke 'http://127.0.0.1:3001/connecare'
    method DELETE
```

```
The URL http://127.0.0.1:3001/connecare at line 145 and column 21
does not accept the HTTP method DELETE. Allowed methods: POST.
Please further check the trusted sources list for the permitted methods
```

7.3. Integration to the Existing System

After the implementation of the syntax and semantic error validators were completed, the error handler was integrated into the system as the second phase of the implementation. For this purpose, Acadela IDE and Acadela Backend projects were updated.

7.3.1. Integration to Acadela IDE

The Acadela IDE initially included only a code editor and a "Submit" button. The "Submit" button's function was to solely send the Acadela source code that the user implemented to an endpoint at the Acadela Backend where the code is compiled and parsed when a request is received.

However, the user was not able to see if their code was compiled successfully or failed due to syntax errors. Consequently, the user was not able to see any error or success messages like most commercial IDEs. For this reason, several new components and new functionality were added to the IDE to retrieve and display success and error messages from the backend.

- **Validate and Submit Functionalities**
  As mentioned earlier, after the Acadela code is compiled, the code is parsed by creating an object tree of the model from the code and it is semantically validated. During the parsing process, if the code turns out to be semantically and syntactically valid and if the interpreter is instructed to do so, the model is sent to SocioCortex in JSON format after some preprocessing is done to the `caseObjectTree`. Whether to run the network operations which include sending the compiled and parsed case to the SACM and stored in SocioCortex is controlled with a flag `CONN_SocioCortex` and its value was initially stored in a configuration file within the backend code. Consequently, to change this value, the configuration file had to be edited which is not possible for end-users to perform themselves as they do not have access to the code. Even though the sole objective of modeling CPs using Acadela is to create case instances within SocioCortex and SACM, keeping this value constantly true means running all of the network operations every time a user compiles their code, which was not convenient for multiple reasons. First of all, the whole process of running all of the network operations is comparably lengthy compared to just validating the code. Therefore, it was decided that it is not user-friendly to make the user wait each time they compile their model if their model is not finalized and ready to submit to the system. Secondly, creating a case item and storing it in SocioCortex from an incomplete but syntactically and semantically correct model pose some issues, for instance, as the version of cases should be unique, the user has to change the case version each time they submit their code even if their code is incomplete and they just want to validate their code. For these reasons, it is decided to add new functionality to the system which allows users to choose whether to run the network operations or not. Initially, only

the Acadela source code that the user inputs into the code editor component were being sent within the body of the HTTP POST request to the endpoint. To prevent users from always creating a case in SocioCortex even if they just want to validate their code, an additional new value is added to the request body with the key `connect`. This new key can have a boolean value, true or false, and the value of this key is used to control whether the network operations should be performed if the code successfully compiles or not. Therefore, if the user only wants to "validate" their code without attempting to create a case instance in SACM, the value of the connect key is set to false and if the user desires to "submit" their code, which means "connecting" to SACM and completing the network operations, the value is set to true. To realize this difference in the Acadela IDE, an additional "Validate" button is added next to the "Submit" button. If the user clicks on the "Validate" button, the value of the key `connect` is set to false and the request is sent to the endpoint, and if the user clicks on the "Submit" button this key is set to true.

- **Error and Success Messages**
  As mentioned before, the initial version of Acadela IDE was not designed to receive and show the response from Acadela Backend after compilation, hence the user was not able to know whether their code is successfully compiled or failed to compile due to the presence of syntax or semantic errors. To display the result of the compilation of their code to the user, first, the returned response from the backend has to be extracted from the HTTP response returned from the endpoint in the Acadela Backend. After making necessary changes to the Acadela Backend which is explained in detail in the next section, the potential HTTP responses returned from the endpoint include HTTP codes `201` in case of a successful compilation and `213` in case of syntax or semantic errors.

  In the case of a response with the HTTP code `213`, the value of `traceback` key of the data returned within the HTTP response includes the error message produced from the Acadela compiler. The error message is extracted from this value and the instances of the ASCII code of newline `\n` are replaced with the corresponding HTML tag `<br\>`. Finally, the extracted and processed HTML code segment is displayed in a text container in the color red as shown in Figure 7.2. Similarly, in the case of a response with the status code `201`, a success message, *"Successfully Compiled!"* or *"Successfully loaded the case into SACM!"*, is displayed in the color green as shown in Figure 7.1.

- **Loading Message**
  Last but not least, to enhance the user-friendliness of Acadela IDE, it is decided to display a message, *"Loading..."* to the user after they send a request to the compile endpoint via clicking on "Submit" or "Validate" buttons until a response with error or success messages is returned from the endpoint. The loading message can be seen in Figure 7.3.

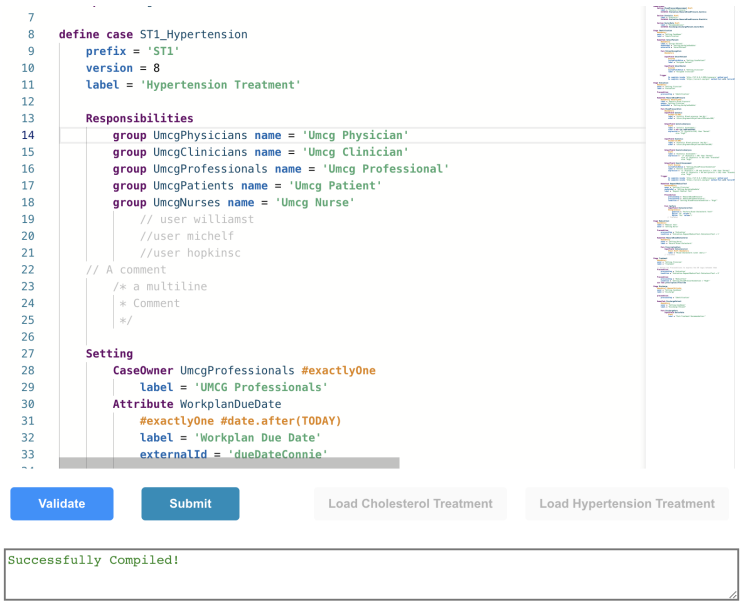

Figure 7.1.: "Successfully Compiled!" message displayed in the text container

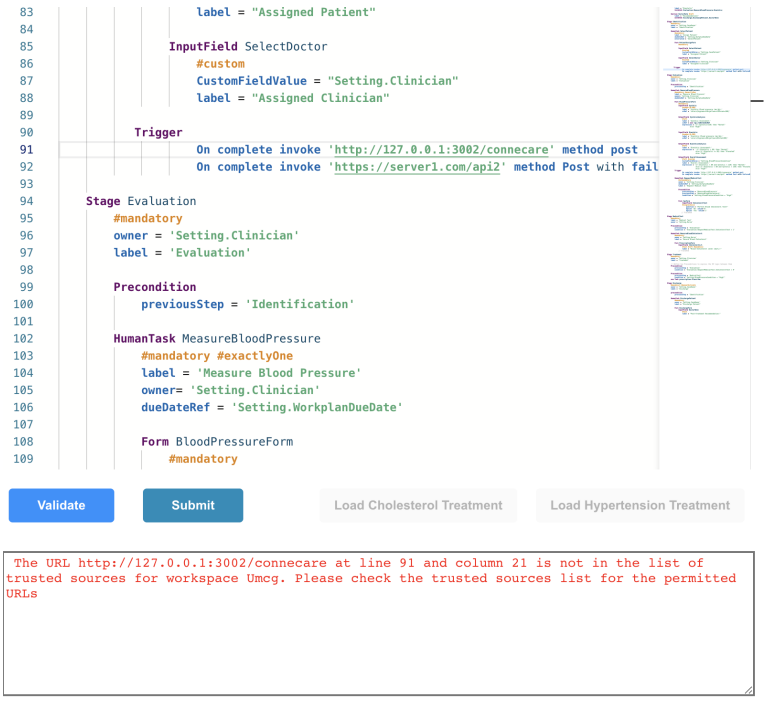

Figure 7.2.: The error message returned from Acadela Backend displayed in the text container

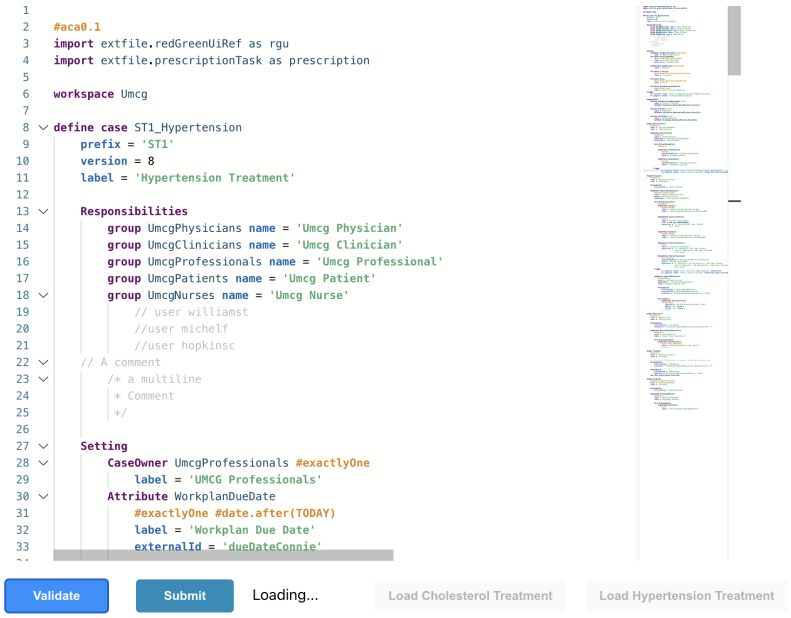

Figure 7.3.: Display loading message after sending the request and before receiving the response

7.3.2. Integration to the Acadela Backend

To make the Acadela Backend compatible with the changes and new features from the Acadela IDE, several small changes are made to the function that runs when a request is sent to the `/compile` endpoint of the Acadela Backend. As mentioned before, initially, whether to run the network operations which include sending the validated CP to SACM or not is controlled with a flag `CONN_SocioCortex` stored in a configuration file. To integrate the changes, the `compile` function is modified to get the value of this flag from the received request instead and pass this value to the Acadela Compiler.

The second change made to the Acadela Backend is to change how the HTTP response is returned from the endpoint. Initially, the `compile` function which is triggered when a request is sent to the compile endpoint of the server was returning a response with status code `201` that includes the message, *"Success"* and it was not handling the cases such as if an error is caught by the Acadela compiler. The message is kept as it is as in case of a successful compilation there is no useful data to include in the response. Unlike the case of a successful compilation, to return the syntax and semantic errors, the `compile` function is modified. After the modifications, the function now returns a response with the HTTP status code `213` that includes the error message in JSON format in the body.

8. Evaluation

This chapter describes the evaluation of the usability and accuracy of the syntax and semantic errors produced by the implemented error handler in the scope of this thesis as well as the overall usability of the Acadela language. It describes the evaluation approach and its quantitative and qualitative results, hence providing answers to the third and last research question: *What is the opinion of modelers regarding the usability and accuracy of the error validator?*

8.1. Evaluation Approach

In the scope of this thesis, the evaluation stage has two goals. The first goal is to evaluate the usability and accuracy of the newly implemented error validator, whereas the second goal is to evaluate the overall usability of the Acadela Language as the new error validator also has an impact on it.

It is important to note that during the user studies conducted, the syntax of Acadela Language is also evaluated by the users. However, as the syntax of Acadela is not directly within the scope of this thesis, the approach and results of this part are excluded from this chapter.

To evaluate the usability of the Acadela language and the syntax and semantic validator, it is decided to conduct user studies with potential users of Acadela as part of the evaluation of this work.

8.1.1. User Studies

The user study designed for participants to complete consisted of two parts: a 50-minute training session and afterward an online experiment session. The training session which is the first part of the designed study was included to give the participants a chance to get familiar with the language as they did not have any previous experience with it. In this session, initially, the participants were introduced to the Acadela language with a presentation that briefly explains the motivation of Acadela and conceptually introduces the elements of the CPs that can be modeled using Acadela. After this short presentation, the participants were introduced to the Acadela Wiki[1], a wiki page that includes the documentation of the Acadela DSL. Together with the participants, we briefly went through the Acadela syntax and element definitions to give them an introduction to how to model CPs using Acadela. Finally, the participants were asked to model some basic

[1] Acadela Wiki Page

elements themselves interactively to give them a chance to practice before they move on to the actual tasks they need to complete in part of the evaluation. In the second part of the user study which is an online experiment session, the participants were expected to complete two tasks:

- Solving a small modeling task by completing an existing CP model with missing elements
- Fixing the errors in another existing model by reading the clues from their error message and correcting the mistakes

After these two tasks were completed by the participants, they were expected to fill out a questionnaire to evaluate the usability of the Acadela language with a focus on the error messages.

8.1.2. Participants

The objective of the evaluation stage of this work was to get actual user feedback from potential users of Acadela on the usability of Acadela as well as the new error messages. For this reason, the participants of the user studies were selected among the professionals working in the medical field, preferably familiar with CPs and having some degree of modeling experience. The user studies were conducted with five participants all working in the medical field as medical doctors, researchers, research assistants, or technicians. As a part of the evaluation questionnaire, the participants were asked to provide their age, years of experience in the technical field, the programming languages they are familiar with and the modeling languages they have experience with to have a better understanding of the impact of technical experience on the perceived usability of Acadela. As can be seen in Table 8.1, almost all of the participants had some experience with at least one programming language and a modeling language. However, as none of the participants were familiar with Acadela, they still needed to be introduced to the language and trained.

Table 8.1.: Subjects participated to the user studies

No.	Age	Occupation	YOE[1]	Prog. Languages[2]	Modelling Languages
1	36-45	Research Assistant	15	C, C++, Java, JS, Python, R	XML, UML, BPMN, CMMN, SQL
2	36-45	Researcher	2	PlantSimulation	-
3	26-35	Research Assistant	1	C#, Python	BPMN
4	46+	Medical Doctor	20	Basic, Pascal	SQL
5	26-35	Technician	9	Java, Python	XML, JSON, SQL

[1] Years of Experience
[2] Programming Languages

8.2. Evaluation of the Acadela Language

After the participants completed the tasks and experienced modeling with the Acadela language they evaluated its usability. The evaluation questionnaire included a section for this with 10 statements and a standardized scale of answers to assess the usability.

For evaluating the usability of the language, it is decided to use System Usability Scale (SUS). SUS is a scale system developed by John Brooke. SUS is similar to a Likert scale where the respondents are presented with statements and expected to indicate their degree of agreement or disagreement with the statement with a 5 point scale. SUS includes ten statements that are designed to assess the effectiveness, efficiency, and satisfaction of users while using the subject system. The statements cover many aspects of system usability such as complexity, need for support, user-friendliness, and need for training. The respondents are expected to respond with a five-point scale that ranges from Strongly Disagree to Strongly Agree [23]. SUS system is originally designed for evaluating the usability of a system as a whole. However, the goal of this evaluation is to evaluate only the usability of the Acadela language and not the whole system which also includes for instance the Acadela IDE. Therefore, the wording of the original statements of SUS was slightly modified by replacing the keyword "system" with "language" to avoid confusion. The original SUS questions can be found in Addenda A.1. The adapted statements are as follows:

1. I think that I would like to use this language frequently.
2. I found this language unnecessarily complex.
3. I think that this language is easy to use.
4. I think that I would need assistance to be able to use this language.
5. I found the various functions such as defining elements, importing modules, and error validations in this language were well integrated.
6. I thought there was too much inconsistency in this language.
7. I would imagine that most people would learn to use this language very quickly.
8. I found this language very cumbersome to use.
9. I felt very confident using this language.
10. I needed to learn a lot of things before I could get going with this language.

The SUS score has a range of 0 to 100 and is calculated based on the participants' answers using the following method described by John Brooke: *"To calculate the SUS score, first sum the score contributions from each item. Each item's score contribution will range from 0 to 4. For items 1,3,5,7, and 9 the score contribution is the scale position minus 1. For items 2,4,6,8 and 10, the contribution is 5 minus the scale position. Multiply the sum of the scores by*

2.5 to obtain the overall value of SU" [23]. The calculated system usability score of each participant can be found in the Table 8.2. The average SUS score is calculated as **75.5** for the Acadela Language where a SUS score above 68 is considered "above average".

Table 8.2.: System Usability Score results

No.	Profession	System Usability Score
1	Research Assistant	62,5
2	Researcher	55
3	Research Assistant	87,5
4	Medical Doctor	95
5	Technician	77,5
Average Score		75.5

According to the other metrics which can be seen in Figure 8.1, this score is in the "Acceptable" range, has a grade scale score of "C", and it has the score "Good" according to the adjective grade scale [24].

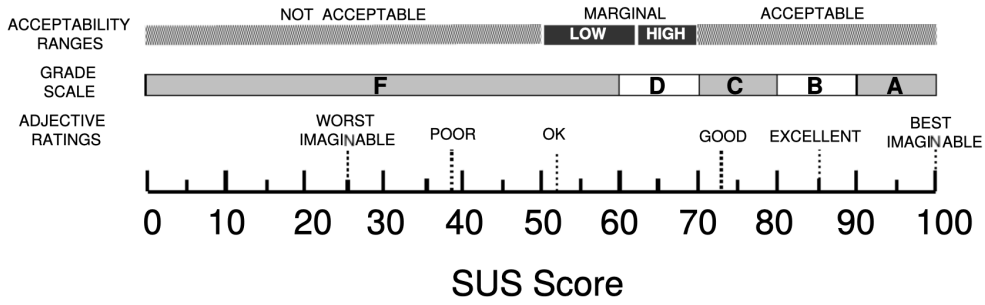

Figure 8.1.: A comparison of SUS scores with acceptability ranges, grade scale, and adjective ratings by Bangor[24]

To ensure the mean is representative and reliable for assessing the usability, the standard deviation is calculates using the following formula: $s = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \bar{x})^2}$ where N is 5.
The standard deviation is calculated as **14.94**. As the average SUS scores are found to be 68 with a standard deviation of **17.5**, the SUS score of the Acadela language is above average and the standard deviation is aligned with the average value and within the desired range required to be considered sufficient[24].

8.2.1. Result Analysis

As mentioned before, the SUS score of the Acadela language is 75.5 which is above the average cumulative value of SUS scores which is 68. When the individual SUS scores of participants were analyzed, it can be seen that except for one score which is 55 and

a relatively higher but still below the average score of 62.5, the SUS scores are above average, rather on the higher end of the scale and considered acceptable. Furthermore, the SUS score calculated from the answers of the participant who is a medical doctor is significantly high. This result is very positive as most of the times clinicians do not prefer to model CPs or feel overwhelmed by them. Even though the sample size is small, it can be argued that the participants who have very varying degrees of technical experience, found the system considerably usable. The lowest SUS score is 55 which belongs to the only participant with no previous modeling experience as well as very small programming language knowledge compared to other participants. Based on this fact, one can argue that the language still requires at least some basic programming and modeling knowledge to be used effectively which is a very much expected result.

8.3. Evaluation of the Error Validator

In order to evaluate the usability of the error messages, an additional task is designed and added to the user study. The subjects were already somehow familiar with the error messages as they mostly had to debug their models from the previous tasks of the user study. However, to make sure that they have the chance to have experience with all of the error types and their corresponding error messages before they evaluate the usability and accuracy of the error messages, a blood cholesterol CP with intentional syntax and semantic errors is created. To complete this task, they were asked to fix the errors one by one and after solving each error, to validate the model and continue until they see the *"Successfully compiled!"* message displayed. The participants were aware that the CP provided included errors however they did not know what were the errors and how to solve them.

After they complete this task by fixing all of the errors, the participants were asked to complete two sections of the evaluation questionnaire which include multiple statements that they have to answer on a scale of 1 to 5 where the value 1 refers to strongly disagree and 5 refers to strongly agree.

The first section consists of 7 statements each addressing specifically a feature or an error type that the validator covers. The statements are as follows:

1. The error messages helped me to fix unexpected attributes
2. The error messages helped me to fix the errors in conditional statements
3. The error messages for the wrong directive helped me to fix the errors
4. The error messages helped me fix typos
5. The error messages helped me to locate and correct the paths to nonexistent elements
6. The error messages helped me to fix duplicate element name

7. The error messages helped me to fix the errors related to URLs and HTTP methods

The statements are designed to effectiveness of the error messages in terms of fixing the corresponding error types. The distribution of the participants' answers to these statements can be seen in the scale bar graph in Figure 8.2.

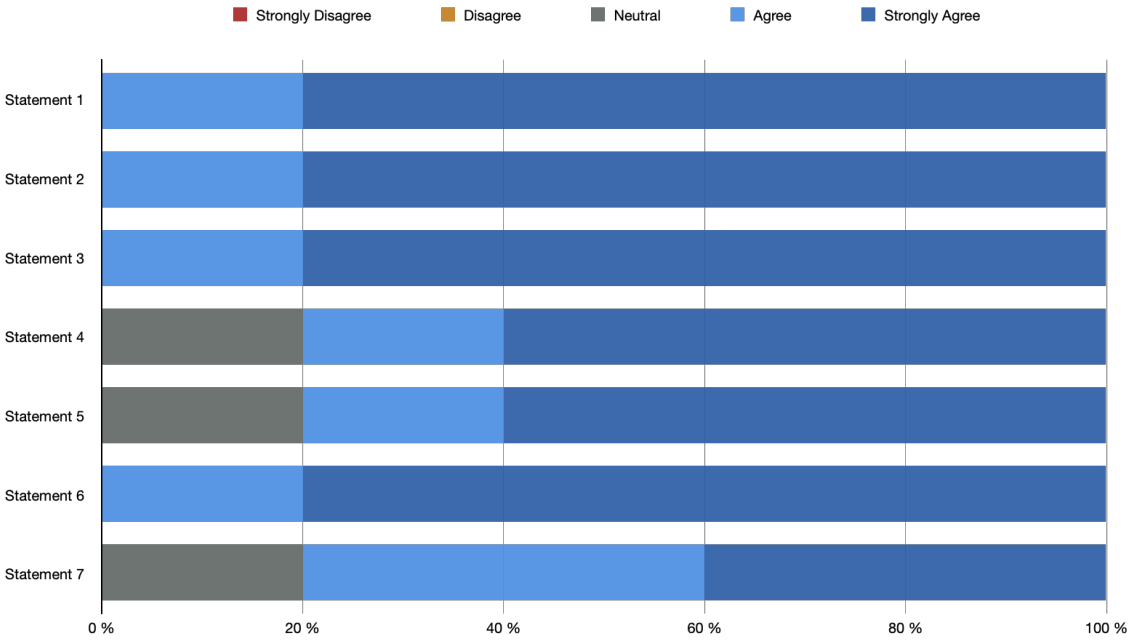

Figure 8.2.: Results of the first section of the evaluation the error messages

The second section consists of 6 statements all designed to evaluate the overall efficiency, usability, and accuracy of the error messages. The statements are as follows:

1. The error messages were easy-to-understand
2. I was easily able to locate the source of error using the error messages
3. I was able to fix the errors easily using the error messages
4. The language of the error messages is clear and precise
5. The error messages were accurate
6. The error messages were consistent

The statements in this section focused on the overall perceived readability and usability of all of the error messages. The distribution of the participants' answers to these statements from the second section of the questionnaire can be seen in the scale bar graph in Figure 8.3.

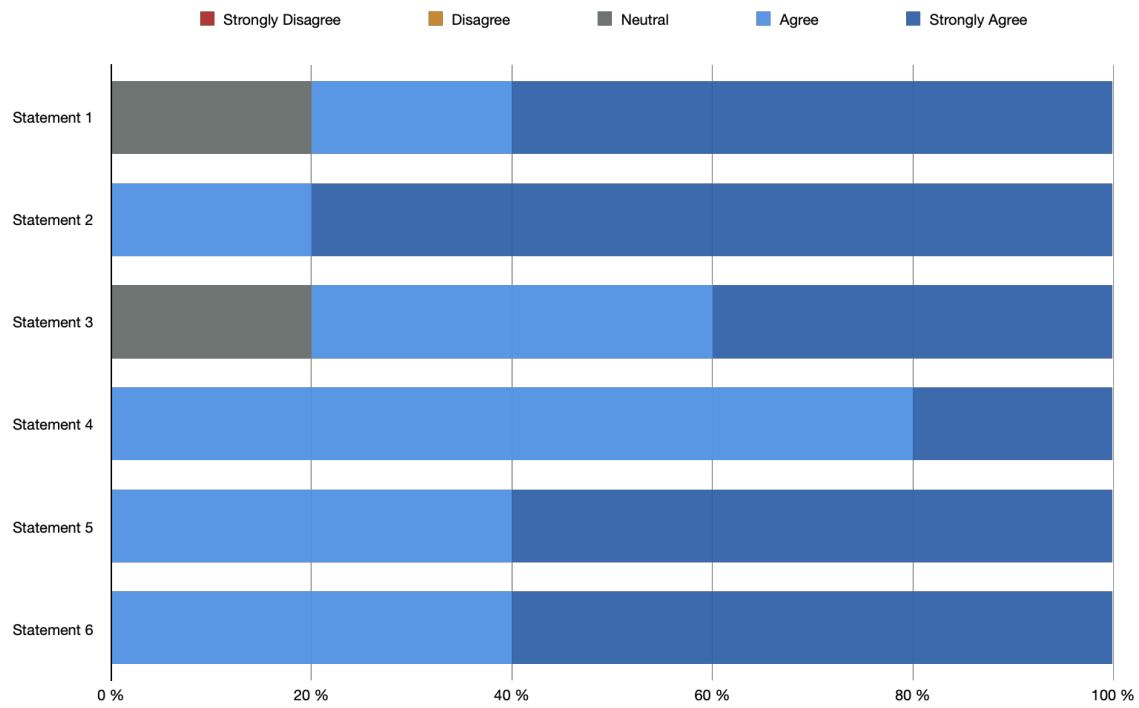

Figure 8.3.: Results of the second section of the evaluation the error messages

8.3.1. Result Analysis

As it can be observed from the graph displayed in Figure 8.2, the feedback of the participants in terms of the effectiveness of each error message specific to each error type covered by the error validator is perceptibly positive with an exception of a few neutral answers. The participants did not disagree with any of the statements so it can be said that the error messages were helpful to users to locate and fix the errors consistently except to a varying degree for a few of them. The results for statements 1, 2, 3, and 6 show that the error types that these statements refer to are handled successfully and the error messages produced for each type are helpful enough for users to fix the corresponding errors. Even though almost all participants(%80), agreed with statements 4 and 5, there is one participant who answered these statements with the option neutral. This shows that the error messages for typos and the error messages for invalid references can be improved. According to the results of the evaluation, the error messages for trusted URLs evaluated by statement 7 received relatively less positive feedback from the participants. Despite the %80 of participants agreeing that these messages were helpful, the number of participants who strongly agree is less than the other statements and one participant was neutral. However, this result is understandable as the participants did not have an access to the trusted URLs lists.
The results of the second section which is designed for evaluating the non-functional requirements of the error messages such as their usability and efficiency are also predominantly positive as can be seen in the Figure 8.3. All of the participants agreed

that they were able to locate the source of the errors using the error messages which is the primary purpose of error messages. They also agreed that the error messages were accurate and consistent. Furthermore, all of the participants somehow agreed that the language of error messages was clear and precise. On the other hand, even though the majority of participants, more specifically 80% of them, agreed with the statement "the error messages were easy-to-understand", one participant answered this statement with the option neutral. This shows that the understandability of the error messages has room for improvement as understanding error messages still require some level of technical knowledge. Similarly, while 80% of the participants agreed that they were able to fix the errors using the error messages, again one participant picked the neutral option. As the primary goal of error messages is to assist users to fix errors, this feedback should be taken into careful consideration.

9. Discussion

In this chapter, the limitations of this work are described along with some reflections and suggestions. Furthermore, the internal and external validity of the system is discussed.

9.1. Limitations

In the scope of this thesis, there were some limitations in the means of the research, the existing system, as well as the limitations of the implemented system, and the evaluation.

9.1.1. Limitations of Acadela and textX

It is important to explain the limitations of the Acadela Language, mainly its grammar, and the textX tool as these limitations carry some notable impact on the implemented system.

textX documentation: As textX is relatively a new tool and is yet to be widely used in comparison to other metalanguages such as Xtext, it was challenging to find enough sources and documentation to design and implement the error validator. Furthermore, there was no related work available that comprehensively explains the error validation for languages developed with textX. Even though studies about approaches for constraint validations using different metalanguages were available and they were somehow helpful to at least give some conceptual insight, their impact was limited. This is mainly because the features and implementations of these tools and metalanguages despite having some similarities differ vastly and in the end, the implementation of the constraint validator is bound to the tool being used offers.

White-space sensitivity: The parser of textX is not white-space sensitive by default and it does not support white-space sensitive grammar development yet. Therefore the grammar of Acadela is not white-space sensitive which means the parser skips occurrences of the white spaces in the source code. This limitation of textX and consequently Acadela limited the flexibility and accuracy of the error messages. To mitigate this limitation to some extent, some of the keywords in the Acadela grammar are explicitly padded with the white-space character,`\s`, however, this was not viable for a small minority of keywords mostly due to other constraints. It can be argued that this solution improved this limitation

substantially however it was rather a workaround. In the future, realizing a more reliable and comprehensive solution or introduction of the white-space sensitivity feature to textX can improve the quality of both the grammar as well as the error messages.

Problems with the Acadela Grammar: Some limitations and fallacies within the Acadela grammar cause the parser to assess models which are supposed to be syntactically valid as invalid and to unexpectedly report syntax errors. For this reason, the validation of string patterns decided to be done with individual textX grammars instead of extending the Acadela grammar itself. This decision was made because the likelihood of this happening increases as the rules for attributes in the grammar gets more complex. Consequently, as mentioned earlier, these string patterns are being validated after compilation and during interpretation which is an anti-pattern as these validations are syntactical. In the future, the grammar can be revised and improved, and eventually, these individual grammars can be integrated into the original grammar.

9.1.2. Limitations of the Proposed Solution

Coverage: The main limitation of the implemented system is that despite the effort to cover as many as the potential errors that a user can potentially make and the edge-cases where textX fails to locate errors, it is virtually impossible to foresee and cover all of them as syntax errors are usually quite arbitrary and not foreseeable. This limitation has the most impact on giving accurate cause and potential suggestions for errors within the new syntax error messages.

Accuracy vs. Specificity trade-off: The features that are designed and implemented to enhance the user-friendliness of the syntax error messages have some drawbacks. It is observed during the tests and the evaluations that on occasion these customizations make the error messages relatively confusing and sometimes misleading if the error is an edge-case that is not foreseen beforehand and consequently the cause of the error is not assessed correctly. However, this was an expected limitation that creates an accuracy vs. specificity trade-off where the more specific the error messages the less accurate the error message if the cause of the error is assessed incorrectly. Even though customizing and extending the content of syntax error messages substantially enhanced the usability and made them more user-friendly, the implemented error handler can only "speculate" about the cause of the errors and if it is speculated incorrectly then these features, as a matter of fact, can increase the debugging effort.

HTTP hooks: Lastly, as mentioned before, it is not yet decided where and in what format the data on trusted URLs and corresponding allowed HTTP methods of workspaces is going to be stored in the new system. For the sake of completeness of this thesis, it is assumed that this information will be stored in a CSV file stored

within the project itself. However, it is very unlikely that this data will be stored within the codebase even if it is stored in CSV format. Hence, this limitation requires some changes to the implementation in the future when this is decided.

9.1.3. Limitations of the Evaluation

Sample Size: The first and foremost limitation of the evaluation phase of this work is the sample size. Throughout the evaluation period, we were able to contact only five participants which is the minimum number of required participants determined at the beginning of this work. The reason for this limitation was that the potential participants that can participate in the user studies and evaluate the system are expected to be from the medical field, preferably familiar with clinical pathways or have some degree of modeling experience. These requirements considerably narrowed down the eligible participants for the user study as they are very specific and it was challenging to find matching candidates and contact them.

Duration: The second limitation of the evaluation was the duration of the user studies. It is decided to conduct the user studies with subjects for at most one and a half hours which already requires a considerable amount of time and effort from the participant's end. Furthermore, more than half of this period had to be spent on training the users about Acadela which was mandatory as the users have to at least have some basic knowledge of the language in order to be able to complete the evaluation. As the evaluation part of the user studies also required a substantial amount of effort and time from the participant, the evaluation tasks needed to be as brief as possible. Hence, the required effort and time from the participants' end limited the desired comprehensiveness of the evaluation.

Evaluating Error messages: Lastly, evaluating the usability and accuracy of the error messages was a challenge. As described in the previous chapter, for evaluating the error messages, the user was expected to debug an existing CP with pre-made syntax and semantic errors. This approach was decided to make sure that each participant sees and evaluates every feature of the new error handler while it is not guaranteed for a user to naturally produce each type of error while they are using the system. However, this approach had its limitations. As mentioned before, the errors were not made by the participant while modeling even though this is the primary use case of the error messages. Therefore, the evaluation of error messages was limited and if the participant did not make a lot of errors while completing the first part of the tasks, they did not experience this main use case and utilize the error messages enough. A more comprehensive and "natural" evaluation approach where the modelers in fact make the errors themselves while they are modeling can provide better insight into the effectiveness of error messages.

9.2. Validity of the Implemented System

9.2.1. Internal Validity

The biggest factor that affects the internal validity of the implemented system is the limitation on testing. Throughout the implementation, only a few simple CP models were used for testing purposes. Consequently, the coverage and validity of the error messages were estimated by only a small number of simple models. The validity and accuracy of the error messages therefore can be different for other models, especially for the models of more complex CPs which may include different paths of executions.

9.2.2. External Validity

The system is designed and implemented specifically for the Acadela language based on the constraints of the CPs modeled in CONNECARE and the requirements of the CONNECARE system. However, these constraints and requirements are not universal for all CPs that exist in different systems and settings. Models of CPs can include arbitrary elements and constraints. For this reason, it is not possible to know the external validity of the solution as it is not yet applied to any other system but the implemented system likely has limitations for being used in different e-health domains. The Acadela language as well as the error validator should be extended beforehand to be used for modeling different CPs that exist in other domains than CONNECARE.

10. Conclusion

This chapter summarizes the objectives, the research, and results of this thesis. It also presents suggestions for future work.

10.1. Summary

The goal of this thesis was to develop a syntax and semantic error validator for the domain-specific language Acadela which aims to replace XML as the modeling language used for modeling the clinical pathways in the CONNECARE system. The error validator was aimed to produce user-friendly and accurate error messages that will eventually help reduce the learning and debugging effort of the Acadela language.
This goal was accomplished throughout this thesis through three stages, namely, the preliminary planning and research stage, the implementation stage, and lastly the evaluation stage. Each of these stages individually provided answers to each of the identified research questions.
In order to answer the first research question, *What syntactic or semantic errors can be validated by the DSL during the modeling of clinical pathways?*, a preliminary research and planning phase was conducted. During this stage of the thesis, the semantic constraints of the Acadela Language that needs to be validated by the semantic validator to ensure logically coherent models were defined. The implementation of this semantic validator which was required to successfully validate these defined constraints was also planned in this stage. Similarly, the requirements and constraints to improve the default syntax error messages of textX were identified and the implementation of the syntax error validator was planned. Last but not least, the requirements for the error messages produced by both of the validators were discussed and specified.
After the preliminary research and planning stage was complete, the focus was to implement a user-friendly error validator based on the findings and results of the first stage. For this purpose, the implementation stage has started with implementing a syntax error handler that has features such as detecting potential typographical errors and simplifying the language of the error messages. Once the implementation of the syntax error handler was completed, the implementation of the semantic constraint validator and error handler started. Modules for each semantic rule identified in the preliminary stage were implemented separately. Furthermore, the new individual grammars for the string patterns were constructed. After the implementation of the semantic and syntax error validators was completed, they were integrated into the Acadela IDE and the Acadela backend so that this stage was finalized. The implementation stage of this

work and its results, therefore, answers the second research question: *How to identify and interpret the errors to modelers in an accurate, user-friendly manner?*
Once the technical implementation of the error validator was completed and it was integrated into the existing system, the usability and user-friendliness of the new error validator were evaluated by the potential users of the Acadela Language as the third stage of this thesis. Therefore, the results of the evaluation phase provide answers to the third and last research question: *What is the opinion of modelers regarding the usability and accuracy of the error validator?*

10.2. Future Work

In order to improve the results of this thesis with future work, it is critical to first and foremost mitigate the limitations that are already described and discussed thoroughly in the chapter 9 with potential solutions and improvements.

Nevertheless, it is important to point out that only more user feedback can substantially improve the usability, accuracy, and efficiency of the implemented system and the new error messages. More testing and regular usage of the Acadela language can reveal new error types that the users are likely to make which are not yet covered by the current system. Dealing with those error types can significantly increase the coverage of the error handler and vastly improve the accuracy of the error messages. Additionally, further user studies can be conducted with larger sample size, and the user feedback from these evaluations can be used to improve the overall quality and usability of error messages.

Lastly, after the issues with the Acadela grammar are fixed, the individual grammars of string patterns can be integrated into the Acadela grammar for consistency.

A. General Addenda

A.1. Original SUS questions

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

List of Figures

List of Tables

Acronyms

Acadela Adaptive CAse DEfinition LAnguage.

AST Abstract Syntax Tree.

CCP Complex Chronic Patient.

CMMN Case Management Model and Notation.

CONNECARE Personalised Connected Care for Complex Chronic Patients.

CP Clinical Pathway.

CSV Comma-Separated Values.

DSL Domain Specific Language.

GPP General-Purpose (programming) Language.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

IDE Integrated Development Environment.

JSON JavaScript Object Notation.

SACM Smart Adaptive Case Management System.

UI User Interface.

XML Extensible Markup Language.

Bibliography

[1] E. Vargiu. "Personalised Connected Care for Complex Chronic Patients Results from the Connecare Project". In: *International Journal of Integrated Care* (2020).

[2] M. Yulianto, R. Arifudin, and A. Alamsyah. "Autocomplete and Spell Checking Levenshtein Distance Algorithm To Getting Text Suggest Error Data Searching In Library". In: *Scientific Journal of Informatics* 5 (2018).

[3] M. Shitkova, V. Taratukhin, and J. Becker. "Towards a Methodology and a Tool for Modeling Clinical Pathways". In: *Procedia Computer Science* 63 (2015), pp. 205–212.

[4] E. Aspland, D. Gartner, and P. Harper. "Clinical pathway modelling: a literature review". In: *Health Systems* 10.1 (2019), pp. 1–23.

[5] M. Mernik, J. Heering, and A. M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (2005), pp. 316–344.

[6] T. Kosar, M. Mernik, and J. C. Carver. "Program comprehension of domain-specific and general-purpose languages: Comparison using a family of experiments". In: *Empirical Software Engineering* 17.3 (2011), pp. 276–304.

[7] F. Ortin, J. Quiroga, O. Rodriguez-Prieto, and M. Garcia. "An empirical evaluation of Lex/Yacc and ANTLR parser generation tools". In: *PLOS ONE* 17 (2022).

[8] E. Book, D. V. Shorre, and S. J. Sherman. "The CWIC/36O System, a Compiler for Writing and Implementing Compilers". In: *SIGPLAN Not.* 5.6 (1970).

[9] F. Di Giacomo, M. Abbadi, A. Cortesi, P. Spronck, and G. Maggiore. "Metacasanova: An Optimized Meta-Compiler for Domain-Specific Languages". In: SLE 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017.

[10] A. Silva. "Model-driven engineering: A survey supported by A unified conceptual model". In: *Computer Languages, Systems and Structures* 20 (2015).

[11] I. Dejanović, R. Vaderna, G. Milosavljević, and Ž. Vuković. "TextX: A Python tool for Domain-Specific Languages implementation". In: *Knowledge-Based Systems* 115 (2017).

[12] T. Baar. "Verification Support for a State-Transition-DSL Defined with Xtext". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 50–60.

[13] I. Dejanovic, M. Dejanović, J. Vidaković, and S. Nikolic. "PyFlies: A Domain-Specific Language for Designing Experiments in Psychology". In: *Applied Sciences* 11 (2021).

[14] S. Graband. "Model Validation in Graphical Cloud Based Editors". Master's Thesis. München: Technische Universität München, 2020.

[15] P. Denny, J. Prather, B. A. Becker, C. Mooney, J. Homer, Z. C. Albrecht, and G. B. Powell. "On Designing Programming Error Messages for Novices: Readability and its Constituent Factors". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, 2021.

[16] A. C. Mühlbacher, V. E. Amelung, and C. Juhnke. "Contract Design: Risk Management and Evaluation". In: *International Journal of Integrated Care* 18.1 (2018).

[17] F. Michel. "A Collaborative Purely Meta-Model-Based Adaptive Case Management Approach for Integrated Care". Dissertation. München: Technische Universität München, 2020.

[18] S. Bönisch. *Discovering Clinical Pathways of an Adaptive Integrated Care Environment*. München, 2021.

[19] F. Michel and F. Matthes. "A Holistic Model-Based Adaptive Case Management Approach for Healthcare". In: 2018.

[20] F. Eckert. "Evaluating the Usability of Acadela: a Domain Specific Language for Defining Clinical Pathways in Integrated Care Platforms". Guided Research. München: Technische Universität München, 2021.

[21] *TextX Semantic Errors*. 2010. URL: `http://textx.github.io/textX/stable/search.html?q=TextXSemanticError`.

[22] *Pyspellchecker*. URL: `https://pyspellchecker.readthedocs.io/en/latest/`.

[23] J. Brooke. "SUS: A quick and dirty usability scale". In: *Usability evaluation in industry* 189 (1995).

[24] A. Bangor, P. Kortum, and J. Miller. "Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale". In: *J. Usability Studies* 4.10 (2009). ISSN: 1931-3357.