TPLP **21** (6): 818–834, 2021.

doi:10.1017/S1471068421000326

Refactoring the Whitby Intelligent Tutoring System for Clean Architecture

PAUL S. BROWN and VANIA DIMITROVA
University of Leeds, Leeds, UK
(*e-mail:* sc16pb@leeds.ac.uk; v.g.dimitrova@leeds.ac.uk)

GLEN HART
Defence Science and Technology Laboratory [dstl]

ANTHONY G. COHN
University of Leeds
Qingdao University of Science and Technology
Tongji University
Shandong University
(*e-mail:* a.g.cohn@leeds.ac.uk)

PAULO MOURA
Center for Research in Advanced Computing Systems, INESC-TEC, Portugal
(*e-mail:* pmoura@logtalk.org)

submitted 10 August 2021; accepted 22 August 2021

Abstract

Whitby is the server-side of an Intelligent Tutoring System application for learning System-Theoretic Process Analysis (STPA), a methodology used to ensure the safety of anything that can be represented with a systems model. The underlying logic driving the reasoning behind Whitby is Situation Calculus, which is a many-sorted logic with situation, action, and object sorts. The Situation Calculus is applied to Ontology Authoring and Contingent Scaffolding: the primary activities within Whitby. Thus many fluents and actions are aggregated in Whitby from these two sub-applications and from Whitby itself, but all are available through a common situation query interface that does not depend upon any of the fluents or actions. Each STPA project in Whitby is a single situation term, which is queried for fluents that include the ontology, and to determine what pedagogical interventions to offer. Initially Whitby was written in Prolog using a module system. In the interest of a cleaner architecture and implementation with improved code reuse and extensibility, the initial application was refactored into Logtalk. This refactoring includes decoupling the Situation Calculus reasoner, Ontology Authoring framework, and Contingent Scaffolding framework into third-party libraries that can be reused in other applications. This extraction was achieved by inverting dependencies via Logtalk *protocols* and *categories*, which are reusable interfaces and components that provide functionally cohesive sets of predicate *declarations* and predicate *definitions*. In this paper the architectures of two iterations of Whitby are evaluated with respect to the motivations behind the refactor: clean architecture enabling code reuse and extensibility.

KEYWORDS: architecture, dependency inversion, Prolog modules, Logtalk

1 Introduction

System-Theoretic Process Analysis (STPA), is an emerging methodology used by system safety analysts from an initial conceptualization before the system design, through to a loss occurring. Trying to determine how a hypothetical system should be designed, built, and maintained in order to prevent potentially catastrophic losses is a difficult and cognitively demanding task. To aid with this task an application has been developed and deployed for a select group in order to test the efficacy of the pedagogical techniques employed within the application. The server-side part of this application is called "Whitby".

Within Whitby three primary domains are discussed. Although it is not necessary to understand these domains to consider the architecture, they are introduced here for orientation:

- Situation Calculus: a many-sorted second order logic for reasoning about situations and actions. The definition used is that of Reiter (2001).
- Ontology Authoring: the knowledge engineering process of defining a formal, ontological model of some simplified world for a purpose (Gruber 1995).
- Contingent Scaffolding: a pedagogical technique in which immediate intervention is offered to a struggling learner at a level of intrusion based upon previous behavior (Wood *et al*. 1976).

Prolog was chosen as an implementation language from the outset due to using both Ontology and Situation Calculus as foundations in the design. However, the entire application has been re-written twice to overcome the architectural issues that forced unsatisfactory solutions. The ramifications of these compromises grew with the complexity of the application. Some of these issues are discussed in Section 3.

In the final rewrite the code base was transitioned to Logtalk and sufficiently decoupled as to extract three libraries. This extraction was the main motivation behind the rewrites: the prior version was functioning but useful parts of it couldn't be shared. The application, including these three libraries, contains approximately 10,000 lines of Logtalk source code plus the Graphical User Interface (GUI) written in ClojureScript. The parts authored in Logtalk cover persistence of user projects, situation calculus reasoning, ontology authoring, contingent scaffolding, web server, and generating HTML including forms based upon the history of a user project.

In this paper, the software engineering principles that guided the application rewriting are presented, the architectures of the logic programming parts of the last two versions of Whitby are compared, the motivations behind the transition from Prolog to Logtalk are discussed, and the lessons learned are summarized.

2 The dependency inversion principle

In refactoring the architecture, the *SOLID principles* of clean architecture (Martin 2018) are applied. These principles are the summary of 20 years of debate between developers attempting to abstract what made their software maintainable and extensible. They are intended to avoid the situation, observed in even market-leading software, where progress is slowing while cost per line of code increases, all while increasing development staff (Martin 2018).

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

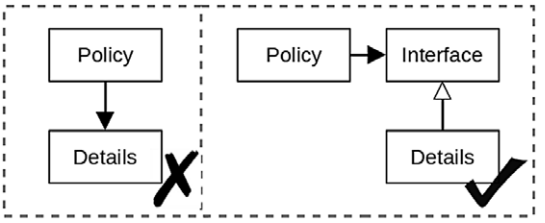

Fig. 1. Dependency Inversion Principle. Left-hand side has high-level policy depending on low-level details, which is not recommended. Right-hand side has the dependency inverted by the policy depending on some interface, which the details extend.

Appropriate application of these principles produces code that is easy to read, maintain, extend and test. These benefits are primarily to those developing and maintaining the software, which then has implications to organizations producing and consuming the software over a period of time. Software with a clean architecture is argued to be easily extensible with new features, typically using a plugin architecture, and robust to changes in business rules, technology, and deployment scenarios (Martin 2018). Thus it reduces application risks and development costs.

The principles of SOLID architecture are:

- **S**ingle Responsibility Principle: each part has one and only one reason to change; it is accountable to one stakeholder
- **O**pen-Closed Principle: code should be open to extension and closed for modification; such as in a plugin architecture where new features are created by adding new code rather than editing existing code
- **L**iskov Substitution Principle: parts should be interchangeable, which makes it robust to even significant changes such as to business rules meeting new legal requirements, or to swapping components such as the database used or GUI framework
- **I**nterface Segregation Principle: do not depend on things not used, which makes dependents of some part robust against changes required by other dependents of that part
- **D**ependency Inversion Principle: high-level policy should not depend on low-level details, but details should depend on policies, such that code which is volatile is not depended upon by code that is stable

The Dependency Inversion Principle, depicted in Figure 1, is the main principle driving this refactor and the technique used for decoupling. Closely related to this principle is the *code for interface, not for implementation* best practice: no concrete module should be imported into any other. Instead, an *abstract definition* of what the module should provide is used. This principle and best practice allows high-level policies to be left untouched as low-level details are swapped or undergo change, which in turn makes reuse of the high-level principles as libraries possible. The concept of interface is thus central to the application of this principle and best practice as further discussed in Section 5. In Logtalk, interfaces are represented using *protocols*, but Prolog module systems do not provide an equivalent feature[1].

[1] The ISO Prolog standard for modules (ISO/IEC 2000) does specify a module interface language construct but only allows a single implementation per interface, thus defeating the main purpose of defining interfaces. Moreover, this standard is ignored by Prolog systems.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

The dependence on an interface, shown in Figure 1, also differentiates the technique from dependency injection or meta-programming where the context of the low-level details are passed to the high-level policy. With or without meta-programming, the policy is dependent on the predicates required being present in the details, however with meta-programming the interface is implicit, ungoverned, and not self-documenting. By using a declared interface as a first-class language feature the required predicates become explicit, the details are governed through a declared promise to define the interface, implementers of the interface can be enumerated using language reflection predicates, and documentation can be automatically generated.

These principles are also considered at the component level (Martin 2018). At this level of abstraction, the key ideas are to enable *reuse* through sensible component contents and *decoupling* through dependency cycle elimination as well as correlating dependency with stability.

3 Whitby before refactoring: OWLSAI

The first version of Whitby, written in Prolog using the module system and depicted in Figure 2, was originally called *OWLSAI* (Web Ontology Language Safety Artificial Intelligence). This application did implement the features required of it[2]. There was no particular issue with it from a *user* perspective. The issues were entirely at the *developer* experience level.

The `kb` directory in Figure 2 is responsible for the ontology authoring with situation calculus. The `oscar` directory, which is responsible for the contingent scaffolding interventions, is only dependent upon the code within this module. This dependence is a necessity as `oscar` needs to know about a user's ontology in order to offer relevant interventions.

Several violations of the SOLID principles are hidden at this level of abstraction. Note that `golog` and `fluents` are aggregated in `kb_manager`. This compromise is necessary because the Golog Situation Calculus reasoner from Reiter (2001) includes a predicate that calls these fluents, and so `golog` is dependent upon `fluents`. It is not uncommon in Prolog to apply some set of rules, like those in `golog`, to some facts, like those in `fluents`.

But for `golog` to reason over the `fluents` and `actions` defined for some particular world under analysis requires that world (details) to be imported into `golog` (policy). In this manner, the abstract is dependent upon the concrete, the stable is dependent upon the flexible, the calculus is dependent upon its own application. It's a violation of clean architecture that prevents code reuse: `golog` cannot be extended to include a defined world to reason over without modification to its own source code. Concurrent handling of multiple defined worlds is also precluded. Although there are workarounds to these problems, some of which are discussed here, they are unsatisfactory as the lack of necessary language constructs to cleanly express the application architecture results in the violation of SOLID principles.

[2] The only feature requirement that changed between OWLSAI and Whitby was a change from handling users with logins and many projects, to only handling many projects. Therefore this aspect is not compared.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

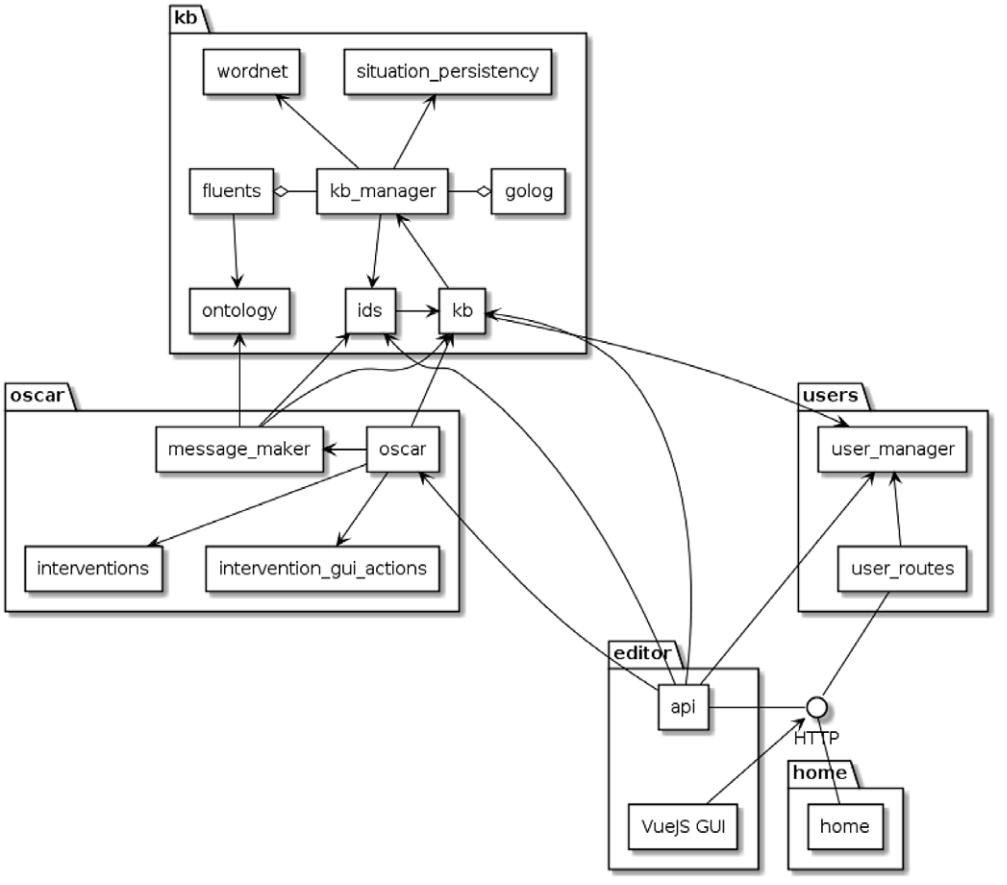

Fig. 2. Dependencies in OWLSAI. Each node is a file, within their directories, which distinguish modules. Arrows denote imports, open diamonds denote consults.

To circumvent the issue of circular dependencies in OWLSAI, the `golog` and `fluents` files were consulted instead. This loaded them both into the `kb_manager` namespace. However, this causes a conundrum to resolve as there are fluents and actions for the `oscar` module that need to be defined in the `fluents` and `kb_manager` file so that `golog` is in the same namespace as them. For example, actions pertaining to ontology authoring, contingent scaffolding, and user interface are defined adjacent to each other in `kb_manager`, in violation of the Single Responsibility Principle, as seen in this snippet:

```
:- consult(kb(golog)).
:- consult(kb(fluents)).

%! action(Action, GologPossQuery)

% Ontology Authoring
action( add_data(_User, _Time, Payload),
        -asserted(Payload)).
```

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

```
action( delete_data(_User, _Time, Payload),
        asserted(Payload)).

% User Scaffolding Actions
action( dismiss_intervention(_User, _Time, Fact, Level),
        intervened(_, Fact, Level, _)).
action( request_intervention_increase(_User, _T, ID, Fact,
                                      Level),
        intervened(ID, Fact, Level, _)).

% Agent Scaffolding Actions
action( intervene(_User, _Time, Fact, Level, _Payload),
        \+some(n, (dismissed(Fact, n) & n >= L))).

% User UI Actions
action( navigate_to_step(_User, _Time, _Step), true).
action( concept_focus(_User, _Time, _Focus), true).
action( glossary_lookup(_User, _Time, term(_Term)), true).
action( nudge(_User, _Time, _R), true).
```

Code belonging to `oscar` resides in files in `kb` that is loaded into a different module. It should reside in files in `oscar` that are somehow made visible to `kb` to maintain separation of responsibilities and to ease code navigation. There are mechanisms to achieve this in Prolog: via consulting which would warn if a predicate were redefined, or via `include/1`, which includes the text of the file within the other.

Another mechanism tried for including actions from different modules into `kb_manager` was to declare `action/2` as a multifile predicate in `kb_manager`. Clauses for the predicate could then be defined in `oscar` and any other module by using a prefix: `kb_manager:action(...)`. However, this violates the Dependency Inversion Principle as high-level policy predicates belonging to general ontology authoring and scaffolding are then dependent on the low-level detail that is the `kb_manager`, which is the module responsible for updating and querying user projects extending the ontology. Furthermore, this prefix referring to a specific, fixed module would prohibit the substitution of that module, thus violating the Liskov Substitution Principle of SOLID.

To use `include/1` directives or multifile predicates would be to take code from `oscar` and have it effect the behavior of `kb`; thus a developer working on either module must understand how the other one is working. A poorly placed cut, unfortunately named predicate, or redefinition of an operator in `oscar` could cause `kb_manager`, upon which it depends, to no-longer function correctly. It opens up the potential for the consumer of some code to break what it should only depend upon. A developer debugging `kb` looking solely at their code in `kb`, believing it has no dependencies as the architecture diagram shows, would have little hope of resolving such an error. For these reasons module systems are favored over the older `consult/1` and `include/1` predicates and why using them is also an unsatisfactory solution. When authoring OWLSAI, the more robust unsatisfactory solution was chosen, putting `oscar` code into `kb`, violating the Open-Closed principle and preventing code reuse, but easing debugging.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

Ideally a module would be used but the dependency needs to be inverted, such that `fluents` depend upon `golog`. One workaround within the module system would be to make `kb_manager` dependent upon `fluents` and `golog` directly. Then the module can be included with the fluent or action where it is defined in all queries to Golog[3]. For example:

```
holds(Module:Fluent0, Situation) :-
  Module:restoreSitArg(Fluent0, Situation, Fluent),
  Module:Fluent.
holds(Module:Fact, Situation) :-
  not Module:restoreSitArg(Fact, Situation, _),
  isAtom(Fact),
  Module:Fact.
```

This example also represents the resultant code of one strategy attempted via using meta-predicates to invert the dependency without using the interface depicted in Figure 1. With `holds/2` defined as a meta-predicate the calling context is passed implicitly, but `restoreSitArg/3` will be defined in the same module as the `Fluent`, which may be in a different module from the calling context: in Whitby there are multiple calling contexts, whereas each fluent is defined once. To make `restoreSitArg/3` available to the calling context would result in name-clashes when more than one module defining fluents is used. Therefore the dependency module where the definitions reside needed to be passed (or injected) for context as per this example.

The concern for this example is in the Golog call to `Module:Fluent`, where `Fluent` could be anything, including a meta-predicate, given in the query, which is a qualified call potentially breaking the encapsulation of the module. Furthermore, it's no longer possible to use `holds/2` with a variable as the first argument to find fluents that hold in a ground situation without explicitly enumerating all modules and testing if they define fluents or not; the lack of protocols/interfaces as first-class entities precludes a simple and clean enumeration of only those modules that would declare conformance to a given protocol. The import semantics of Prolog modules also would force the use of these explicitly qualified calls for the conforming modules to prevent predicate import clashes. This goes against what is considered best practice with Prolog modules: the use of implicit imports and implicit module-qualified predicate calls. But that is not the primary issue: by making a module that defines fluents and actions an *explicit* argument, we are forced to anticipate all predicates that, although not accessing fluents and actions directly, may be indirectly calling a predicate that requires that access (and thus require the module argument to be passed from upstream).

In Whitby however, which makes use of the required language constructs provided by Logtalk, `SitCalc` is loaded as a third-party package. As Logtalk does not use module-like imports semantics, there are never any loading conflicts when two or more loaded objects define the same public predicates. Furthermore, Whitby also loads packages defining ontology authoring terms and contingent scaffolding terms. The only place in Whitby

[3] Available at: http://www.cs.toronto.edu/cogrobo/kia/

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

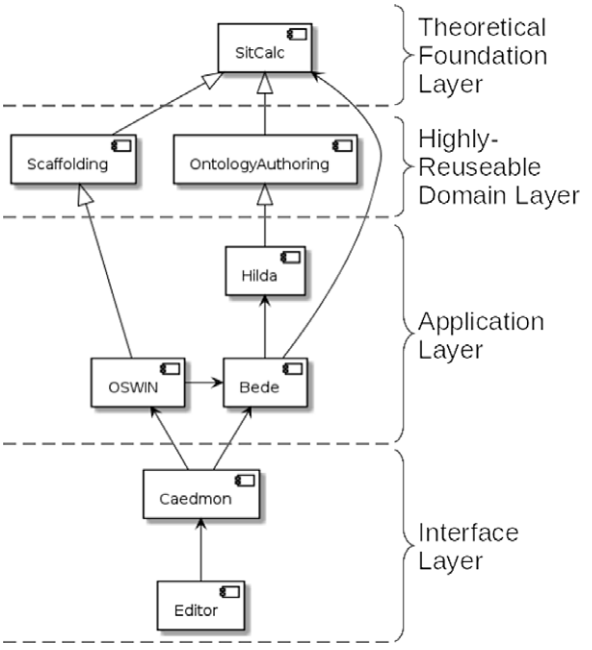

Fig. 3. Dependencies between components of Whitby. Open arrows denote extension, closed arrows denote dependence.

where the contents of those packages need be considered is in the use of their fluents in queries of a situation and in the doing of their provided actions, both of which are done without the requirement to explicitly define the correct context to reason about them in.

Although dependency inversion is the crucial issue, there are additional violations of clean architecture that need to be addressed. The dependency cycle between `kb_manager`, `ids`, and `kb` can cause a small edit in one of them to have perpetual ramifications as its dependency graph is also adapted to the change. Golog is more than a Situation Calculus reasoner; it is a parser for a Situation Calculus based language; thus `kb` is depending on code that it does not use. Furthermore, the four dependencies from the `oscar` module to the `kb` module suggest substitution would require more effort than necessary.

4 Refactored Whitby

The abstract architecture of Whitby is depicted in Figure 3, whereas a detailed view is in Figure 4. From the abstract view it can be seen how Whitby was designed to decouple the components of OWLSAI enabling code reuse. It is not possible to layer OWLSAI in a similar manner due to the compromises made and tight coupling.

`SitCalc` provides the theoretical foundation, which can be used to tackle a multitude of problems, it depends on nothing. The next "Highly Reusable Domain Layer" is the application of `SitCalc` to two domains; these libraries depend on `SitCalc`, but nothing in Whitby. Therefore they can be reused by any application wishing to apply Situation Calculus to Contingent Scaffolding or Ontology Authoring. The "Application Layer" is the core of the Whitby application, it is this code that applies the reusable libraries to the particular task at hand: Contingent Scaffolding an STPA analyst who is unwittingly

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

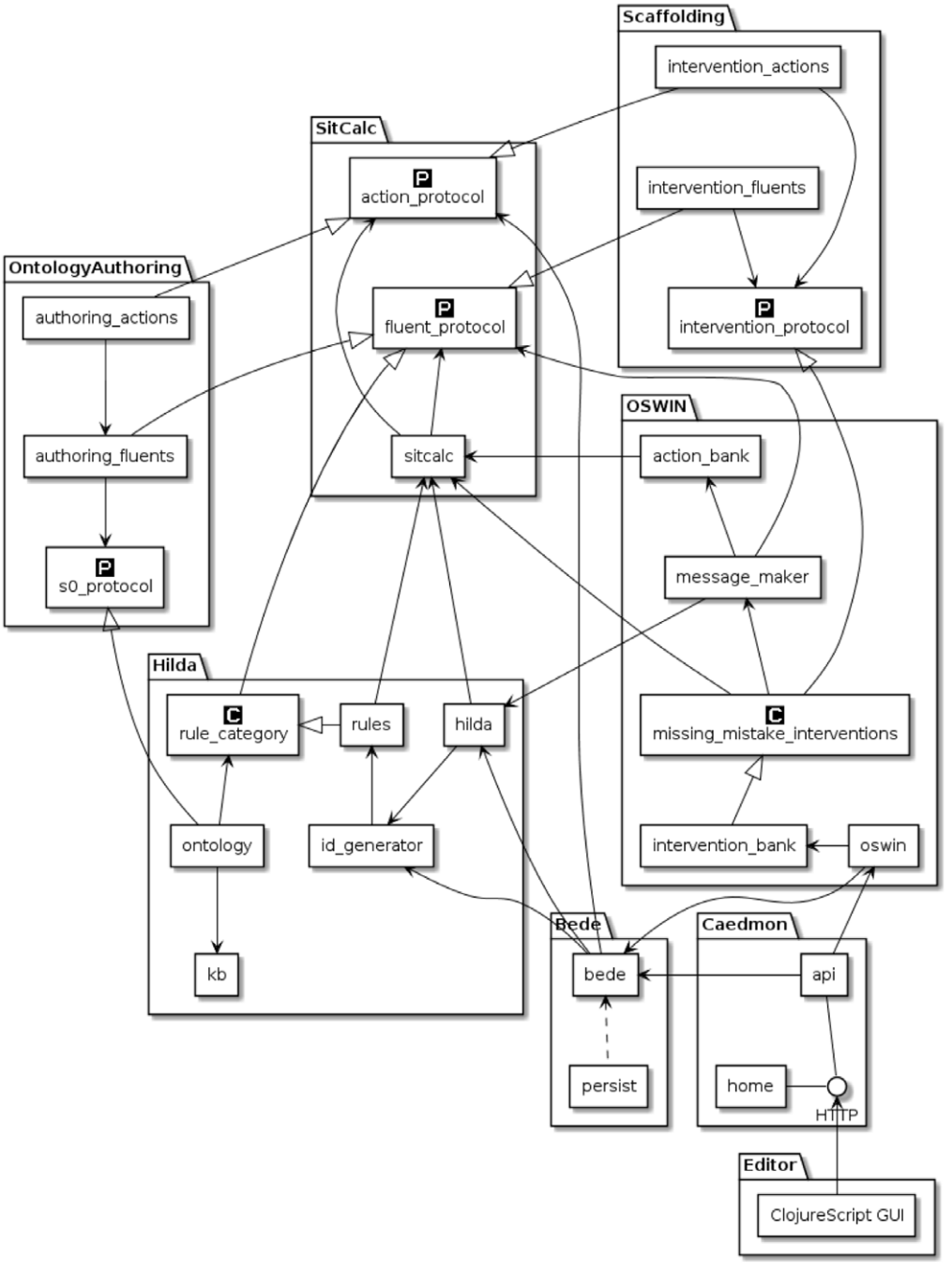

Fig. 4. Dependencies in Whitby and extracted libraries. Each node (without a mark) is an object, within their directories. Protocols are marked with a "P", categories with a "C". Closed arrows denote dependence, open arrows denote implementation or extension, dashed arrow denotes event monitoring.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

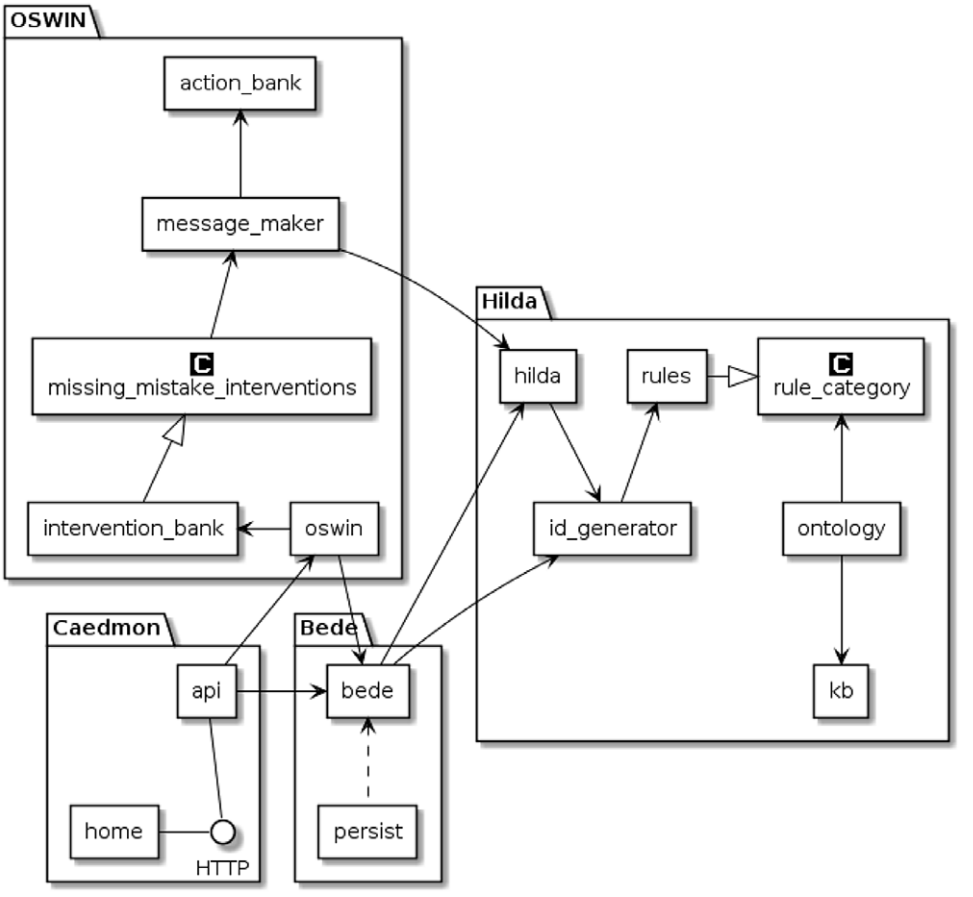

Fig. 5. Dependencies within Whitby only. Each node is an object, within their directories. Arrows denote dependence, open arrows denote implementation or extension, dashed arrow denotes event monitoring. Categories are marked with a "C".

authoring an extension to an ontology. Finally the "Interface Layer" provides a convenient means for the user to interact with the application.

Whitby loads `SitCalc`, `OntologyAuthoring`, and `Scaffolding` defined as third-party libraries. In this application a naming convention around Whitby Abbey was adopted to aid in organizing the code, thus the modules are:

- **OSWIN**: **O**ntology-driven **S**caffolding **W**ith **I**nteractive **N**udges (extends `Scaffolding`)
- **Hilda**: The wise, Hilda handles the ontology authoring (extends `OntologyAuthoring`)
- **Bede**: The historian, records the actions that are done
- **Caedmon**: The poet, responsible for the user interface

The architecture, shown in Figure 4, initially appears more complex than OWLSAI in Figure 2 as the third-party libraries that were extracted are also included, together with the protocols used to achieve dependency inversion. Figure 5 shows only the internal dependencies: how the application appears to a developer working on it. Such a developer

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

need not concern themselves with the working of any of the imported libraries; they are only responsible for what is depicted in Figure 5. For example, Whitby required a fluent describing what the user is looking at in the GUI; this is particular to the application of Whitby and so is not defined in `OntologyAuthoring`. To add this fluent to Whitby requires creating a new object that conforms to the `fluent_protocol`: new behavior via extension rather than modification and exposing fluents that the application developer has no business editing.

Figure 5 is a cleaner architecture, with no dependency cycles. However it is not yet perfect. For example, `bede` should not depend upon `id_generator`. That particular predicate should be exposed through `hilda`, which provides an interface enabling easier substitution of the objects that `hilda` depends upon. Early in the refactoring to Whitby, each of the named directories was implemented as its own microservice communicating over HTTP. Correcting this issue would make it simple to split Whitby back into microservices for scalability, which isn't possible to achieve with OWLSAI due to the tight coupling between components.

5 Dependency inversion using Logtalk protocols

To achieve the desired architecture requires the application of the Dependency Inversion Principle, which can be accomplished via the Abstract Factory design pattern (Gamma *et al*. 1997; Martin 2018) described as:

> Provide an interface for creating families of related or dependent objects without specifying their concrete classes. (Gamma *et al*. 1997)

In logic programming, we can reinterpret the implicitly imperative idea of *creating families* as *declaratively defining families*. Therefore, with Logtalk it becomes possible to do Dependency Inversion without dynamically creating objects. The concept of interface, in turn, is readily available using Logtalk *protocols*, as described below.

The Dependency Inversion Principle is applied to decouple the application into three major components. First a `SitCalc` library is extracted. Then `SitCalc` is extended, not modified, to create `OntologyAuthoring` and `Scaffolding` libraries. Finally, Whitby is created by importing these libraries as third-party libraries. The final architecture, with these libraries included, is shown in Figure 4. We start with a brief overview of Logtalk followed by a detailed account of how we applied this design principle to each component.

5.1 Logtalk overview

Logtalk as a language reinterprets object-oriented concepts from first principles to provide logic programming with code encapsulation and code reuse mechanisms that are key in expressing well understood design principles and patterns (described in depth in "The Logtalk Handbook" (Moura 2021)). A key feature is the clear distinction between *predicate declarations* and *predicate definitions*[4], which can be encapsulated and reused as follows:

[4] This distinction exists in standard Prolog (ISO/IEC 1995) only for predicates declared as *dynamic* or *multifile*. Notably, *static* predicates exported by a module must be defined by the module.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

- *protocols*: Group *functionally cohesive* predicate declarations that can then be implemented by any number of *objects* and *categories*. Allows an object or category to promise conformance to an interface.

- *objects*: Group predicate declarations and predicate definitions. Objects can be *stand-alone* or part of hierarchies. Object enforce encapsulation, preventing calling predicates that are not within scope. Predicates are called using *message sending*, which decouples calling a predicate from the predicate definition that is used to answer the message.

- *categories*: Group a *functionally cohesive* set of predicate declarations and predicate definitions, providing a fine-grained unit of code reuse that can be imported by any number of objects, thus providing a *composition* mechanism as an alternative to the use of inheritance.

Predicates can be declared *public*, *protected*, or *private*. A predicate declaration does not require that the predicate is also defined. Being able to *declare* a predicate, independent of any other predicate properties, without necessarily *defining* it is a fundamental requirement for the definition of protocols. It also provides clear *closed world semantics* where calling a declared predicate that is not defined simply fails instead of generating an error (orthogonal to the predicate being *static* or *dynamic*).

Logtalk defines a comprehensive set of *reflection* predicates for reasoning about the use of these components in the program. In particular, the `conforms_to_protocol/2`, which is true if the first argument implements or is an extension of something that implements the protocol named in the second argument, and `current_object/1`, which is true if its argument is an object in the application current state (categories and protocols have their own counterparts). These predicates are used in the implementation of the SOLID principles as illustrated in the next sections.

Logtalk also provides a comprehensive set of portable developer tools, notably for documenting, diagramming, and testing that were used extensively. These tools reflect how the language constructs are used in applications, from API documentation to diagrams at multiple levels of abstraction that help developers and maintainers navigate and understand the code base and its architecture.

5.2 A reusable sitCalc library

The `SitCalc` library includes predicates that need to send messages to fluent and action objects. Rather than depend on these fluents and actions directly, it depends instead on objects conforming to `action_protocol` and `fluent_protocol`. This is the Dependency Inversion Principle of SOLID: to depend only on protocols/interfaces and not on concrete code (Martin 2018). These protocols declare the predicates that an action and fluent are expected to define:

```
:- protocol(action_protocol).

        :- public(do/2).
        :- info(do/2, [
```

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

```
                comment is 'True if doing action in ``S1``
                results in ``S2``.',
                argnames is ['S1', 'S2']
        ]).

        :- public(poss/1).
        :- info(poss/1, [
                comment is 'True if the action is possible in
                the situation.',
                argnames is ['Situation']
        ]).

:- end_protocol.

:- protocol(fluent_protocol).

        :- public(holds/1).
        :- info(holds/1, [
                comment is 'True if the fluent holds in the
                situation.',
                argnames is ['Situation']
        ]).

:- end_protocol.
```

Thus any object that is an action or fluent can be found or validated, using the Logtalk built-in reflection predicates[5]. For some strategies attempted without an interface in OWLSAI, such as when passing the definition context explicitly (as previously discussed in Section 3), the enumeration of modules requires a hand-coded alternative to mark the modules, which is fragile and not self-documenting. These predicates are used to validate or enumerate:

```
is_action(Action) :-
  conforms_to_protocol(Action, action_protocol),
  current_object(Action).

is_fluent(Fluent) :-
  conforms_to_protocol(Fluent, fluent_protocol),
  current_object(Fluent).
```

[5] The `conforms_to_protocol/2` predicate enumerates both objects and categories that implement a protocol. As we are only interested in objects, we use the `current_object/1` predicate to filter out any categories as these are used only to provide common definitions for utility predicates.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

Now within the `sitcalc` object when it is necessary to call a fluent or action they can be called, even if the argument is a variable, without depending on the fluents or actions. Here are two extractions from the code within `sitcalc` that demonstrate doing so:

```
holds(Fluent, Situation) :-
  is_fluent(Fluent),
  Fluent::holds(Situation).

poss(Action, Situation) :-
  is_action(Action),
  Action::poss(Situation).
```

In addition to this, the extraneous code in Golog is not included in `SitCalc` such that unused code is not depended upon. Also there is more than one way to represent a situation in Situation Calculus: either as a history of actions or as a collection of fluents. Therefore in the publicly available version[6] of the `SitCalc` library, the common parts of both representations are combined into a `situations` category, with both representations importing it to ease substitution.

The final detail abstracted from Figure 4 is the definition of action and fluent categories, which import their respective protocols. The action category defines the `do/2` predicate and the fluent category applies tabled resolution to `holds/2` if available in the backend, which greatly improves performance of context-dependent queries over long situation terms.

5.3 Extending sitCalc with reuseable libraries

The two `OntologyAuthoring` and `Scaffolding` libraries both extend `SitCalc`, but both are also defined in a way that they can be used as third-party libraries with `SitCalc` as a dependency. They extend `SitCalc` by defining fluents and actions that are pertinent. `OntologyAuthoring` includes a fluent to see what triples hold in the initial situation. Here, `s0` is a *marker protocol*, allowing easy enumeration of initial situations by using the reflection predicates, and also dependency inversion via a protocol (`fluent` is a category that implements `fluent_protocol`):

```
:- object(initial_assertion(_Subject_, _Predicate_, _Object_),
  imports(fluent)).

  holds(_AnySit) :-
    conforms_to_protocol(S0, s0),
    current_object(S0),
    S0::asserted(_Subject_, _Predicate_, _Object_).

:- end_object.
```

[6] Comprised of the libraries made available at: `https://github.com/PaulBrownMagic/Situations`, `https://github.com/PaulBrownMagic/Sitcalc`, and `https://github.com/PaulBrownMagic/STRIPState`

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

`Scaffolding` includes an action to intervene (here `action` is a category that implements `action_protocol`):

```
:- object(intervene(_Intervention_, _Query_, _Lvl_, _Time_),
  imports(action)).

  poss(Sit) :-
    conforms_to_protocol(Interventions, interventions),
    current_object(Interventions),
    Interventions::intervention(_Intervention_, _Query_),
    sitcalc::holds(_Query_, Sit),
    intervention_level(_Intervention_, _Query_, _Lvl_)::
    holds(Sit),
    \+ live_intervention(_Intervention_, _Query_, _Lvl_)::
    holds(Sit).

:- end_object.
```

Both objects are *parametric* objects (Moura 2011). The object parameters (e.g. `_Subject_` are logic variables shared with all the object predicates.

Due to the implementation of `SitCalc`, all these fluents and actions are visible to `sitcalc` whilst it does not depend on them. However, these two examples both depend upon some implementation details: some `S0::asserted/3` and some `interventions::intervention/2`. These dependency issues are solved in the same manner as for `SitCalc`: through dependency on a protocol (as illustrated in Figure 4).

Between these two libraries a total of 14 fluent and action terms are introduced that can be queried via `SitCalc`. Although these depend on `SitCalc`, they do not depend on any application that makes use of them. Whitby is such an application, by importing these libraries it gains these 14 fluents and actions, needing only to implement both the `s0_protocol` and `intervention_protocol`. In contrast to OWLSAI, the contingent scaffolding is also not dependent on code that includes ontology authoring, meaning it can be applied to other activities than ontology authoring.

6 Conclusion

Taking a set of rules and applying them to some facts is a typical task in Prolog. However, the limitations of the module system often result in code that only handles a fixed set of facts at a time, either imported into the rules module or loaded into the `user` special module. But sometimes these rules are useful to many applications, as is the case with Situation Calculus. When the rules are to be shared as third-party libraries, any dependency of rules on facts needs to be inverted to decouple the rules from a particular set of facts. This dependency inversion allows multiple set of facts to be loaded and used concurrently (providing an alternative solution for implementing the *many-worlds* design pattern). Key to this dependency inversion is the concept of *interface* or *protocol*, supported by Logtalk but absent in Prolog module systems.

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

This inversion was achieved in Logtalk by taking inspiration from the Abstract Factory design pattern and considering how it could be achieved with *protocols* and *categories*. The final solution is simpler than the Abstract Factory design pattern as no dynamic creation of objects is necessary. Instead, dependency upon a *protocol* and conforming to it is all that is required. This is an elegant pattern for Logtalk that can be repeated when creating third-party libraries to reason about definitions in an application without depending upon them.

The use of *protocols* in this manner results in a *plugin architecture*. A third-party can "plug-in" code to the `SitCalc` library, or other libraries, to work with it. This is a very versatile design pattern as it allows an application developer, or even third-parties and end-users provided with a plugin loading interface, to adapt the behavior of the application to their needs without editing the core application code. It also leaves the application immune from changes made elsewhere via the plugin, with the provision they are not malicious, by the drawing of boundaries in the architecture (Martin 2018).

This use of *protocols* has focused on their application for dependency inversion due to the specifics of the Whitby application architecture. It should be noted their use also resulted in adherence to the Single Responsibility and Open-Closed Principles. Protocols also have significant contribution to adherence to the Liskov Substitution Principle, making it a simple matter to swap objects that adhere to the same protocol, as well as the Interface Segregation Principle by providing explicitly defined interfaces as first-class entities.

The refactor from OWLSAI to Whitby decoupled code from OWLSAI that can be reused, which are published as third-party libraries to satisfy the motivation behind the refactoring. This has simplified Whitby, where there is less functionality now to maintain, and has enabled other applications and libraries to use Situation Calculus reasoning while also keeping a clean architecture. The workarounds that we attempted to compensate for the lack of required features in the Prolog module systems accumulated and increased the complexity of the application. Those workarounds are not supported by development tools (especially documenting and diagramming tools) and raised new issues, thus creating additional burden on developers while not solving the reusable goals that prompted the refactoring.

By using the language constructs provided by Logtalk to apply SOLID principles in the refactoring, the Whitby application documentation and diagrams trivially reflect the actual architecture of the application, further simplifying development and maintenance. But hand-coded workarounds that try to compensate for missing language features (in this case: the module system in the original version of the application) required additional effort to document as they are not visible to developer tools as first-class constructs. These workarounds must also be repeated in every application with the impact of their limitations carefully taken into account.

This refactoring has benefited the Whitby application, the Situation Calculus reasoning is open to extension without modification, which was used to add application specific fluents and actions as the need arose. Additionally, the separation of responsibilities has made it easier to navigate and edit the code base. But the primary benefit is to other applications that wish to make use of the extracted libraries. Whitby demonstrates how

https://doi.org/10.1017/S1471068421000326 Published online by Cambridge University Press

they can be reused. Bedsit[7] is one example of such reuse: it is an exploratory framework for rapidly prototyping applications using `SitCalc` and includes both TicTacToe and ToDo example applications with a variety of UIs. The first author has also reused `SitCalc` and `OntologyAuthoring` to quickly prototype a proprietary ontology browser and editor.

As part of the AI4EU[8] initiative, a third-party has been provided with Whitby to adapt to a new project in the domain of robotics planning. Due to Whitby's adherence to the Single Responsibility and Liskov Substitution Principles, which was not possible with OWLSAI, the third-party should need only to make changes at the periphery of the code base: telling `kb` to load a different OWL file, optionally substituting any reasoning rules specific to their domain, substituting `intervention_bank` for an object with appropriate interventions, and substituting the `Editor` GUI to be appropriate for that domain. There is still room for improvement, however. For example, they also need to change a list in `action_bank`, which contains the classes used as tabs in the GUI.

Whitby is currently deployed to test the efficacy of the pedagogical techniques implemented. Should the application prove useful, any remaining architectural issues will be addressed, although the lesson learned regarding using dependency inversion to decouple the abstract rules from concrete facts is consistently applied in all other current software development efforts.

Acknowledgements

The authors gratefully acknowledge the financial support provided: an EPSRC CASE studentship partially funded by the Defence Science and Technology Laboratory. The fourth author is partially funded by the EU AI4EU project (825619) and is a Fellow of the Alan Turing Institute.

References

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1997. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Massachusetts.

Gruber, T. R. 1995. Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies 43*, 5–6, 907–928.

ISO/IEC. 1995. *International Standard ISO/IEC 13211-1 Information Technology — Programming Languages — Prolog — Part I: General core*. ISO/IEC.

ISO/IEC. 2000. *International Standard ISO/IEC 13211-2 Information Technology — Programming Languages — Prolog — Part II: Modules*. ISO/IEC.

Martin, R. C. 2018. *Clean Architecture: A Craftman's Guide to Software Structure and Design*. Prentice Hall, Hudson, New Jersey.

Moura, P. 2011. Programming patterns for Logtalk parametric objects. In *Applications of Declarative Programming and Knowledge Management*, S. Abreu and D. Seipel, Eds. Lecture Notes in Artificial Intelligence, Vol. 6547. Springer-Verlag, Berlin Heidelberg, 52–69.

Moura, P. 2021. *The Logtalk Handbook* (Release 3.46.0 ed.).

Reiter, R. 2001. *Knowledge in Action*. The MIT Press, Cambridge, Massachusetts.

Wood, D., Bruner, J. S. and Ross, G. 1976. The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry 17*, 2, 89–100.

[7] `https://github.com/PaulBrownMagic/BedSit`

[8] `https://www.ai4eu.eu/` Established to build the first European Artificial Intelligence On-Demand Platform and Ecosystem with the support of the European Commission under the H2020 programme.