Science of Computer Programming

Software Evolutionary Architecture: Automated Planning for Functional Changes

--Manuscript Draft--

Manuscript Number:	SCICO-D-22-00337
Article Type:	Research Paper
Keywords:	Evolution Planning; Formal Method; Model Checking; Functional changes; Evolutionary Architecture

Abstract:	Software systems often evolve over time due to frequent changes in user requirements. Many functional changes result in refactoring of the architectural design, which dramatically impacts the software system. Evolutionary architecture is a design principle that helps to support making frequent changes. The key aspect of evolutionary architecture is defining fitness functions to ensure that the changes meet the goals to be achieved. However, planning the evolution of architectural design in incremental steps remains a challenge. This paper presents an approach to automatically generating evolution plans to refactor the architectural design aimed at supporting new functionalities. Formal modeling has been applied to allow functional properties to be verified against the design. With the generated evolution plan, we can determine the safe path to evolve the software system with minimal risk of failure. We have evaluated the rigorousness and effectiveness of the evolution plan generated by our approach for six software systems. Our experimental results showed that the proposed approach is effective in generating evolution plans. Moreover, we are able to identify the most suitable planning strategy, which generates the evolution plan with minimal system interruptions.

Powered by Editorial Manager® and ProduXion Manager® from Aries Systems Corporation

Highlights

Software Evolutionary Architecture: Automated Planning for Functional Changes

Nacha Chondamrongkul, Jing Sun

- Automated planning to refactor the architectural design for evolutionary architecture.
- Modelling of architectural design to support automated planning and checking.
- Formal verification as a fitness function to guarantee the functional properties.
- Safe evolution path can be determined to minimise interruptions during the evolution.

Manuscript
Click here to view linked References

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

Highlights

Software Evolutionary Architecture: Automated Planning for Functional Changes

Nacha Chondamrongkul, Jing Sun

- Automated planning to refactor the architectural design for evolutionary architecture.
- Modelling of architectural design to support automated planning and checking.
- Formal verification as a fitness function to guarantee the functional properties.
- Safe evolution path can be determined to minimise interruptions during the evolution.

Software Evolutionary Architecture: Automated Planning for Functional Changes

Nacha Chondamrongkul

School of Information Technology, Mae Fah Luang University, Thailand

Jing Sun

School of Computer Science, University of Auckland, New Zealand

---

Abstract

Software systems often evolve over time due to frequent changes in user requirements. Many functional changes result in refactoring of the architectural design, which dramatically impacts the software system. Evolutionary architecture is a design principle that helps to support making frequent changes. The key aspect of evolutionary architecture is defining fitness functions to ensure that the changes meet the goals to be achieved. However, planning the evolution of architectural design in incremental steps remains a challenge. This paper presents an approach to automatically generating evolution plans to refactor the architectural design aimed at supporting new functionalities. Formal modeling has been applied to allow functional properties to be verified against the design. With the generated evolution plan, we can determine the safe path to evolve the software system with minimal risk of failure. We have evaluated the rigorousness and effectiveness of the evolution plan generated by our approach for six software systems. Our experimental results showed that the proposed approach is effective in generating evolution plans. Moreover, we are able to identify the most suitable planning strategy, which generates the evolution plan with minimal system interruptions.

Keywords: Evolution Planning, Formal Method, Model Checking, Functional property, Evolutionary Architecture

---

Email addresses: nacha.cho@mfu.ac.th (Nacha Chondamrongkul), jing.sun@auckland.ac.nz (Jing Sun)

1. Introduction

The ecosystem of software products generally consists of various elements such as user requirements, runtime environment, development libraries and programming languages. These elements inevitably change over time due to the evolution of user requirements. The previous studies [1] have found that the changes in functional requirements are usually the main cause of software changes. Some functional changes need refactoring in the architectural design to support them. However, the architectural refactoring may impact original functionalities and cause system failures. Evolutionary architecture [2] has been proposed to mitigate the impact of changes by frequent but fine grained updates towards the system structure. The fundamental concept of the evolutionary architecture is making frequent small changes over time in agile manner through continuous delivery [3]. Similar to evolutionary computing, the evolutionary architecture has fitness functions defined to ensure that the architectural refactoring can achieve the desired goals [4]. Even though the evolutionary architecture can be applied to support constant changes, many challenges have been posed as discussed in Section 2.2 and it is still lacks of tool to support planning.

Many researchers have tried to automate refactoring software architecture and planning software evolution, as discussed in detail in Section 7. Existing approaches, such as [5, 6, 7] have been proposed to automatically guide how the refactoring should be performed to achieve certain defined objectives using search-based algorithms. Some approaches [8, 9, 10, 11, 12] have been proposed as a tool to recommend refactoring to enhance particular set of quality attributes. Even though the existing approaches could enhance the efficiency of architectural refactoring, they fail to take the functional properties into consideration. Hence, how to guarantee system functionality and integrity during a software architecture evolution is still a challenging topic. Some approaches, such as [13, 14, 15, 16] have been proposed to automate the evolution planning, but they focus on the code level checking. Hence, they are limited to a particular infrastructure or programming language. There are approaches, such as [17, 18, 19] proposed to plan the evolution at the architecture level. These approaches aim at generating the optimal plan for the evolution of software architecture. However, fitness functions cannot be defined to verify that the desired properties can be achieved, as they are not designed for the evolutionary architecture. To the best of our knowledge, there has been a lack of an approach to automating the evolution planning

for the evolutionary architecture.

This paper presents a solution that can automatically generate the evolution plan for the evolutionary architecture. We previously presented the architectural migration technique in [20], which supports the transformation of the architectural pattern without interrupting functionalities. However, many changes involve functional changes that require architectural refactoring. Therefore, this work focuses on planning the evolution of functional changes. The contribution of our approach can be summarised as follows:

- The functional changes to the software architecture design can be formally defined. The new functional properties can be formally described, as can the reference architecture to incorporate into the new design.
- Algorithms have been proposed to generate the problem description, which serves as an input to the automatic planning of the evolution process. These algorithms support three types of refactoring that apply different changes to the architectural design configuration.
- The evolution plan can be automatically generated as a sequence of architectural refactoring steps to support functional changes. It is then used to generate a path consisting of models and fitness properties, which represent the architectural design after each step of the evolution in the plan is performed.
- Formal verification can be performed to determine whether certain functional properties can be achieved on the models along the evolution path. Moreover, it helps to determine which evolution steps may cause potential interruptions and hence need to be managed to avoid the risk.

The remainder of this paper is organised as follows. Section 2 discusses the evolutionary architecture and its challenges. The architectural modelling is presented in Section 3. Section 4 presents how the evolution can be automatically planned to support different types of architectural refactoring. The implementation of tools to support using this approach in practice is presented in Section 5. Our approach has been evaluated and the results are presented in Section 6. Section 7 discusses the related works. Finally, section 8 concludes the paper and outlines future research directions.

2. Motivation

This section presents our motivation. We explain the concept of evolutionary architecture and discuss its challenges.

2.1. Evolutionary Architecture

As software systems need to evolve over time, certain quality attributes and functionalities that architects design to support at the beginning often require changes later on. These changes are usually made without a thorough understanding of the current architectural design, which may cause the degradation of system qualities [21]. Evolutionary architecture is a concept proposed by Ford et al.[2] to prevent such degradation in the context of the evolution of software architecture. Its definition is as follows:*"An evolutionary architecture supports guided, incremental change across multiple dimensions"*.

The evolutionary architecture aims to support frequent small changes according to an agile methodology to embrace the feedback loop that allows engineers to learn how the system is evolving, without losing the important characteristic of architecture [2]. This change is often fine-grained to allow developers to embrace modularity by focusing on a particular part of the system and minimising coupling among other parts. The deployment of the incremental change is often performed with the least interruption to the running system [22]. Therefore, the new configuration has to be able to run alongside the old configuration, which will be obsolete when the old system is no longer in use. For example, if A and B consume service X, which needs to be changed to service Y. During the migration, A may consume a new service Y, while B still needs to consume X. Eventually, when B is ready to consume Y, X will be obsolete.

As the system evolves, important requirements such as functionalities must be preserved. Therefore, the changes must be guided to protect them. In evolutionary computing, fitness functions determine the candidate solution to achieve the goals [23]. The fitness function in evolutionary architecture is inspired by that of evolutionary computing. Unlike the fitness function in evolutionary computing that helps to find the best candidates, the fitness functions in evolutionary architecture help to ensure that the architectural refactoring can achieve the desired goals without degrading any original requirements or quality attributes [2]. These fitness functions need to be defined as early as when the software architecture was designed [4]. When

an architect designs the software architecture to support functionalities or quality attributes that must be protected as the system evolves, one or more fitness functions are introduced for such a purpose. The fitness functions are usually implemented as unit tests that verify the correct output of defined test cases. When implementing evolutionary architecture, unit tests as fitness functions can be included in the deployment pipeline, which is executed every time the source code is updated and deployed by the system. As the evolutionary architecture aims at supporting instant and frequent changes, there are a number of challenges discussed below.

2.2. Challenges of Evolutionary Architecture

With the evolutionary architecture, important design decisions can be enforced while the architectural changes are applied to support changing business and execution environments of software systems.

1. To support the evolutionary architecture, unit tests are usually developed as the fitness function. Therefore, the implementation has to be made to prove whether the architectural changes are correct [4]. However, the architectural changes at the coding level are often complex and time-consuming, as it requires an understanding of the current design to reason about the impact of changes[24]. Moreover, it is a repetitive task of making adjustments until the fitness function is satisfied [25]. Therefore, huge effort is required to carry out any changes at the architectural level.
2. Ensuring the correctness of architectural change requires thorough analysis in different aspects. The analysis should cover both functionalities and architectural constraints. Some original functionalities need to be preserved, while changes need to be made to the design to support new functionalities [26]. With the informal sketches of the design model, the analysis is usually conducted manually by simulating scenarios to illustrate how the system behaves at runtime [27]. This kind of analysis could be a time-consuming and daunting task.
3. As the changes need to be deployed incrementally without interrupting the running system, planning the refactoring and deployment remains challenging. An accurate understanding of current architectural design is required to plan the evolution process (i.e., steps) that allows new design configurations to be smoothly incorporated into the software system while the old design configurations can still be used [28].

Moreover, it is hard to determine when the old configuration can be safely cut off from the rest of the system, which evolution steps need to be performed carefully, or when the new functionality is ready. To plan the evolution, architects need to reason about system behaviour at each step of the evolution process [29]. As the planning is usually performed manually, there is a high risk of mistakes being introduced, which causes system failures during the evolution.

3. Architectural Modelling for Changes

Formal modelling is the first step in our approach, as it describes both the structural and behavioural aspects of an architectural design. As a result, we can make an automated verification of the functional properties against the evolving design. When the architectural design requires changes to support new functionalities, an additional model is created to describe the sequence of changes that need to take place.

3.1. Architectural Modelling

Applying architectural changes requires reasoning on both structural and behavioural aspects of software architecture design [30]. Our modelling supports both aspects, i.e., the structural detail is defined in Ontology Web Language (OWL) [31] and the behavioural detail is represented in Wright# [32]. OWL is expressive in describing the relationships among entities, while Wright# is an Architecture Description Language (ADL) used to describe the interactive behaviours of the components. To allow the design of complex system structures, various architectural patterns have been formally described. Furthermore, the associated tools for ontological reasoning and model checking enable the analysis to be performed automatically and effectively.

The model is based on a C&C (Component and Connector) view [33] that consists of architectural elements such as component, connector, role, and port. A component has one or more ports that serve as interfaces to other components. A connector represents connectivity among components, which have roles to represent the interacting parties. A port can be attached to one or more roles to form the interaction among components. To support architectural patterns, types can be specified for architectural elements to define their structural and behavioural details accordingly. These architectural patterns are predefined and can be reused across the different models.

For the structural description in OWL, we applied the modelling presented in [34]. The architectural elements are defined as ontology classes, and the relationship among them is defined as object properties. With the ontology class descriptions, we can create the structural model for a software system by defining the ontology individuals.

Figure 1: LifeNet's current design

Throughout this paper, we use 'LifeNet' as an illustrative running example, whose software architecture design is shown in Figure ?? is a software system that helps patients with dementia related symptoms. Patients wear a wristband called *Lifeband*, which allows them to make a request for emergency service when they need it. The emergency request is received by *SOSGateway*, before it is forwarded to *RequestDispather* to process. This component serves as a central part of the system that finds an emergency centre and a practitioner closest to the patient, through related storage, namely *EMCenter* and *Practitioner*. Then, the notification is sent to the practitioner and the emergency van's driver via *Lifeguard*, a mobile application. *Lifecare* is a mobile application that caregivers use to track the status of the patient. *Patient* is a storage component that manages the medical records that can be updated through the *HospitalMgmt* component.

Listing 1: Structural Model of LifeNet

```
Individual(ex: Lifeband value(ex: hasPort ex: alert))
Individual(ex: EMCenter value(ex: hasPort ex: emaccess))
Individual(ex: soswire value(ex: hasRole ex:  responder))
```

```
Individual(ex: alert value(ex: hasAttachment ex: requester))
Individual(ex: emaccess value(ex: hasAttachment ex: readstorage)
```

We can create ontology descriptions in OWL representing the structural model of LifeNet. The components, connectors, roles, and ports in the design are created as ontology individuals [1]. Listing 1 shows the partial structural model of LifeNet. The notation is according to descripion logic presented in [35]. *Individual* represent an ontology individual with *ex* to specify its name. Each individual may include properties indicated by *value*, which specifies the property name and its value. According to this model, *Lifeband* individual has associated to a port *alert* through *hasPort* property. The alert port is attached to requester role of *soswire*. This ontology description is processed by an ontology reasoner to infer the type of component and connector according to the architectural pattern. For example, *Lifeband* is inferred to be a client of the client-server as it has *alert* port that is attached to the *requester* role. EMCenter is inferred as read-only storage according to the repository pattern. These inferences help identify the architectural patterns according to the structure. Additionally, the inferred pattern also helps us identify the interactive behaviour of the component.

The behavioural model can be defined in Wright# [32]. With Wright#, the behaviour of the component is predefined according to the architectural pattern. Listing 2 shows partial connector definitions. For example, *CSConnector* is for the client-server pattern. The type of connection consists of a definition (*role* statement) to describe the behaviour of a role as a sequentially occurring event. The *system* statement defines how the system is executed to support the functionalities. In the part of the model shown in Listing 2, *soswire* and *dispatcherwire* are defined as *CSConnector* using *declare* statement. The ports are configured by the *attach* statement that defines the attached role and how they are executed. The *execute* statement specified how the system are executed with ports of components to support functionality. The operator || is used to specify that all ports are executed in parallel. More details about Wright# syntax can be found in [32].

Listing 2: Behavioural Model of LifeNet

```
connector CSConnector {
  role requester(j) = process -> req!j -> res?j -> Skip;
  role responder() = req?j -> invoke
```

[1]The OWL model of LifeNet can be found at https://bit.ly/39U8FBy

```
        -> process -> res!j -> responder();}
system LifeNet {
        declare soswire = CSConnector;
        declare dispatcherwire = CSConnector; ...
        attach Lifeband.alert() = soswire.requester();
        attach RequestDispatcher.accept()
                        = dispatcherwire.responder() ...
        execute Lifeband.alert() ||
                        RequestDispatcher.accept() || ...
```

With the definition of behaviour in place, the model can be verified to prove whether the desired behaviour can be achieved. The desired behaviour to verify can be expressed in LTL (Linear Temporal Logic), which can represent system functionalities or the behaviour of specific architectural patterns. Additionally, the verification properties can be specified to describe expected behaviour as liveness properties. For instance, in LTL, we can define a liveness property as $\Box(SrcEvent \rightarrow \Diamond ResEvent)$, where *SrcEvent* is a source event that occurs when the system responds to serve a function and *ResEvent* is a response event that occurs when the system completes responding for a function.

Below is a list of functional properties that are defined for LifeNet. These properties can be explained as follows: E1 – When the emergency is requested, the request is acknowledged, and the wristband's status is switched to alert mode. E2 – After the emergency request is dispatched for processing, the request is acknowledged by the system. E3 – After the request is dispatched to process, the system must find the emergency centre in the proximity of the patient location. E4 – After the emergency staff acknowledges the emergency request (through the Liferescue application), the patient record can be fetched and verified. E5 – After the caregiver acknowledges the emergency request (through the Lifecare application), the patient record can be fetched and verified.

- $E1$ $\Box(SOSGateway.sos.acknowledge \rightarrow \Diamond LifeBand.alert.onalert)$
- $E2$ $\Box(RequestDispatcher.accept.dispatched \rightarrow \Diamond SOSGateway.sos.acknowledge)$
- $E3$ $\Box(RequestDispatcher.accept.dispatched \rightarrow \Diamond EMCenter.emaccess.emaccessed)$
- $E4$ $\Box(Lifeguard.gnotify.acknowledge \rightarrow \Diamond Patient.ptaccess.ptaccessed)$
- $E5$ $\Box(Lifecare.cnotify.acknowledge \rightarrow \Diamond Patient.ptaccess.ptaccessed)$

With the PAT model checker [36], these properties can be verified to

guarantee that the design encoded in the model supports the required functionalities. The verification provides the state trace as a result, describing how components, ports, connectors, and roles interact to serve the property. The model checker only gives one possible state trace that shows how the system satisfies the property. This is acceptable for the purpose of verifying the logical view of architecture design, as there should only be one way that the system logically satisfies the functional property. Listing 3 shows a partial state trace given from verifying the property E1 against the LifeNet model. From this state trace, the LifeBand's *alert* port is called and requests to SOSGateway's *sos* port via the *soswire* connector.

Listing 3: State traces of LifeNet's E1 property

```
init -> LifeBand_alert_onalert
-> LifeBand_soswire_requester_process
-> soswire_req!63 -> soswire_req?63
-> SOSGateway_soswire_responder_invoke   ...
```

The formal verification serves as an early check at the design stage before the next stage of implementation.More details is explained later in Section 4.4

3.2. Evolution Modelling

Our approach aims at automating the planning of frequent incremental system changes over time, which reflects the key focus of the evolutionary architecture. Therefore, the evolution modelling is developed to describe the sequence of functional changes as shown in Figure 2.

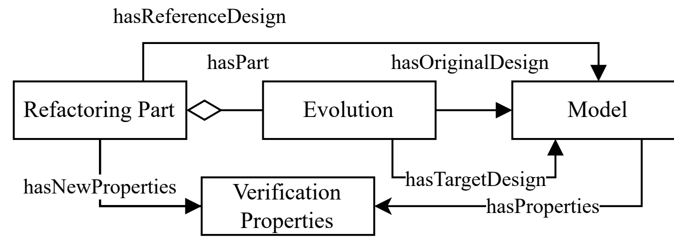

Figure 2: Metamodel of Evolution

The *Evolution* entity is associated to two instance models, namely the original design and the target design. The original design is the current software architecture design, whereas the target design is what the system will evolve into after making changes. An *Evolution* consists of *Refactoring Part*

that represents an incremental phase of functional changes. A refactoring part has a new set of functional properties defined and a reference model as a template describing the structure and behaviour of architectural design to support the new properties. The reference model includes details that are not specific to any system, so it can be reused when the same function needs to be incorporated in other systems. The functional properties defined for the reference model are called the reference properties. In some cases, we may use the original design as a reference model when some part of the original design needs to be replicated to support similar functionality for different purposes (e.g., refer to LifeNet's Change#1 as an example below).

For LifeNet, we define four changes in the evolution process as follows. Change#1: Lifefriend will be developed as an application for family members. Lifecare is used as a baseline to develop Lifefriend with additional features, such as tracking patients' locations and receiving news. Change#2: Family members will be able to make payment on lifefriend for the service subscription fee. Change#3: Instead of querying the emergency centre on our self-managed database, we will integrate with the LifeGrid, a public database that provides countrywide information about emergency centres. Change#4: The lifecare will be obsolete and removed out of service. We have defined these changes as refactoring parts. The reference models of these parts are as shown in Figure 3, where a) is a reference model for the payment gateway to be integrated with Lifefriend according to Change#2 and b) is a reference model for LifeGrid according to Change#3. We will use the original design as a reference model for Changes#1 and #4, as we want to replicate or modify the existing structure (more details are explained in Section 4).

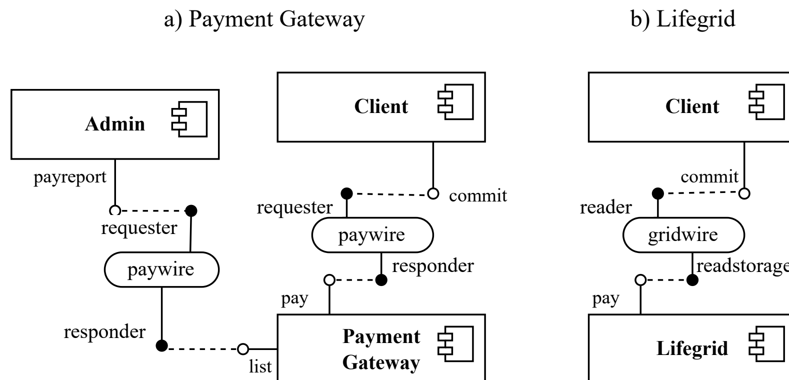

Figure 3: Reference model for LifeNet

As shown below, we have described the system behaviour to support new functionalities as the properties (i.e., new properties of the refactoring part) in LTL. These properties can be explained as follows: Change#1: When the LifeFriend is notified, the patient record, including the location, is accessed. Change#2: The payment is preprocessed on lifefriend before the payment gateway checks it. Change#3: After the request dispatcher component accepts the emergency request, it looks up the closest emergency centre to the patient using LifeGrid. Change#4: When the request dispatcher accepts the request, it will not notify LifeCare anymore.

$N1$ $\Box(LifeFriend.cnotify.acknowlege \rightarrow \Diamond Patient.ptaccess.ptaccessed)$

$N2$ $\Box(LifeFriend.pay.preprocess \rightarrow \Diamond PaymentGateway.commit.checked)$

$N3$ $\Box(RequestDispatcher.accept.dispatched \rightarrow \Diamond LifeGrid.lookup.returned)$

$N4$ $!\Box(RequestDispatcher.accept.dispatched \rightarrow \Diamond Lifecare.cnotify.acknowledge)$

The refactoring part and newly defined properties aid in the planning of design refactoring. The property determines how the structure should be changed and verifies whether the refactored design can support the new functionality.

4. Architectural Evolution Planning

The architectural evolution is usually planned based on agile methodology, as it is effective in developing software systems that are continuously changing [37, 38]. According to agile methodology, the changes should be made to the software system through short iterations. Our planning approach is therefore developed based on this principle.

4.1. Automated Planning Process

The evolution planning is automated with AI planning technique using PDDL4J [39]. AI planning [40] is a common technique to explore the state space using a specified strategy and find the sequence of actions to achieve a defined goal. In our approach, AI planning is used to plan a sequence of refactoring actions to achieve the goal, where the configuration of the refactored design supports new functional properties. Without an automated approach, these steps need to be determined manually, as we highlight the challenges in Section 2.2. The overall planning process is shown in Figure 4, which can be explained as follows.

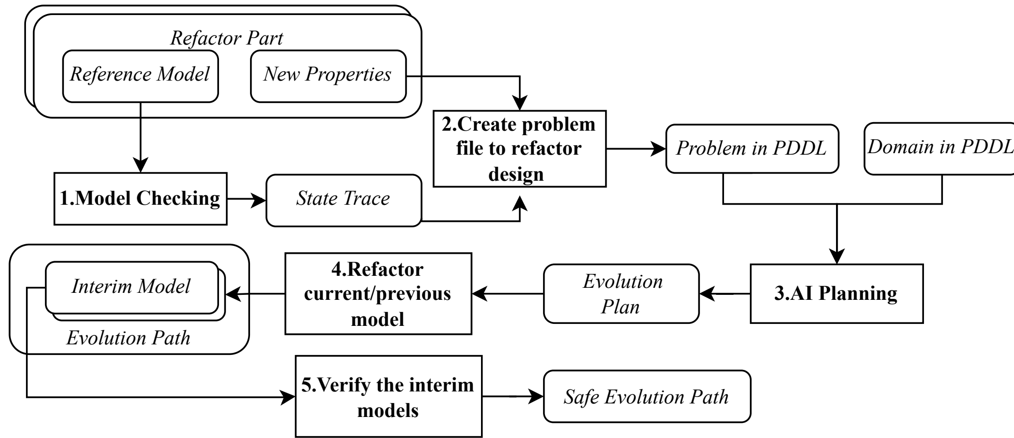

Figure 4: Overall evolution planning process

Firstly, the reference model and its functional properties (called reference properties) inside a refactoring part are processed by the model checker to generate a state trace that describes how system responses are made in order to achieve the properties. Secondly, the new functional properties and state traces serve as an input to create a problem description in the PDDL, which is further explained in the Section 4.2. Thirdly, the AI planner processes the problem description and the domain description to generate the evolution plan, which will be explained in detail in section 4.3. An evolution plan is generated according to the initial states and the goals defined in the problem description. It consists of the steps to apply structural changes to the architectural design. Fourthly, the evolution path consisting of the interim models is generated by refactoring the architectural design corresponding to each step in the evolution plan. As a result, the interim models are generated according to the evolution plan. Lastly, we verify the set of functional properties against the interim models to determine a safe evolution path.

AI planner requires domain description to be defined in PPDL [2]. We have used PDDL 3.1 [40] in our work. In the domain description, we declare the problem domain with object types, predicates, and actions. This part of the description is generic, and users do not need to recreate it to plan the evolution of a specific software system. Listing 4 shows a partial domain description, where component, connector, port, and role are defined as types

[2]PDDL specification can be found at http://pddl4j.imag.fr/pddl_tutorial.html

for the architectural refactoring domain. The predicates define the states of types, which relate to the architectural configuration, such as *connect*, *hasport*, *hasrole* and *attach*, as well as the state of a component or connector, such as *enabled-component* and *disabled-component*.

Listing 4: Domain Description of Refactoring

```
(:types component connector interface - object
        port role - interface)
(:predicates (connect ?con - connector
              ?com1 ?com2 - component)
  (hasport ?com - component ?prt - port)
  (hasrole ?con - connector ?rle - role)
  (attach ?com - component ?prt - port
      ?con - connector ?rle - role)
  (enabled-component ?com - component)
  (disabled-component ?com - component)
  (enabled-connector ?con - connector)
  (disabled-connector ?con - connector) )
```

The actions are defined in the domain description to specify the common actions that help to refactor the architectural design. Due to space limit, we shows partial actions[3] in Listing 5. These actions specify what needs to be performed on the system. For example, *setup-component* is to prepare a new component to function, the *remove-component* is to disable the component, *remove-connector* is to disable the connector, and the *establish-link* attaches a connector's role to a port of a component to form connectivity.

Listing 5: Actions in Domain Description

```
(:action setup-component
    :parameters (?com - component)
    :precondition (disabled-component ?com)
    :effect (enabled-component ?com))
(:action remove-component
    :parameters (?com - component)
    :precondition (enabled-component ?com)
    :effect (disabled-component ?com))
(:action setup-connector ...
(:action remove-connector ...
(:action establish-link ...
```

[3]The full domain description can be found at https://bit.ly/3GhiVjE

The AI planner parses the domain description and the problem description. The problem description includes the defined initial states and goals, which are used to determine the sequential steps to achieve the goals. In our work, each refactoring part has initial states and goals defined in the problem description in order to determine the sequential steps of architectural refactoring.

4.2. Refactoring via Referencing

Our approach supports three types of refactoring, namely, extension, substitution, and obsoletion. The extension is used when new functional properties are introduced and require new architectural elements added to the original architecture design to support. The configuration obsoletion is used when the original properties need to be purged from the system, which causes the removal of some architectural elements. The substitution is used when the original properties need to be completely replaced with the new properties and the architectural configuration needs to adapt to provide support. This means that the replaced properties and the new ones will not be supported at the same time; otherwise, extension and obsoletion can be applied. Each refactoring part requires the reference model to be defined. The reference model can be selected to guide how the configuration will be refactored, so it must include the architectural elements that support new or replaced properties.

To plan the architectural evolution, the problem description must be created to define the initial states and goals. The initial states represent the structure of the architectural design prior to refactoring, whereas, the goals represent the changed design configurations that need to be made on the current design. The current design can be the original design or the prior interim design while the system is evolving. We have developed algorithms to create the problem description. These algorithms aim at finding the initial state and goals, which can be achieved by inspecting the state traces given from verifying the reference properties against the reference model. The state trace indicates the sequence of how components and connectors interact to serve the new functions, so it guides how the design should be refactored. The rest of this subsection presents three types of refactoring and the algorithms to generate the problem description.

4.2.1. Configuration Extension

This type of refactoring inserts additional architectural elements into the system, such as new components and connectors. The inserted elements associate with the rest of the system to support new functionality. For example, according to Change# 3 of LifeNet, a payment gateway will be inserted into the system to handle customers' payment functions. Algorithm 1 below demonstrates how the problem description is generated for the configuration extension.

Algorithm 1 Generating problem description for configuration extension

```
1: FuncProp is a new functional property
2: RefProps is reference properties
3: for RefProp ∈ RefProps do
4:     RefProp.statetrace is state trace of this prop
5:     for Event ∈ RefProp.statetrace do
6:         if Event is an event triggerd by role then
7:             if connector that triggers Event ∄ current design then
8:                 conn ← new Connector according to connector that triggers Event
9:                 disable-connector conn in init
10:            SrcComp ← ProcEvent(backtrace(Event), RefProp)
11:            attach conn with SrcComp in goals
12:            TgrComp ← ProcEvent(forthtrace(Event), RefProp)
13:            attach conn with TgrComp in goals
14: procedure ProcEvent(event, refFunc)
15:     comp is a component to return
16:     if event is a port event then
17:         if event is a response event of refFunc then
18:             TgrEvent ← response event of FuncProp.
19:         if component that triggers TgrEvent ∃ current design then
20:             comp ← component that triggers TgrEvent
21:         else
22:             comp ← new Component according to component that triggers TgrEvent
23:             disable-component comp in init
24:     return comp
```

Algorithm 1 takes a new functional property and the reference properties (i.e., verification properties in the reference model) as input. The loop at line#3 processes each reference property by obtaining its state trace to be processed by the inner loop at line#5. The inner loop takes each event in the state trace and finds an event triggered by a connector's role that does not yet exist in the current model. If satisfied, a new connector is defined

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

and set to disable at the init state (with *disable-connector* predicate). After that, at line#10, the algorithm backtraces the state trace (using the *backtrace* function) until the event triggered by the component is found. *ProcComp* subroutine checks if the event is a source event for the reference property. If it is, we replace it with the source event of the new property. *ProcEvent* also checks if the component triggering this source event exists in the current model. If it does not exist, a new component is defined and set to disable at the initial state at line#25 using the *disable-component* predicate. Then, at line 13, the algorithm finds the next event in the state trace that is triggered by the component(using *forthtrace* function). This event is a target event that responds to support a function. We apply *ProcEvent* subroutine to determine whether the component triggering this event should be defined and set to disable at the init state. Lastly, a new connectivity is set as the goal with the *attach* predicate.

For LifeNet, we apply configuration extensions to Change#1 and #2 (as previously explained in the Section 3.1). According to Change#1, the lifefriend will be inserted into the system. This change uses the original design as a reference model, as it aims to initially replicate the function of lifecare. The problem description for this change is generated as shown in Listing 6 (note that the *ref* is added as the postfix of all elements that are added to the design for tracking and testing purposes). The *init* specifies the initial states, which includes certain part of the current design involving to the functional change and the new elements initially setting to disable state. For example, lifefriend (*LifeFriendref*) is defined as an object and set to disable at the initial state. The *goal* specifies that the LifeFriend will be connected to the patient component.

Listing 6: Problem Description for lifefriend

```
(:objects LifeFriendref Patient – component ...
(:init
(disabled–component LifeFriendref)
(hasport LifeFriendref cnotifyref)
(hasrole ptwire readstorage–ptwire)
(enabled–component Patient)
(enabled–connector ptwire)) ...
(:goal (and
  (attach LifeFriendref cnotifyref ptwire reader–ptwire)
  (attach Patient ptaccess ptwire readstorage–ptwire))))
```

For Change#2, the problem description is generated, as partially shown

in Listing 7. The new component payment gateway (*PaymentGatewayref*) is defined and initially set to a disable state. It can be seen that the goal specifies that the LifeFriend will be linked to the payment gateway component.

Listing 7: Problem Description for payment

```
(:objects LifeFriendref PaymentGatewayref - component...
(:init
(disabled-connector paywireref)
(disabled-component PaymentGatewayref)...
(:goal (and
(attach LifeFriendref payref paywireref requester-paywireref)
(attach PaymentGatewayref commitref
        paywireref responder-paywireref))))
```

4.2.2. Configuration Substitution

Configuration substitution handles refactoring when certain elements in the existing design need to be immediately replaced with the new elements. The algorithm 2 is designed to generate the problem description for this type of refactoring. Similar to Algorithm 1, it processes the new functional properties and reference properties defined in the refactoring part. The inner loop at line #5 iterates through the events in the state traces. Lines #6-7 check whether the connector that triggered the event exists in the current design. If it exists, a new connector is defined and initially set to disable (at line #9), while the old one is set to disable as a goal (at line #10). Lines #10-12 trace the states back to where the event is triggered by the component and use *ProcEvent* subroutine to process that event. This subroutine checks whether the component that triggered the source event exists in the current design. If it does not exist, a new component will be defined and set to disable at the initial state. The same routine is applied to the target event that is fetched by forward tracing the states. At last, a new goal of connectivity is established with the *attach* predicate (at line #12 and #14).

Change#3 is applied to the system with the configuration substitution, such as when the 'LifeGrid' replaces the 'Emcenter' component in the running example. The problem description is generated, as partly shown in Listing 8. The lifegrid (*LifeGridref*) is defined as a component and initially set to disable. A new connector, *emwireref* is defined and initially set to disable. The goal specifies that *emwire* connector will be disabled. Eventually, the request dispatcher component will be connected to the lifegrid to query data.

Algorithm 2 Generating problem description for configuration replacement

1: *FuncProp* is a new functional property
2: *RefProps* is reference properties
3: **for** $RefProp \in RefProps$ **do**
4:     *RefProp.statetrace* is state trace of this prop
5:     **for** $Event \in RefProp.statetrace$ **do**
6:         **if** *Event* is an event triggerd by role **then**
7:             *refconn* ← *connector* that triggers *Event*
8:             **if** refconn ∃ current design **then**
9:                 conn ← **new** Connector according to *refconn*
10:                 **disable-connector** *conn* in **init**
11:                 **disable-connector** *refconn* in **goal**
12:             SrcComp ← ProcEvent(backtrace(*Event*), RefProp)
13:             **attach** *conn* with *SrcComp* in **goals**
14:             TgrComp ← ProcEvent(forthtrace(*Event*), RefProp)
15:             **attach** *conn* with *TgrComp* in **goals**
16: **procedure** PROCEVENT(*event*, $refFunc$)
17:     *comp* is a component to return
18:     **if** SrcEvent is a port event **then**
19:         **if** *event* is a source event of *refFunc* **then**
20:             SrcEvent ← source event of *FuncProp*
21:         **if** component that triggers *SrcEvent* ∃ current design **then**
22:             *comp* ← component that triggers *SrcEvent*
23:         **else**
24:             *comp* ← **new** Component according to component that triggers *SrcEvent*
25:             **disable-component** comp in **init**
26:     return *comp*

Listing 8: Problem Description for lifegrid

```
(:objects LifeGridref RequestDispatcher - component...
(:init
(disabled-connector emwireref)
(disabled-component LifeGridref)
(enabled-connector emwire)...
(:goal (and (disabled-connector emwire)
(attach RequestDispatcher accept emwireref reader-emwireref)
(attach LifeGridref lookupref emwireref readstorage-emwireref)))
   )
```

4.2.3. Configuration Obsoletion

Configuration obsoletion is a type of refactoring that disconnects the obsolete elements from the rest of the system. Algorithm 3 below helps generate

the problem description for this scenario. The loop at line #5 searches for the connectors involved in the state trace and disables them. After that, the loop at line #10 finds the components in the model that will no longer be connected to the rest of the system and disables them.

Algorithm 3 Generating problem description for configuration obsoletion

1: *RefProps* is properties to purge
2: **for** $RefProp \in RefProps$ **do**
3:     *RefProp.statetrace* is state trace of this prop
4:     **for** $Event \in RefProp.statetrace$ **do**
5:         **if** *Event* is an event triggerd by role **then**
6:             *conn* $\leftarrow$ connector that triggers *Event*
7:             **if** *conn* $\exists$ current design and no other attachment **then**
8:                 **disable-connector** *conn* in **goals**
9: **for** $comp \in$ all components in Model **do**
10:     **if** *comp* will have no connection to other component **then**
11:         **disable-component** *comp* in **goals**

For LifeNet, the Change#4 is applied using this type of refactoring, as the lifecare component will be obsolete. Listing 9 below shows the partial problem description, in which it can be seen that the Lifecare component will be disabled. Note that the connector *guardnotiwire* is still used by other components such as Lifeguard; hence, it will not be disabled in this case.

Listing 9: Problem Description for lifecare

```
(:init
(enabled-component lifecare)...
(enabled-connector guardnotiwire)...
(:goal (and
(disabled-component lifecare))))
```

4.3. Generating Evolution Plan

After the problem description is generated, it is processed together with the domain description to generate the plan to refactor the architectural design. We use the PDDL4J tool as an AI planner to generate the evolution plan. The enforced hill-climbing is used as the planning strategy without any modification, as it is a default strategy provided by PDDL4J. The generated plan consists of a sequence of actions that are defined in the domain description, such as, *setup-component*, *remove-component*, *establish-link* and

so on. These actions identify how software engineers apply the architectural changes to support new or updated functions.

The refactoring plan in Listing 10 is for Change#2 of the LifeNet example. It is generated by processing the domain description and the problem description, as partly shown in Listing 7. This plan can be explained as follows: Firstly, a new connector paywire (*paywireref*) is setup. This connector will help connect the Lifefriend with the payment gateway. Secondly, a new component *paymentgatewayref* (payment gateway) is setup. These setup actions will make the component and connector enabled and ready to establish connectivity. Thirdly, the link is established between the *paymentgatewayref* and the *paywireref* connector. Lastly, the link is established to the other end between the *paymentgatewayref* and the *lifefriendref* component (Lifefriend).

Listing 10: Generated for payment

```
Step 0: (setup-connector paywireref) [1]
Step 1: (setup-component paymentgatewayref) [1]
Step 2: (establish-link paymentgatewayref commitref
                paywireref responder-paywireref) [1]
Step 3: (establish-link lifefriendref payref
                paywireref requester-paywireref) [1]
```

With the generated plans to refactor the design, we sequentially connect the plan of each refactoring part together as the evolution plan. A sample of LifeNet's evolution process is shown in Figure 5. The circles represent the models of design, while the rectangles represent the refactoring action according to the generated evolution plan. The evolution process starts with the original design and ends with the target design, where all changes have been successfully applied. According to the changes in LifeNet, the steps 1-2 are for Change#1, steps 3-6 are for Change#2, steps 7-11 are for Change#3 and step 12 is for Change#4. During the evolution, the models of interim design (*int1* to *int11*) are created to illustrate how the architecture will be after each step of change has been applied. For LifeNet, eleven models of an interim design were created.

This evolution plan serves as a guideline to show how the software system will evolve into the target design. However, as the changes are made, the system interruption should be kept to a minimum. Therefore, we need to be able to determine when the system is ready to handle new functionalities by compiling the models with the required functional properties.

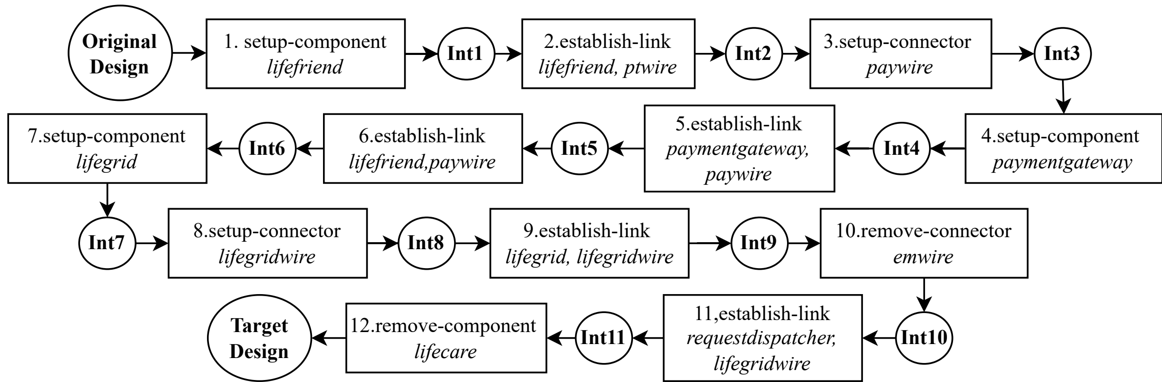

Figure 5: LifeNet's Evolution process

4.4. Executing Formal Fitness Function

Fitness function in the evolutionary architecture helps to ensure that the original functionalities are preserved while the system evolves to support new functionalities. In our approach, we used formal verification as the fitness function. As the functional properties are formally defined in LTL, the model checker can be used to automatically verify the properties against the behavioural model of the software architecture design. We can verify the original and new properties against the evolution path consisting of the interim designs and a target design. This verification helps to determine when the system is ready to support a new functional property and when it will be interrupted due to architectural changes.

Table 1: Verification Result of LifeNet's Evolution

Prop#	Int1	Int2	Int3	Int4	Int5	Int6	Int7	Int8	Int9	Int10	Int11	Tgt
E1	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲
E2	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲
E3	▲	▲	▲	▲	▲	▲	▲	▲	▲	△	△	△
E4	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲
E5	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	△
N1	△	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲	▲
N2	△	△	△	△	△	△	▲	▲	▲	▲	▲	▲
N3	△	△	△	△	△	△	△	△	△	△	▲	▲
N4	△	△	△	△	△	△	△	△	△	△	△	▲
Iteration	#1	#2	#3	#4	#5	#6	#6	#7	#8	#9	#9	

The verification results of the evolution path of LifeNet is shown in Table 1. The *Prop* column shows the identification numbers of properties that were

previously explained in Section 3.2. E1 to E5 are the original properties, and N1 to N4 are the new properties. The column *Tgt* shows the verification results of the target design. The column *Int1* to *Int11* show the verification result of the interim design according to the evolution plan in Figure 5. △ indicates the invalid result, whereas ▲ indicates the valid result. The *iteration* row shows the iteration number as the system evolves. It is clear that the system can support the original functional properties E1 and E2 throughout the evolution process, as they have been demonstrated to be valid on all interim models and the target model.

With the evolution path and the verification results, we identify where interruption may occur (highlighted in blue in Table 1). These interruptions are caused by the incomplete connectivity among components. For example, step 6 attaches the Lifefriend to the connector, while the connection is complete in step 7 when the payment gateway is attached to the other end of the connector. Therefore, step 6 and 7 need to be performed together in the same iteration to prevent the interruption. The interruption can also happen when the configuration substitution is applied. For example, N3 will replace E3 for Change#3. The property E3 is proved to be invalid since the interim model #10 after the *emwire* connector is disabled at step 10. However, in the interim model #11, N3 is shown to be valid. This means that the user won't be able to use the function during these two steps. Therefore, steps 10 and 11 have to be performed together in the same iteration, as the system will be interrupted between these steps. For the refactoring that is configuration obsoletion, the new property must be proved valid, while its opposite property is proved invalid. The property N4 is proved to be valid on the target design after the Lifecare is removed at step 12. This change consequently makes property E5 invalid, as no more notifications can be sent to the Lifecare at this point. The steps that cause no interruption can be performed individually as an iteration. For LifeNet, we have nine iterations toward the target design, as shown in the last row of Table 1.

With the verification results of the interim design and target design, we can determine the following: Firstly, we can check whether the original functionalities are preserved throughout the evolution process. Secondly, we can identify when the new functionalities are ready to be delivered to the user. Lastly, we can pinpoint when the interruption may occur and plan to mitigate it by performing the refactoring steps together in the same iteration. Hence, we can achieve a safe and sound pathway for the desired evolution. Grouping the refactoring steps in the plan helps to mitigate the risk of restor-

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

ing the system when failure occurs. The fewer refactoring steps performed, the simpler the system restoration.Therefore, the interruption interval in the generated plan is an important constraint that should be minimal. We evaluated the generated plan given by different planning strategies to find the strategy to generate the best evolution plan in Section 6.

5. Implementation

The formal techniques we used in our approach are difficult for software engineers to apply because few have prior knowledge of them.To resolve this, we developed our approach on top of ArchModeller [20], a tool framework that we have been developing to support the modelling and verification of software architectural designs. This section presents an overview of tool development to support evolution planning. In addition, a use case is presented to demonstrate how software engineers can use the tool in practice.

5.1. Tool Development

The extension of ArchModeller has been developed to aid the evolution planning of functional changes, as shown in Figure 6 (the extension is highlighted in grey). The extension includes new diagrams, such as change configuration and evolution process, to describe the functional changes. In [20], the AI planner has been applied to find the migration steps to new architecture patterns, while preserving the original functional properties. This paper presents a different domain description, which involves reconfiguring architectural elements to support new functional properties. A new domain description is created, as explained in Section 4 to describe the actions that support functional changes at the architectural level.

As most software engineers do not have knowledge of formal modeling, such as OWL and Wright#, used in our approach, ArchModeller is provided to allow engineers to draw the software architecture design model as a graphical representation and seamlessly conduct the verification. The tool is able to automatically convert the design into OWL and Wright# models in order to perform the verification by the *Structural verifier* and *Behavioural verifier*. This tool supports three major diagrams, namely the architecture model, the change configuration, and the evolution process. These diagrams can be explained as follows: 1) the architecture model illustrates the architectural design in C&C view; 2) the change configuration visualises the sequence of refactoring parts to be applied; and 3) the evolution process presents the

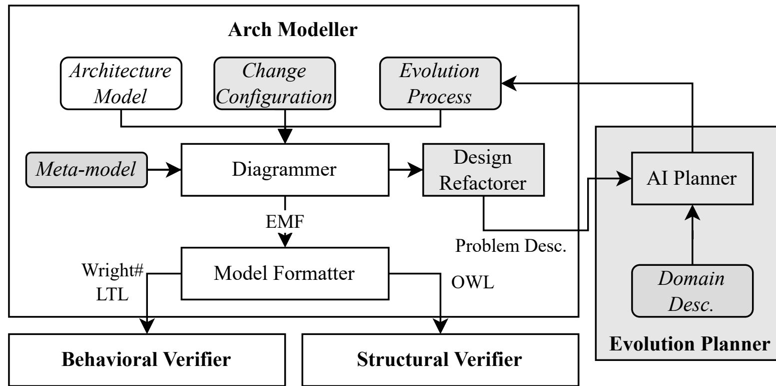

Figure 6: ArchModeller Overview

sequential process as generated by the *AI planner*, similar to what we presented in Figure 5. *Diagrammer* tool uses the meta-model as previously presented in Section 3.1 to support the modelling of architectural design and the changes to be applied.

The refactoring algorithms presented in Section 4.2 have been implemented as the *Design Refactorer*. This module generates the problem description in PDDL. The evolution planner module[4] takes the problem description and the domain description as input to generate the evolution plan. The diagrammatic tool's evolution path visualises the plans and interim models as the sequence of changes to be performed on the system.

5.2. Use Case

With the evolution planning feature built into the ArchModeller, software engineers can model the software architecture design and plan the evolution process seamlessly. Figure 7 shows the steps of how it can be used in practice. This process has two stages, namely modelling and planning. In the first stage, the original architecture design is modelled as a diagram in the C&C view, in which the functional properties are defined to describe the existing system functionalities. After that, the change configuration is created by defining the refactoring parts as a sequential process. Each refactoring part has the new functional properties and a reference model specified.

[4]The source code of Evolution Planner can be found at https://github.com/cnachamfu/evolution-planner

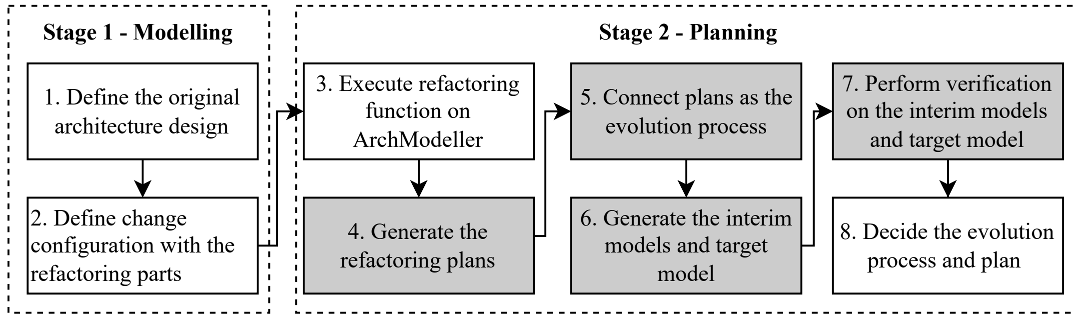

Figure 7: ArchModeller Use-case

With the change configuration in place, the planning stage is mostly automated by the tool, as the highlighted grey box in Figure 7 indicates such automated steps. The tool calls the evolution planner module to generate the evolution plans and connect them as a sequential evolution process. The tool then processes the evolution process to generate the interim models and the target model, according to the steps in the evolution plan. Finally, the user can determine the iteration of evolution by verifying the set of functional properties against the interim models and the target model.

The verification results also help the user make a decision in the following ways: Firstly, the original properties to be reserved should be supported throughout the evolution process. If an existing property could be lost during the evolution, the user needs to reconsider the reference model and adapt the reconfiguration of design to preserve it. Secondly, the user can plan the system's operation based on when the new functionality will be delivered by setting a timeframe for each iteration. Lastly, when the steps causing interruption to the user's operations are identified, these steps should be performed outside of service hours or during periods with minimal traffic.

6. Evaluation

In order to evaluate our proposed solution, we have conducted experiments to answer the following research questions.

- **RQ1**: How rigorous is our approach to refactoring the architectural design to support new functional properties?

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- **RQ2**: How effective are the evolution plans generated by different planning strategies?

- **RQ3**: What are the factors that impact the execution performance of evolution planning?

6.1. Experimental Setup

To the best of our knowledge, there is no standard benchmark to evaluate the evolution planning approach. Therefore, we use the software architecture design of six software systems in this evaluation, namely Life Net(LN) [41], Ride Sharing (RS)[42], AgriDigital(AD)[43], Sock Shop(SS) [44], Hotel booking (HB) [45], and Reimbursement Management (RM) [46]. Details on size and complicities of these systems are provided in Table 2. The total C&C# column shows the model size by the number of components and connectors. Existing Func.# column shows the number of original functional requirements. These systems were selected because the documentation of their software architecture designs was available. It also includes the records of functional changes that we can use to compare its evolution plan with that generated by our approach.

Table 2: Subject Systems

System	**total C&C#**	**Original Func.#**
Life Net (LN)	22	5
Ride Share (RS)	16	5
Agri Digital (AD)	23	4
Sock Shop (SS)	21	7
Hotel Booking (HB)	18	7
Reimbursement Management(RM)	15	5

Our evaluation was conducted according to the following steps: Firstly, we assigned a group of software engineers to create the model of architecture design using ArchModeller [5] based on existing design documents. These engineers are selected based on their knowledge of system implementation,

[5]The models can be found at https://github.com/cnacha-mfu/evolution-models

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

and they have been trained in the modelling notation used in our approach. The main functionalities were specified in the functional properties, which were later verified to prove that they were valid against the design model. Secondly, the change configurations were defined according to the record of functional changes. The new functional properties are defined in this step to reflect the functional changes. In addition, reference models are created as necessary. Thirdly, the evolution planning function was executed to create the evolution plan and path consisting of the interim models and the target model. Lastly, we performed the verification of properties on the interim models and target models in order to obtain the safe evolution path. To answer the research questions, we gathered the information to analyse as follows:

RQ1: The verification results of the target models can prove whether the refactoring was performed correctly. The target model should have both the new and the original properties (together called fitness properties) to preserve. Therefore, the precision rate is calculated by the formula below to measure the rigorousness of refactoring. $Prop_{val}$ represents the number of fitness properties that are proved correct, while $Prop_{inv}$ is the number of fitness properties that are proved wrong. Furthermore, the correctness of design refactoring was evaluated to identify the limitations of our refactoring approach. This has been achieved by identifying and investigating the refactoring parts whose new functional properties are proven to be valid on the target design, but whose refactored design configuration does not meet the architect's expectation (referred to as$Prop_{abn}$) due to specification errors. We have manually checked the target design to identify the cause of $Prop_{abn}$ results.

RQ2: Different planning algorithms often generate the solution as different sequences of actions. The sequence of refactoring actions calculated by the AI planner is significant to the evolution planning, as the proper sequence will guide the evolution to proceed incrementally and safely. We experimented with three planning strategies provided by the PDDL4J tool. These strategies are Enforced Hill-Climbing (EC), A-Star (AS) and Breadth-First Search (BF). The following metrics have been defined to measure the efficiency of the generated plan:

- *num* - The total number of steps generated by the planner. As each step in the plan requires time and resources to apply the change to the architectural design. The less *num* is, the more efficient the plan is.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

- *interrupt* - The number of system interruptions. The system interruption could occur for the number of reasons, as previously explained in Section 4.4. The fewer *interrupt* there are, the more efficient the plan is.

- *interval* - The number of interval steps taken during the interruption. The interval steps during the interruption need to be performed at once (in the same iteration). The plan is more efficient when the *interval* are smaller. Because making many architectural changes at once increases the likelihood of system failure, which is difficult to recover. The fewer the *interval* there are, the more efficient the plan is.

RQ3: When the planning function was executed on the planner module, the times taken to generate such an evolution plan for each subject system were recorded by a script[6] to measure the computational performance. We performed the planning 10 times for each model to calculate the standard deviation and the standard error in order to statistically analyse the results to answer this question. The EC algorithm was used to search for the optimal plans. The tests were carried out on an Intel Core i7-7500U CPU running at 2.7 GHz with 8.00 GB of RAM.

6.2. Results & Discussion

The result of functional rigidity after the refactoring can be found in Table 3 below. The $Prop_{val}$ shows the number of fitness properties that were verified against the target model. As the $Prop_{val}$ indicates, all fitness properties are proved to be correct on the target model for every subject system. It can

Table 3: Evaluation Results of Functional Requirements

System	**Fitness Props#**	$Prop_{val}$	$Prop_{inv}$	$Prop_{abn}$
LN	7	7	0	0
RS	9	9	0	2
AD	6	6	0	1
SS	9	9	0	0
HB	8	8	0	1
RM	8	8	0	0

[6] This script can be found at https://bit.ly/3ctHnUm

be observed that some models have $Prop_{abn}$, such as those of RS, AD, and HB. These results infer that even though some properties are proved correct on the target design, the refactored design configuration, however, does not meet the architect's expectation. After we carefully investigated the cases that produced these results, we found three cases where the refactoring could not be performed as expected. These cases relate to how the new functional properties are defined for the refactoring parts. They can be explained as follows:

- The reference design does not have the trigger event defined in the new functional property, and the original design contains the response event. After refactoring, the target design contains an isolated group of components and connectors according to the reference design, without any connectivity to the rest of the system.

- The original design does not have the trigger event defined in the new functional property, but the response event is found in the reference design. This case only occurs when the reference design is not the original design. After refactoring, an isolated group of components and connectors can be found in the target design.

- The reference design contains both response events and trigger events defined in the new functional property. Hence, the target design contains an isolated group of components and connectors, similar to the ones found in the reference design.

These three cases are invalid functional properties since they are not correctly defined to apply changes to the original design. To resolve these limitations, we can implement a checking procedure to rule out these cases when the user defines the new functional property. Nevertheless, we can conclude to answer RQ1 that our refactoring approach is sufficiently accurate in producing the architectural design to support new functional properties, as the precision rate is generally 1.0. However, there could be mistakes in defining the new functional properties, which can be resolved by performing the prechecking step to rule out the invalid functional properties.

To answer RQ2, we have gathered the metrics num, $interrupt$ and $interval$ from the generated evolution plans. These metrics are plotted in the radar charts, as shown in Figure 8. From the radar chart, the efficiency of the generated plans can be compared by the size of the triangles. The smaller

the triangle is, the better the evolution plan is. Breadth-first search could generate a plan similar to that of A-Star for most of the models, except for LifeNet whose line is behind that of A-Star. It can be assumed that the A-Star (AS) (the smallest triangle) can generate the best plan for all systems, while the Enforced Hill-Climbing (EC) generally generates the worst plans compared to the other strategies used in this evaluation. This result suggests that the A-Star should be used as the planning strategy to generate the evolution plan.

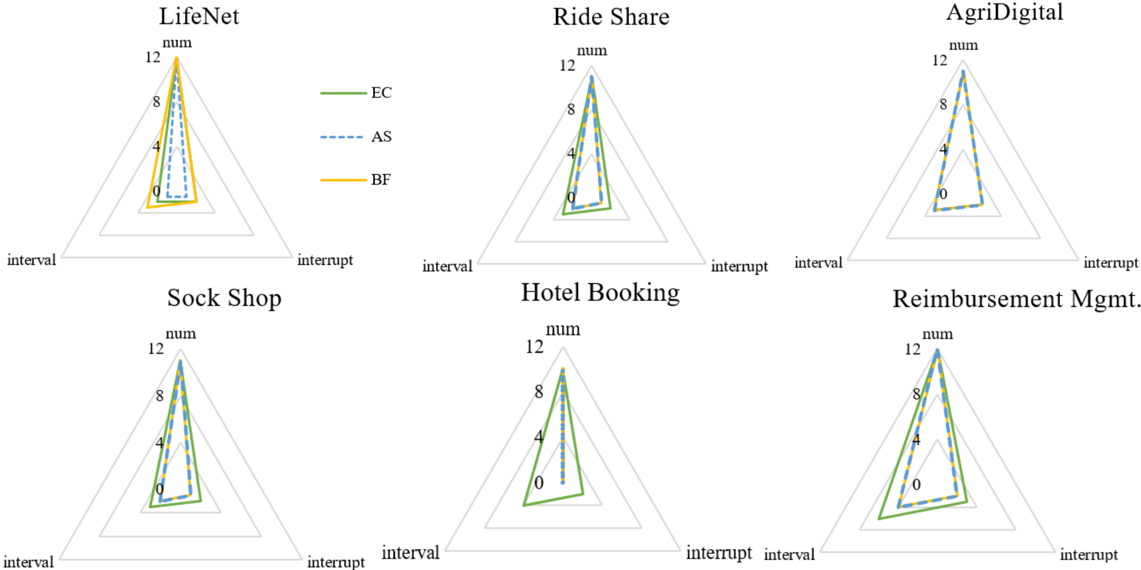

Figure 8: Efficiency of generated plan

We have analysed the processing time of generating the evolution plan to answer RQ3. The processing time is plotted as a graph in Figure 9, where the Q1 median and Q3 median are shown. The error deviations are also calculated and plotted. The time taken by the AI planner to generate the evolution plan is generally around 10 to 40 milliseconds. The processing times are insignificantly different among the subject models. We can conclude that the execution time is reasonable for practical usage. However, the size of the model may impact the time taken for the generation of the plan, as AI planning relies on exploring the state space, which can be large on some models. In our approach, we mitigate this by generating the plan for

some parts of the design that need refactoring, which is more efficient than generating the plan with the design model of the whole system.

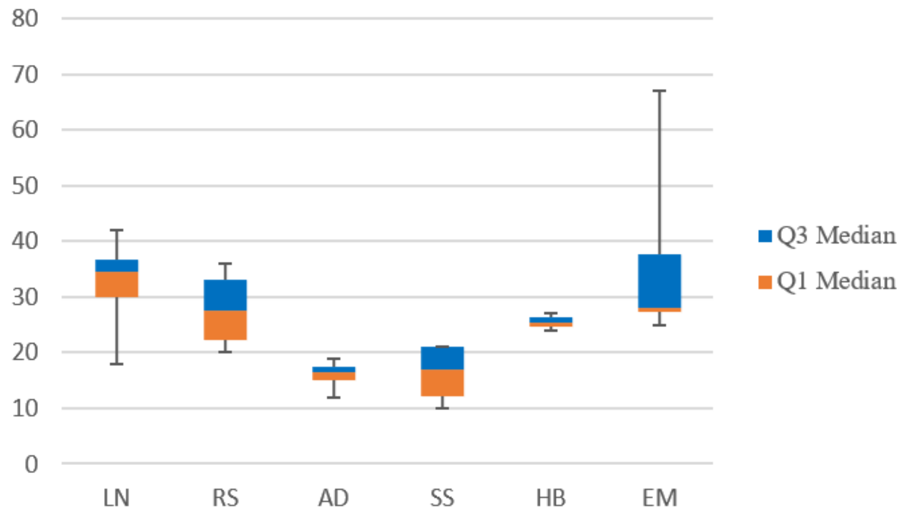

Figure 9: Process time of plan generation

6.3. Threats to Validity

There are limitations in our evaluation, which can be discussed as internal and external threats to validity.

Internal treat: In our evaluation, software engineers are responsible for creating the model of architecture design using ArchModeller based on existing design documents. As there is a lack of updated documentation on the software architecture design, such modelling relies on software engineers' interpretations of the design to create models. As the rigorousness of our approach relies on the model, there can be a gap between the model and the ground-truth architecture of the software system. This threat has been minimised by peer review among software engineers. As an engineer is assigned to create the model of the system for which he or she has knowledge of its implementation, we give the model to other engineers who have knowledge of the same system. Another internal treat is with the subject systems used in our evaluation, as there are a limited number of systems that have documentation of architectural design available. We have ensured that these subject systems have all three kinds of refactoring that our approach sup-

ports. Therefore, the evaluation results only represent the effectiveness of our approach against the design models of these subject systems.

External threat: lies with the architectural design of the subject system and reference architectures used in the evaluation. These architecture designs are mostly based on event-sourcing, client-server, publish-subscribe, and repository patterns. Therefore, the evaluation results are limited to the software architecture design that applies these patterns.

7. Related Work

The related works are discussed in this section. As our approach focuses on design refactoring and evolution planning, these two groups of works are discussed as subsections below.

7.1. Design Refactoring

Many researchers have focused on automating the refactoring process through various kinds of techniques. The search-based algorithm is a prominent technique proposed to find an optimal refactoring path. For example, Rizzi et al.[5] proposed a search-based algorithm to identify an architecture smell in the source code and propose a refactoring path that the developers could follow. However, the approach focuses only on resolving a particular architecture smell called cyclic dependencies. Lin et al.[6] presented an interactive system that recommends a series of refactoring steps toward the desired target design. Even though the refactoring recommendation can help the developer refactor the system more efficiently, the recommendation heavily relies on user feedback, which could be biased.

As search-based techniques generate a large number of possible refactoring paths, other techniques have been applied to find the optimal paths. Fadhel et al.[7] combined the genetic algorithm with the heuristic search algorithm to generate a possible series of refactoring steps for the design model. The approach could suggest optimal changes in the model. However, it requires the original and target models to plan the refactoring, unlike our approach, which can generate the target model with the original and reference designs as the inputs. The evolutionary algorithms such as the multi-objective optimisation have been proposed to improve software architecture design with respect to various quality attributes, e.g. performance [8] and availability[9]. Cortellessa et al. [10] applied a multi-objective optimisation approach to identify performance-critical actions and select the

optimal configuration of the software architecture design. Even though these approaches work well with evolutionary algorithms, they only focus on improving a particular quality attribute. Other approaches focus on refactoring to improve security. For example, Alshayeb et al. [11] applied a genetic algorithm to detect vulnerabilities and suggest refactoring in the UML sequence diagram. The approach relies heavily on a set of rules, which need to be maintained to support new vulnerabilities. Holmes and Zdun [12] proposed an automated approach to improve security and availability through architectural refactoring. However, the approach focuses on the security analysis of the cloud structure; therefore, many details about the cloud infrastructure need to be included in the model.

To the best of our knowledge, none of these approaches guarantees the preservation of the original functionalities or takes into account the functional properties as those presented in our approach. Moreover, most existing techniques do not support the evolutionary architecture, in which small incremental changes are continuously delivered over time.

7.2. Evolution Planning

The approaches that tackle the refactoring usually determine the changes that need to be made, however, they generally do not address how the system could evolve smoothly with minimal interruptions. Therefore, another group of works focuses on evolution planning. Mokni et al. [13] proposed a formal approach to generate reliable evolution plans based on the given change requests. With the formal specification of architectural design, architectural inconsistencies can be checked and avoided. However, this approach cannot guarantee that the functional requirements can be satisfied while the system evolves. Similarly, Tanhaei et al. [14] also applied a formal technique to keep software product lines (SPLs) consistent with the feature model. Applying changes on SPLs is usually conducted through accumulated small steps to achieve large-scale changes. Their approach could be used to ensure that the evolution is according to various refactoring patterns, however, it relies on a customised checking algorithm specific to the pattern to guarantee correctness. Hoff et al.[15] proposed an evolution planning approach for SPLs. With their approach, undesired impacts on structural and logical consistency can be prevented through formal verification. These approaches focus on the evolution of SPLs to ensure that the configurable features are consistent, so the details of the changes at the implementation level must be provided. Brito et al.[16] proposed an algorithm to create the graphs that describe the series

of refactoring operations. This approach can help to understand the code changes needed to study software evolution, however, it only focuses on the changes at the code level and is specific to a programming language. Most of the approaches discussed earlier focus on applying changes at the code level, whereas our approach tackles the design level to ensure that architectural changes can be performed safely.

There are approaches that apply the automated planning technique to software evolution planning at the architecture level. The works such as [17] and [18] presented how the architecture evolution problem can be translated into the planning problem in order to automatically generate the evolution paths. Djibo et al.[19] applied the principles of sequential pattern extraction to plan the future evolution path based on previous evolutions of the evolving architecture. These works demonstrate how automated planning can be applied to the problem domain, however, ensuring that the desired properties can be achieved in the target design has not yet been addressed. Therefore, manual analysis is required to verify whether the plan is correct and ensure that the desired properties can be achieved. To the best of our knowledge, none of these approaches supports the evolutionary architecture.

Unlike other evolution planning approaches, our solution applies formal verification as the fitness function to ensure that the desired functional properties can be achieved to support the evolutionary architecture. Moreover, the evolution plan can be automatically generated as a sequential process consisting of small incremental changes. The verification results of interim models allow us to determine the safe evolution path, which has not yet been addressed in other works.

8. Conclusion

This paper presents an approach to automatically generating the evolution plan for an evolutionary software architecture. The PAT model checker has been applied as a fitness function to verify whether the new functional properties can be achieved after refactoring the design. This way, we are able to ensure that the original functional properties as well as the new ones are preserved after the architecture is evolved. The AI planning technique has been applied using PDDL4J to automatically generate the evolution plan, which serves as a sequential process consisting of refactoring steps. With the generated plan, an evolution path consisting of interim models can be generated. The verification of interim models aids in determining and eliminating

refactoring steps that may disrupt the system, allowing software engineers to better plan ahead of time to mitigate risk during the evolution. We have developed a tool to support the modelling of software architecture design, define functional changes, generate the evolution plan, and verify the properties in order to determine a safe evolution path. Furthermore, we have evaluated the rigorousness and effectiveness of the evolution plan generated by our approach with six software systems. Our experimental results demonstrated that the proposed solution is effective for the purpose of evolutionary architecture. Moreover, we are able to determine the most suitable planning strategy, which generates the evolution plan with minimal system interruptions.

For future work, we plan to automate code refactoring based on the refactoring steps in the evolution plan. To achieve this, the relationship between architectural elements and implementation constructs must be defined. With the code refactoring automated, the system can evolve with minimal human intervention.

References

[1] T. Hynninen, J. Kasurinen, A. Knutas, O. Taipale, Software testing: Survey of the industry practices, in: 2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2018, pp. 1449–1454.

[2] N. Ford, R. Parsons, P. Kua, Building Evolutionary Architectures: Support Constant Change, 1st Edition, O'Reilly Media, Inc., 2017.

[3] L. Chen, Continuous delivery: Overcoming adoption challenges, Journal of Systems and Software 128 (2017) 72–86.

[4] A. Ramírez, J. R. Romero, S. Ventura, Interactive multi-objective evolutionary optimization of software architectures, Information Sciences 463-464 (2018) 92–109.

[5] L. Rizzi, F. A. Fontana, R. Roveda, Support for architectural smell refactoring, in: Proceedings of the 2nd International Workshop on Refactoring, IWoR 2018, Association for Computing Machinery, New York, NY, USA, 2018, p. 7–10.

[6] Y. Lin, X. Peng, Y. Cai, D. Dig, D. Zheng, W. Zhao, Interactive and guided architectural refactoring with search-based recommendation, in: Proceedings of the 2016 24th ACM SIGSOFT FSE, FSE 2016, Association for Computing Machinery, New York, NY, USA, 2016, p. 535–546.

[7] A. ben Fadhel, M. Kessentini, P. Langer, M. Wimmer, Search-based detection of high-level model changes, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 212–221.

[8] R. Li, R. Etemaadi, M. T. M. Emmerich, M. R. V. Chaudron, An evolutionary multiobjective optimization approach to component-based software architecture design, in: 2011 IEEE Congress of Evolutionary Computation (CEC), 2011, pp. 432–439.

[9] A. Koziolek, H. Koziolek, R. Mirandola, R. Reussner, A hybrid approach for multi-attribute qos optimisation in component-based software systems, 2010, pp. 84–101.

[10] V. Cortellessa, D. Di Pompeo, Analyzing the sensitivity of multi-objective software architecture refactoring to configuration characteristics, Information and Software Technology 135 (2021) 106568.

[11] M. Alshayeb, H. Mumtaz, S. Mahmood, M. Niazi, Improving the security of uml sequence diagram using genetic algorithm, IEEE Access 8 (2020) 62738–62761.

[12] T. Holmes, U. Zdun, Refactoring architecture models for compliance with custom requirements, in: Proceedings of the 21th ACM/IEEE MODELS '18, ACM, New York, NY, USA, 2018, p. 267–277.

[13] A. Mokni, C. Urtado, S. Vauttier, M. Huchard, H. Y. Zhang, A formal approach for managing component-based architecture evolution, Science of Computer Programming 127 (2016) 24 – 49, special issue of the 11th FACS.

[14] M. Tanhaei, J. Habibi, S.-H. Mirian-Hosseinabadi, A feature model based framework for refactoring software product line architecture, Journal of Computer Science and Technology 31 (09 2016).

[15] A. Hoff, M. Nieke, C. Seidl, E. H. Sæther, I. S. Motzfeldt, C. C. Din, I. C. Yu, I. Schaefer, Consistency-preserving evolution planning on feature

models, in: Proceedings of the 24th ACM Conference on SPLC: Volume A - Volume A, SPLC '20, ACM, 2020.

[16] A. Brito, A. Hora, M. T. Valente, Refactoring graphs: Assessing refactoring over time, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 367–377.

[17] J. M. Barnes, A. Pandey, D. Garlan, Automated planning for software architecture evolution, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2013, pp. 213–223.

[18] S. Ciraci, H. Sözer, M. Aksit, Guiding architects in selecting architectural evolution alternatives, in: Proceedings of the 5th ECSA, ECSA'11, Springer-Verlag, Berlin, Heidelberg, 2011, p. 252–260.

[19] K. Djibo, M. C. Oussalah, J. Konate, Modelling and planning evolution styles in software architecture, Modelling 1 (1) (2020) 53–76.

[20] N. Chondamrongkul, J. Sun, I. Warren, Software architectural migration: An automated planning approach, ACM Trans. Softw. Eng. Methodol. 30 (4) (jul 2021).

[21] A. Baabad, H. B. Zulzalil, S. Hassan, S. B. Baharom, Software architecture degradation in open source software: A systematic literature review, IEEE Access 8 (2020) 173681–173709.

[22] N. Febbraro, V. Rajlich, The role of incremental change in agile software processes, in: Agile 2007 (AGILE 2007), 2007, pp. 92–103.

[23] L. Shi, K. Rasheed, A Survey of Fitness Approximation Methods Applied in Evolutionary Algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 3–28.

[24] M. Stal, Chapter 3 - refactoring software architectures, in: M. Ali Babar, A. W. Brown, I. Mistrik (Eds.), Agile Software Architecture, Morgan Kaufmann, Boston, 2014, pp. 63–82.

[25] A. M. Eilertsen, Refactoring operations grounded in manual code changes, in: Proceedings of the ACM/IEEE 42nd ICSE: Companion Proceedings, ICSE '20, ACM, New York, NY, USA, 2020, p. 182–185.

[26] P. Di Francesco, P. Lago, I. Malavolta, Migrating towards microservice architectures: An industrial survey, in: 2018 IEEE International Conference on Software Architecture (ICSA), 2018, pp. 29–2909.

[27] N. Mangano, T. D. LaToza, M. Petre, A. van der Hoek, How software designers interact with sketches at the whiteboard, IEEE Transactions on Software Engineering 41 (2) (2015) 135–156.

[28] M. Erder, P. Pureur, Chapter 4 - evolving the architecture, in: M. Erder, P. Pureur (Eds.), Continuous Architecture, Morgan Kaufmann, Boston, 2016, pp. 63–101.

[29] M. Shahin, M. Zahedi, M. A. Babar, L. Zhu, An empirical study of architecting for continuous delivery and deployment, Empirical Software Engineering 24 (3) (2019) 1061–1108.

[30] O. Zimmermann, Architectural refactoring for the cloud: a decision-centric view on cloud migration, Computing (10 2017).

[31] G. Antoniou, , G. Antoniou, G. Antoniou, F. V. Harmelen, F. V. Harmelen, Web ontology language: Owl, in: Handbook on Ontologies in Information Systems, Springer, 2003, pp. 67–92.

[32] N. Chondamrongkul, J. Sun, I. Warren, Pat approach to architecture behavioural verification, in: 31th International Conference SEKE, 2019, pp. 187–192.

[33] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, P. Merson, Documenting Software Architectures: Views and Beyond, 2nd Edition, Addison-Wesley Professional, 2010.

[34] N. Chondamrongkul, J. Sun, I. Warren, Ontology-based software architectural pattern recognition and reasoning, in: 30th International Conference SEKE, 2018, pp. 25–34.

[35] I. Horrocks, Owl: A description logic based ontology language, in: Principles and Practice of Constraint Programming - CP 2005, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 5–8.

[36] J. Sun, Y. Liu, J. S. Dong, C. Chen, Integrating specification and programs for system modeling and verification, in: Proceedings of TASE'09), IEEE Computer Society, 2009, pp. 127–135.

[37] R. Sindhgatta, N. C. Narendra, B. Sengupta, Software evolution in agile development: A case study, in: Proceedings of the ACM International Conference Companion OOPLA, OOPSLA '10, ACM, New York, NY, USA, 2010, p. 105–114.

[38] V. Rajlich, Software evolution and maintenance, in: Future of Software Engineering Proceedings, Association for Computing Machinery, New York, NY, USA, 2014, p. 133–144.

[39] C. Gérard, D. Pellier, pddl4j: Pddl4j v3.5.0 (Sep. 2016). `doi:10.5281/zenodo.61692`.

[40] A. Gerevini, D. Long, Plan constraints and preferences in pddl3 - the language of the fifth international planning competition, Tech. rep. (2005).

[41] GitHub - cnacha/lifeguard — github.com, `https://github.com/cnacha/lifeguard`, [Accessed 26-Aug-2022].

[42] C. R. o. Eventuate, Introduction to microservices (Jan 2022).
URL `https://www.nginx.com/blog/introduction-to-microservices/`

[43] X. Xu, I. Weber, M. Staples, Case Study: AgriDigital, Springer International Publishing, Cham, 2019, pp. 239–255.

[44] W. inc., Microservice sockshop demo (Jan 2022).
URL `https://github.com/microservices-demo/microservices-demo`

[45] HouariZegai, Houarizegai/hotelreservationsystem: A web application to book a room in a hotel (room reservation).
URL `https://github.com/HouariZegai/HotelReservationSystem`

[46] Acrenwelge, Acrenwelge/ers-java: Java / spring implementation of the expense reimbursement system.
URL `https://github.com/acrenwelge/ERS-Java`