Synergizing domain expertise with self-awareness in software systems

Chen, Tao; Bahsoon, Rami; Yao, Xin

DOI:
10.1109/JPROC.2020.2985293

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (Harvard):
Chen, T, Bahsoon, R & Yao, X 2020, 'Synergizing domain expertise with self-awareness in software systems: a patternized architecture guideline', *Proceedings of the IEEE*, vol. 108, no. 7, 9089006, pp. 1094-1126. https://doi.org/10.1109/JPROC.2020.2985293

INVITED PAPER

Synergizing Domain Expertise With Self-Awareness in Software Systems: A Patternized Architecture Guideline

By Tao Chen, *Member IEEE*, Rami Bahsoon, and Xin Yao, *Fellow IEEE*

ABSTRACT | To promote engineering self-aware and self-adaptive software systems in a reusable manner, architectural patterns and the related methodology provide an unified solution to handle the recurring problems in the engineering process. However, in existing patterns and methods, domain knowledge and engineers' expertise that is built over time are not explicitly linked to the self-aware processes. This link is important, as knowledge is a valuable asset for the related problems and its absence would cause unnecessary overhead, possibly misleading results, and unwise waste of the tremendous benefits that could have been brought by the domain expertise. This article highlights the importance of synergizing domain expertise and the self-awareness to enable better self-adaptation in software systems, relying on well-defined expertise representation, algorithms, and techniques. In particular, we present a holistic framework of notions, enriched patterns and methodology, dubbed DBASES, that offers a principled guideline for the engineers to perform difficulty and benefit analysis on possible synergies, in an attempt to keep "engineers-in-the-loop." Through three tutorial case studies, we demonstrate how DBASES can be applied in different domains, within which a carefully selected set of candidates with different synergies can be used for quantitative investigation, providing more informed decisions of the design choices.

KEYWORDS | Architectural patterns; human-in-the-loop; self-adaptive software systems; self-aware software systems.

I. INTRODUCTION

Engineering software systems has become increasingly complex and labor-intensive due to the continuous changes in requirements, the underlying environments, and the relevant data. Such complexity is prevalent when engineering self-aware and self-adaptive software systems—a category of systems that is capable of obtaining and maintaining knowledge on themselves and the environment, reasoning about this knowledge, and eventually adapting their operations to better cope with the changes. In this respect, engineers need some sets of high-level guideline that provides a clear overview of the software system to be built, based on which they are able to make better-informed decisions during the engineering process. Such a high-level guideline for engineering software systems can be represented in the form of architectural patterns and their methodologies. In essence,

Manuscript received May 30, 2019; revised November 10, 2019 and March 6, 2020; accepted March 23, 2020. This work was supported in part by the Guangdong Provincial Key Laboratory of Brain-Inspired Intelligent Computation, in part by the Program for Guangdong Introducing Innovative and Enterpreneurial Teams under Grant 2017ZT07 × 386, in part by the Shenzhen Science and Technology Program under Grant KQTD2016112514355531, and in part by the Program for University Key Laboratory of Guangdong Province under Grant 2017KSYS008. *(Corresponding authors: Tao Chen; Xin Yao.)*

Tao Chen is with the Department of Computer Science, Loughborough University, Loughborough LE11 3TU, U.K. (e-mail: t.t.chen@lboro.ac.uk)

Rami Bahsoon is with the School of Computer Science, University of Birmingham, Birmingham B15 2TT, U.K. (e-mail: r.bahsoon@cs.bham.ac.uk).

Xin Yao is with the Guangdong Provincial Key Laboratory of Brain-Inspired Intelligent Computation, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (e-mail: xiny@sustech.edu.cn).

Digital Object Identifier 10.1109/JPROC.2020.2985293

architectural patterns are particular solutions for common and recurring domain-specific problems, culminating best practices, and described at high level [1]. A variety of architectural patterns and methodologies exist, each of which aims at a different context, for example, distributed systems [2], [3], service-oriented systems [4], [5], self-adaptive systems [6], [7], and more recently, self-aware and self-adaptive software systems [8]–[12] (also known as self-awareness architectural patterns).

Unlike the other patterns, self-awareness architectural patterns particularly document the common primitives and different capabilities of self-awareness for obtaining and maintaining knowledge about different aspects, such as time, goals, or interactions between different nodes of software systems. Although these patterns are abstract, they can be instantiated to meet particular needs for engineering a self-aware and self-adaptive software system, thereby providing more concrete guideline on how to align the capabilities of self-awareness with the requirements. Over the last few years, those patterns and the related methodology have proved to be promising when engineering self-aware and self-adaptive software systems, as evident by the fact that they have been referenced and used in various autonomic domains, such as cloud resource and configuration management [13]–[15], multiprocessors systems scheduling [16], sensor network control [17], and multicamera coordination [18].

Traditionally, engineering self-awareness in software systems have been primarily supported by various artificial intelligence (AI) algorithms, which serve as cheap and "black boxes" that can be directly applied with little specialization [19], [20]. However, as reviewed by Menzies [21], an emerging question in the application of AI algorithms to various engineering problems is the validity of the assumptions that underlie the creation of those algorithms. Therefore, the practice of applying standard AI algorithms as "black boxes," where researchers do not tinker with internal design choices of these algorithms with respect to their expertise on the problem is not ideal [21]. Indeed, self-aware and self-adaptive software systems have never been created by nonexperts. This means that software and systems engineers often accumulate domain expertise that is built over time. Such expertise, if captured and exploited, would provide an important added value to consolidate the self-awareness capability of the software system. Utilizing domain expertise to guide the processes of underlying AI algorithms, and thus the self-awareness, can bear additional benefit. In this way, the software system would be more controllable, which helps to monitor and avoid some abnormalities in behaviors, providing a foundation for keeping "engineers-in-the-loop." There exist many research studies [22]–[26] that show superior results can be obtained by specializing AI algorithm to the particulars of engineering problem with domain expertise.

With this in mind, despite the successful applications of the existing architectural patterns and methodology for engineering self-awareness, the consideration of engineers' expertise, particularly on how they can be "synergized" into the self-awareness capabilities, is weak, *ad hoc* and left implicit. By the term synergy, we refer to the process of incorporating domain expertise, which involves the knowledge of the problem that is not naturally initiative (on contrary to, e.g., the range and type of parameters, various equality, and inequality constraints) but can be extracted following engineering principles, into the underlying algorithms/techniques that realize self-awareness. Indeed, the lack of a holistic framework of patterns and methodology would inevitably create a barrier for the domain information/knowledge to be maintained, reused, and exploited to steer the design process, especially given a large variety of existing expertise representations and AI algorithms. This absence can eventually result in some strong domain expertise being overlooked, causing unnecessary overhead and possibly misleading results [27], [28].

To overcome such a gap, in this article, we formalize a holistic framework that provides a principled guideline to perform Difficulty and Benefit Analysis for Synergizing domain Expertise and Self-awareness (hence dubbed as DBASES). Our aim is to elaborate and showcase how DBASES can support the "synergy" and reveal its importance, taking into account the self-awareness in software systems based on well-defined and widely used expertise representations, algorithms, and techniques. It is indeed an ambitious plan, therefore, we intend to be introductory rather than comprehensive. However, we hope that this article can spark a dialogue about the diverse and representative research on combining domain expertise with self-awareness, and that some level of consensus on the design of such synergy will be achieved.

Specifically, our key contributions of the DBASES framework in this article are as follows.

1) We introduce general notions that captures the domain expertise of the engineers and their synergies with the concept of self-awareness, providing intuitive, extracted, and readily available information to enrich the self-awareness architectural patterns. Specifically, we contribute to the following.
   a) We present the notions of expertise representation with concrete examples, based on which we form a classification and the related rules that help capture the expertise knowledge.
   b) Drawing on the expertise representation, we codify a taxonomy that describes their nature in terms of structurability and tangibility.
   c) We then discuss their possible synergies with different capabilities of self-awareness, and present rules that classify different levels of synergies and the relative difficulty,[1] with respect to the structurability and tangibility of expertise representation.
2) We illustrate, by means of examples, how the proposed notions can be used to enrich the well-defined

[1]Difficulty is related to complexity, which can impact both the implementation and maintenance of a software system.

self-awareness architectural patterns from the literature [8], and in what ways they can be instantiated to cope with different styles of synergies.

3) Supporting by the proposed notions and the enriched patterns, we present a practical, intuitive, and step-by-step methodology that assists the engineers to analyze the difficulty and benefit for alternative synergies of domain expertise with self-awareness, revealing their importance. This would help the engineers to elicit the most preferred candidate(s) for further investigation, while ruling out some of the options that are clearly undesired, thus saving great effort in the development.
4) We demonstrate three recent tutorial case studies [23], [24], [26], which relied on DBASES, that seek to build self-aware and self-adaptive software systems. Through quantitative results, we show how DBASES can be applied to the engineering process for analyzing the difficulty and benefit of different synergy candidates, leading to a set of more promising ones for further investigation.

The remaining article is organized as follows. We motivate the needs and discuss the related work in Section II, followed by a brief overview of the capabilities of self-awareness and the existing self-awareness architectural patterns in Sections III and IV. After that, in Section V, we present the notions and theoretical foundation that underpins DBASES. In Section VI, we illustrate how the existing self-awareness patterns can be enriched with DBASES. In Section VII, we present, as part of DBASES, a practical step-by-step methodology that assists the engineers in selecting the possible ways of synergies. Three tutorial case studies from different domains are drawn in Section VIII to demonstrate how DBASES can be practically applied. Finally, Section X concludes this article with discussion on future work.

II. PRELIMINARIES

A. Problem Nature and Domain Expertise

As mentioned, self-aware and self-adaptive software systems have been increasingly relying on AI algorithms and techniques. Indeed, given the significant growth of the AI community, it is not uncommon to see that successful engineering of self-awareness is underpinned by several AI algorithms [23], [26], [29]–[36], which conducts learning, reasoning, and problem solving.

In general, the application of standard AI algorithms and techniques may need to be combined with sufficient domain information, and thus they can better serve the purpose. Yet, it is important to note that domain information can be distinguished between the problem nature and the domain expertise. In fact, the former refers to the nature information of the problem, which is the basic elements required to appropriately apply the algorithm/techniques (e.g., the range of parameters) [27], [28]; and the latter is the engineers' domain expertise, which is specifically related to the engineering problem to be addressed and is often deemed optional, but desirable. In essence, what make the additional engineers' domain expertise differ from the basic problem nature is as follows.

1) Problem nature refers to commonly known properties and characteristics of the problem domain, such that the AI algorithms have to comply with in order to be used appropriately. This may, for example, include the range of parameters, sparsity of the values, various equality, and inequality constraints. Directly applying standard AI algorithms is often considered as exploiting only the problem's nature due primarily to the generality of existing AI algorithms [21].
2) In contrast, the domain expertise is represented as or produced by typical software and system engineering methods, practices, and models. Most commonly, the knowledge of domain expertise is not naturally intuitive form the problem context but can be extracted through engineering practices, skills, and tools, for example, design models, formatted documents, or even concepts.

B. Lessons From Applying Standard AI Algorithms in Engineering

In the software and system engineering community, there is an increasing recognition on the limitation of applying standard AI algorithms to various engineering problems. A very recent study, conducted by Agrawal and Menzies [32], on a wide range of software engineering problems has revealed the following fact.

> *Our conclusion is that the algorithms which we call "general AI tools" may not be "general" at all. Rather, they are tools which are powerful in their home domain but need to be used with care if applied to some new domains like software engineering. Hence, we argue that it is not good enough to just take AI tools developed elsewhere, then apply them verbatim to software engineering problems. Software engineers need to develop AI tools that are better suited to the particulars of software engineering problems.*

The conclusion delivers a very clear message that the standard AI algorithms combined with the necessary information of problem nature, including those that realize self-awareness [27], [28], cannot fully meet the complex requirements of engineering software systems. It, therefore, calls for better specialization of these AI algorithms based on the domain expertise of engineers.

From the literature, it is not uncommon to see that greater benefits can be obtained by synergizing domain expertise. For example, there is a thread of research that seeks to synergize the feature model, which represents domain expertise on requirement analysis, with evolutionary search to reason about behaviors of the self-aware and self-adaptive software systems [37]. This is motivated by the fact that domain expertise on the requirement

cannot be easily captured by simply applying the AI algorithms. Another example of "software/system engineering needs different AI algorithms" comes from the work of Hindle *et al.* [38], in which they stated that, unlike the common areas where AI was most originally applied, domain expertise in software engineering may suggest some important terms in the code which is used exponentially less frequent. This can provide useful information when model the software system with AI, enabling more accurate self-awareness of the faults.

The domain expertise of engineers can often serve as useful information to engineer self-aware and self-adaptive software systems, thus they should not be simply ignored. To this end, a better synergy between domain expertise and AI algorithms is required. Although in this case the AI algorithms may be made less general and pose extra difficulty, they are expected to work better under the given problem where the domain expertise lies, and more importantly, rendering the self-awareness more controllable. It is in fact part of our contributions in DBASES to provide better analysis on the tradeoff between difficulty and benefit when designing different ways of synergy.

C. Problems

As summarized in the recent surveys [27], [28], majority of the work incorporates merely the necessary information of the problem nature with self-awareness in an *ad hoc* manner, which is the direct application of the standard AI algorithms. This is because the problem nature can be easily obtained and the existing AI algorithms are designed to be as general as possible, such that they will cope with the basic properties of different domains. However, it is clearly difficult to perform the same for synergizing domain expertise without omission. The key issue is that there is a lack of general guideline that assists the engineers when engineering self-awareness into software systems with explicit consideration of the domain expertise. For example, it is not uncommon that engineers would have certain domain expertise represented as models, documentations, or even artifacts of software systems, but how they may be related to self-awareness is unknown. Below, we illustrate some common but difficult decisions and problems to deal with during the synergy process, together with what contributions in DBASES can help on each.

1) Which available domain expertise can be synergized into which aspect of self-awareness?
   a) Answering such a question would require understanding on both the available domain expertise and which aspect of the self-awareness is required, for example, time, goal, or interaction [39]. Clearly, there will be constraints that prevent certain synergies, for example, a feature model cannot usually help in terms of interaction, as its notation does not embed any knowledge of it. In essence, the feasible synergies form the possible candidates for the engineers to make design decision. Yet, it is challenging to build the set of candidates for synergy in the absence of systematic guideline, especially when multiple forms of domain expertise and aspects of self-awareness exist.

> **What parts in DBASES can address this?**
> Self-awareness architecture patterns with explicit synergy between domain expertise and self-awareness, as discussed in Sections IV and VI.

2) To what extent can a synergy be completed and what are the difficulties?
   a) Synergies can often be done in different levels, for example, whether the domain expertise can be directly incorporated into the algorithm/techniques or certain internal components need to be specialized [8], [11]. This is a crucial design decision to make and it should not be conducted without knowing the relative difficulty, which directly related to the cost of the engineering and maintenance process. However, without guideline, it would be difficult for the engineers to obtain a full picture of the possible extent of synergy and their difficulties.

> **What parts in DBASES can help this?**
> Generic notions and categories of different synergy levels and their relative difficulties, as shown in Section V.

3) How to make decision taking into consideration the difficulty of synergy and the expected benefit?
   a) The different candidates of alternative design options would inevitably lead to a decision space [40], [41]. As a result, it would be challenging to enable well-informed decision making without the support of quantifiable and intuitive metrics. In particular, given the potentially large number of alternatives, it would be nicer to intuitively understand which can be ruled out and what needs to be investigated further.

> **What parts in DBASES can improve this?**
> Methodological guideline and quantification metrics for visualizing possible candidates of synergies with respect to their difficulties and benefits, as elaborated in Section VII.

As a result, the lack of general guideline on how to exploit engineers' expertise when engineering self-awareness into software systems would hinder the benefits of domain expertise synergy, causing barrier to create more advanced and controllable self-awareness driven by the expertise of engineers.

This is what we seek to achieve in this article with DBASES for engineering self-aware and self-adaptive software systems, in which we conduct the first attempt

to propose a general, yet holistic framework to assist the engineers in making decisions of synergy, or at least a more concise set of options that are subject to further investigation.

D. Related Work on Architectural Pattern and Methods

Software and system architecture, as the highest level of abstraction for all software systems, serves as the framework for satisfying requirements; as the managerial basis for cost estimation and process management; and as an effective basis for reuse and dependence analysis [42]. From the community of software and systems engineering, architectural patterns and the related methods seek to abstract common features of architecture instances in a specific domain, which is known to serve as a useful guide to the engineers when designing software systems [42]. Among others, cost–benefit analysis method (CBAM) [41] and architecture tradeoff analysis method (ATAM) [40] are two most widely used methodologies that help to reason about different design options on architectures and their patterns. However, they were designed to deal with general software system and thus are irrelevant to the concept of self-awareness.

Over the past two decades of research for architecting self-aware and self-adaptive software systems, several architecture patterns and their methodologies have emerged. Among others, feedback loop-based architecture pattern [43], whether as single loop or multiple loops, have been the most widely adopted approach. Such pattern merely assumes that the software system can be monitored, and that it can be influenced after certain process is completed based upon the collected data. The reason behind its popularity is due to its simplicity and flexibility, such that there is no constraint on how and what should be architected in the patterns. Yet, as the software system becomes more complex, such simplicities turna barrier, as the software and systems engineers require more specific guideline when designing the architecture which has been missing from the feedback loop-based architecture pattern.

In light of these, the MAPE-K architecture pattern [6] and its design guidelines are proposed to provide more specific codification about what should be achieved within a feedback loop when engineering self-aware and self-adaptive software systems. In MAPE-K, the (K)nowledge component is shared by the (M)onitor, (A)nalyser, (P)lanner, and (E)xecutor components, which provides primitives for expressing domain knowledge in K. This knowledge is used to reason about run-time adaptation. Two other patternized methods, which are subclasses of MAPE-K but generic enough to be classified as representative styles with distinct qualities. The first one is proposed by Oreizy *et al.* [44] in which the pattern consists of an adaptation layer and an evolution layer. Particularly, the adaptation layer is responsible for monitoring and adapting, whereas the evolution layer caters to ensuring that changes in the running system are performed in such a way that the operation of the system is not disrupted. The second patternized method is Rainbow [7], which is explicitly designed for engineering rule-based adaptation in software systems. Since the above patterns assume a centralized scenario where there is only one instance of software system to be adapted, the MAPE-K is then further extended by Weyns *et al.* [45] into a decentralized version, such that they are specialized into contexts with different degrees of decentralization that the software system encounters, with some guidelines.

Inspired by Gat's three-layer architecture in the robotics domain [46], Kramer and Magee [47] presented a conceptual three layered architecture patterns and methods for self-adaptive software system. The three layers, namely goal layer, change layer, and component control layer, work in a hierarchical way such that the goal layer provides change plans, which are then further translated into change actions by the change layer, and eventually those actions are run by the component control layer. The opposite of the direction would occur when data need to be collected.

Alternative to the MAPE-K and the three-layer pattern, the SElf-awarE computing framework (SEEC) [48] is another set of architectural patterns and methods that claim self-aware capabilities. In a nutshell, SEEC relies on the basic (O)bserve-(D)ecide-(A)ct (ODA) loop [48]. Here, the O and A components in ODA are equivalent to the M and E components in MAPE-K, respectively, while analysis and planning tasks are subsumed in the Decide component. Another more recent effort, namely the model-based learning, reasoning, and acting loop (LRA-M loop) [9], aims to capture the knowledge of self-awareness in terms of universal models, which can then be exploited by the reasoning.

However, all the above patterns and methods have focused on providing guideline about how to exploit the obtained knowledge to inform adaptation, but limited in modeling the knowledge at a coarse grain, without explicit distinction between knowledge concerns for different levels, for example, at goals, time, or interaction. In 2014, we proposed a set of self-awareness patterns and methodolgoy [8], [10]–[12] derived from the general concept of self-awareness [49] for engineering self-aware and self-adaptive software systems. Unlike others, we explicitly encode the pattern based on the fine-grained capability of self-awareness with respect to stimulus, goal, time, interaction, and meta-self, considering their distinctions and interplays (we elaborate the patterns in Section III). Those patterns have been followed by a considerable amount of work from other research groups and have attracted a wide range of attention. However, our experience with industrial partners when using those patterns and methods (together with the other state of the art) is that they fail to capture how domain expertise, and more importantly how they can be combined with the AI algorithms that underpin self-awareness, which has now become a major barrier for them to follow.

III. CAPABILITIES OF SELF-AWARENESS IN SOFTWARE SYSTEMS

Self-awareness is certainly not new in the other disciplines, but it is challenging to model such a concept in the context of software systems. In this article, we use the term node to refer to a software system that can either work alone, or as one individual in a networked group of different systems. Drawing on Neisser's notions on the self-awareness from the psychology domain, different capabilities of computational self-awareness have been codified [49], which are what the DBASES framework based upon. As illustrated below, each self-aware capability captures distinct knowledge that a software system would need in order to perform self-adaptation and self-expression at certain degree.

1) *Stimulus Awareness:* A software system is stimulus-aware, if it has knowledge of stimuli. The software system is not able to distinguish between the sources of stimuli. It does not have knowledge of past/future stimuli but enables the ability in a software system to respond to events. It is a prerequisite for all other capabilities of self-awareness.
2) *Time Awareness:* A software system is time-aware if it has knowledge of historical and/or likely future phenomena. Implementing time-awareness may involve the software system possessing an explicit memory, capabilities of time-series modeling and/or anticipation.
3) *Interaction Awareness:* A software system is interaction-aware if it has knowledge that stimuli and its own actions form part of interactions with other systems and the environment. It has knowledge via feedback loops that its actions can provoke, generate, or cause specific reactions from the environment. It enables a software system to distinguish between other nodes of software systems and environments. Simple interaction-awareness may just enable a software system to reason about individual interactions. More advanced interaction-awareness may involve the possessing knowledge of social structures such as communities or network topology. In this article, from the pattern's perspective, we strictly treat interaction awareness with respect to the different nodes of software systems and/or the environment, and thus the internal information about interactions between different elements within a single software system is not considered as knowledge of interaction.
4) *Goal Awareness:* A software system is goal-aware if it has knowledge of current goals, objectives, preferences, and constraints. It is important to note that there is a difference between a goal existing implicitly in the design of a software system, and it having knowledge of that goal in such a way that it can reason about it. The former does not describe goal-awareness; the latter does. Example implementations of such knowledge in a software system include state-based goals (i.e., knowing what is a goal state and what is not) and utility-based goals (i.e., having a utility or objective function).
5) *Meta-Self-Awareness:* A software system is meta-self-aware if it has knowledge of its own capability(ies) of awareness and the degree of complexity with which the capabilities(ies) are exercised. Such an awareness permits a software system to reason about the benefits and costs of maintaining a certain capability of awareness (and degree of complexity with which it exercises this capability).

IV. SELF-AWARENESS ARCHITECTURAL PATTERNS

Although the notions of self-awareness can be well conceptualized with respect to a software system, the presence of various requirements would still need more concrete guideline on how those concepts can be modeled within the needs. This urges a formal documentation of the self-awareness as architectural patterns when engineering self-aware and self-adaptive software systems. An architectural pattern refers to an architectural problem-solution pair for a given domain, which, in the context of self-aware software systems, means that they are linked to the capabilities of self-awareness. Our previously proposed self-awareness architectural patterns [8], [10]–[12] have been showing great potential in engineering self-aware and self-adaptive software systems.

In such context, different capabilities of computational self-awareness enable capability of the systems to obtain and react upon certain knowledge, which could be either about its own states or about the environment. The patterns provide a formal way to ensure that only relevant capabilities of self-awareness are included and their inclusion justified by identified benefits. There is no need for a system to become unnecessarily complex, learning, and maintaining self-awareness capabilities which do nothing to advance the outcomes for that system, generating only overhead. Each of the self-awareness architectural patterns is decentralized by design. That is, structurally they resemble a peer-to-peer network of interconnecting self-aware nodes, varying only in the number of the capabilities and the type of interconnection between them. Even with the decentralized expression, a centralized software system can be easily modeled by considering only one node. In this section, we provide an overview of these well-defined patterns with selected examples.

A. Notations

In general, an architecture of software system consists of two fundamental elements, the component and connector, which are described as follows [8].

1) *Component:* A smaller and more manageable part of a software system, which is often divided based on requirements, functionality, and purpose.

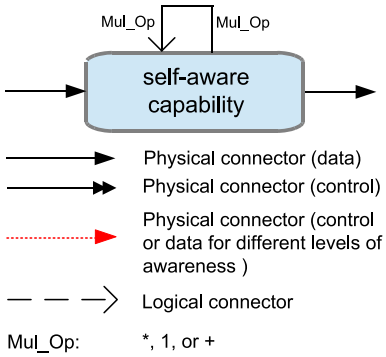


Fig. 1. ***Basic notation for self-awareness architectural patterns.***

2) *Connector:* A bridge that represents the possible interaction between components and the multiplicity involved.

The uniqueness of the self-awareness architectural patterns is that, the component is replaced by the notion of capability of self-awareness, sensoring, and actuating, in which case they are not necessarily to be a one-to-one mapping. In other words, depending on the context, two or more capabilities may be combined and realized in one component; or one capability can be implemented in separate components. The basic notation used to describe the patterns is depicted in Fig. 1.

In particular, the connectors are used to express the physical and logical interactions, which have different notations.

1) *Physical Connector:* This means there is a direct interaction between the components, and each component is required to directly interact with the others. Notably, the physical connectors are further divided into two types. The first type, expressed as a red arrow, particularly refers to the interactions for different capabilities of the self-awareness (e.g., goal and time-awareness); in contrast, the second type, denoted by black solid arrows, represents the interactions for the self-awareness of the same capability (e.g., the interaction-awareness from different interacting software systems).
2) *Logical Connector:* This does not require direct interaction, but rather the data or control in the interaction is sent/received through the sensors and actuators, which have the physical connector. For instance, self-expression and self-adaptation might be logically required to reach consensus among different nodes, but such an interaction is physically realized through sensors and actuators.

Note that the sensors and actuators can be either external or internal, where the former refers to the case that information/control is aimed for external nodes; whereas the latter means such data/control exchange happen only internally at the current node. The benefit of additionally introducing the logical connector is that, for example, when designing a capability of self-awareness where the communication protocol is not needed, the pattern can still illustrate that the software system needs to interact with the others. Thus, this provides the engineers with a more precise view on the architecture.

The multiplicity operators are used to represent how many concrete components (which may realize one or more capabilities of self-awareness), including those from different nodes of software systems, are involved in the interaction. In the self-awareness architectural patterns, there are three types of multiplicity operators (denoted as *Mul_Op*).

1) $+$ expresses that the number of components that realize the same capability in the interaction is restricted to at least one.
2) 1 indicates that one and only one component that realizes the same capability is permitted.
3) * indicates that zero, one, or many components that realize the capability specified is permitted in the interaction.

B. Patterns

Drawing on the feasible combinations of the self-aware capabilities, we have previously documented eight well-defined patterns for engineering self-aware software systems [8]. In a nutshell, these patterns are summarized in Table 1. Noteworthily, the meta-self-awareness is considered as an optional capability, and thereby it is not explicitly coded into a particular given pattern. Each pattern was documented using standard pattern template [50] as follows.

1) *Problem/Motivation:* A scenario where the pattern is applicable.
2) *Solution:* A representation of the said pattern in a graphical form.
3) *Consequences:* A narration of the outcome of applying the pattern.
4) *Example:* Instance of the pattern in real applications or systems.

We designed the patterns following the principles of architectural patterns.

> *An architectural pattern is a named collection of architectural design decisions that are applicable to a recurring design problem parameterized to account for different software development contexts in which that problem appears [52].*

In other words, we provide a collection of architecture design decisions to realize the self-aware capability, parameterized by the level of knowledge available (which is behavioral in essence). The provided description is generic, providing template solution to recurring problems. Level of knowledge can range from stimuli, time, goal, interaction, and so the parameterization of the design decisions that invoke the self-aware capability.

In fact, codifying different structures has been commonly used as the way to patternize software

Table 1 Self-Awareness Architectural Patterns

Pattern	Self-Aware Capabilities	Characteristics
Basic Pattern	stimulus-awareness	For cases where some actions need to be triggered in order to cope with emergent events and stimuli
Basic Information Sharing Pattern	stimulus- and interaction-awareness	For cases where more nodes may be required with loosely shared data to meet the scalability requirement of the system
Coordinated Decision-making Pattern	stimulus- and interaction-awareness (with additional interactions to external nodes)	For cases requiring consistent global decision making in a cooperative setting
Temporal Knowledge Sharing Pattern	stimulus-, interaction- and time-awareness	For cases where timing of actions and availability of historical knowledge have an impact on the integrity of information sharing in the software system
Temporal Knowledge Aware Pattern	stimulus- and time-awareness	For cases where timing of actions and availability of historical knowledge is required only at the local level
Goal Sharing Pattern	stimulus-, interaction- and goal-awareness	For cases where goal reasoning and optimization is required with strong consensus
Temporal Goal Aware Pattern	stimulus-, time- and goal-awareness	For cases where timing of actions and availability of historical knowledge are required for local optimization and reasoning of goal
Fully Self-Aware Pattern	stimulus-, interaction-, time- and goal-awareness	For cases where timing and historical knowledge is required for performing goal reasoning with strong consensus

architecture when engineering self-aware and self-adaptive systems [45]. Our ways of formulating and describing the patterns were inspired by Weyns *et al.* [45], who proposed patterns for self-adaptive systems with a special focus on their interactions in a decentralized manner. It is worth noting that the pattern can be instantiated, such that a capability may be decomposed into more than one actual components.

Indeed, the key differences of the patterns are what combination of self-aware capabilities is involved, but they also exhibit different forms of interactions and multiplicity. This is important, as the combination of capabilities cannot be done arbitrarily. For example, all the patterns would need stimulus-awareness; stimulus- and goal-awareness cannot be the only capabilities to form a pattern, as merely obtaining information about the stimulus does little help to reason about goals. There are also examples where the combination of self-aware capabilities are the same but differ on how they interact with each other, for example, the basic information sharing pattern and coordinated decision-making pattern.

In the following, we elaborate on two patterns as examples; the more comprehensive specification can be found in our handbook [8].

1) Temporal Knowledge Aware Pattern:

a) Problem/Motivation: The knowledge of timing enables the capability of proactive adaptation and potentially, better adaptation quality. However, the other capability of awareness, for example, interaction, might not be a necessity, therefore it could affect the self-aware system as it is suffering unnecessary overhead.

b) Solution: As shown in Fig. 2, in this pattern, the knowledge of timing enables the capability of proactive adaptation and potentially, better adaptation quality, which is specifically supported via time-awareness. The temporal knowledge aware pattern incorporates only time-awareness working in conjunction with stimulus awareness, which eliminates the unnecessary overhead introduced by the other capabilities of self-awareness, that is, the goal, interaction, and meta-self-awareness may not be needed.

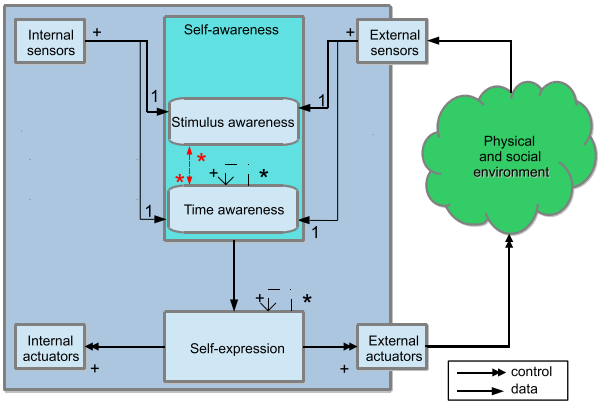


Fig. 2. ***Temporal knowledge aware pattern.***

c) Consequences: When using this pattern, the key benefit is that the software system can be equipped with knowledge about historical data. The categories of data are vast, ranging from the internal states or the environment. However, this should not include data about the other nodes, as interactions have been omitted. It should be noted that this pattern does not cater for changing goals and their related reasoning. That is, it assumes that the goal of the software system is known at design-time and statically encoded in the system, without the opportunity to modify and reason about at run-time.

d) Example: A concrete example of where this pattern is applied could be for the cloud environment where resource is sharing via virtual machine (VM) on each node of a software system. In this context, by leveraging the historical usage of resources, time-series prediction would

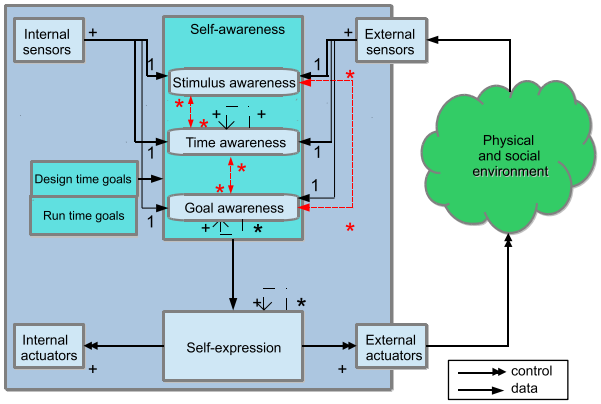


Fig. 3. ***Temporal goal aware pattern.***

be able to predict the demand of VMs on a node of software system for the near future, which assists proactive provisioning of resource and potentially, prevents requirements violation and/or resource exhaustion.

2) Temporal Goal Aware Pattern:

a) Problem/motivation: The knowledge of goals and time together might not necessarily to be shared among nodes, especially in cases where the optimization of local goals could lead to acceptable global optimum.

b) Solution: As shown in Fig. 3, in the temporal goal aware pattern, the goal-awareness provides explicit capability to reason about and even modify the goal at runtime, which offers further guarantee on the optimality of certain goals. However, the knowledge of goals and time might not necessarily to be shared among different software systems, especially in cases where the optimization of local goals could lead to acceptable global optimum. Specifically, in this pattern there is no notion of "sharing" information as the software system is not aware of any interactions and, therefore, it does not have the awareness of the presence of the other nodes. It is worth noting that the absence of interaction awareness does not mean there is no interaction—the software system and the environment could still interact with each other, but it merely is not aware of the details involved in the process.

c) Consequences: A key benefit of this pattern is that the knowledge of historical events can be used in conjunction with the ability to reason about goals. This often provides emergent adaptation behaviors [23], [25]. However, a major limitation is the removal of interaction awareness, especially when the goal-awareness is present, implies that different nodes of software systems could be in inconsistent state. The engineers should carefully verify that such situation would not result in the violation of system requirements. In addition, the self-expression and self-adaptation on a software system could not use any information from others when making decisions.

d) Example: Example application domain of the pattern could be: for adaptive web application in a centralized mode, there is only a single software system exist, and thus no interaction is needed. Another more complex example is when orchestrating fully decentralized harmonic synchronization among different mobile devices, which requires each node of software system to be aware of stimulus, time, and goal but not necessarily interaction. In such a case, each software system receives phase and frequency updates from the others or the environment, and reacts upon based on its own time and goal information. This is a typical example where there are occurrences of interaction but no occurrences of interaction awareness because a single software system only aware of the incoming phase and frequency updates but it has no knowledge of where they come from.

C. Guideline on Selecting Patterns and Underlying Algorithms/Techniques

In our handbook [8], we have codified a comprehensive guideline that assists the software engineers to select the self-awareness architectural patterns for a node, and the underlying algorithms/techniques[2] that realize each capability. In a nutshell, the selection of patterns and algorithm/techniques follows the general processes of ATAM [40], which is a well-known methodology on design selection, such that the choice is made based on qualitative assessment and quantitative evaluation, supported by simulation and profiling.

Noteworthily, albeit that the patterns are alternative for a single node, different nodes can be based upon distinct patterns, or different instantiations of an identical pattern, under systems-of-system or distributed environment. Therefore, the patterns can be used in a composite manner.

As shown in Fig. 4, the overall guideline is an iterative process, in which the selection of pattern and the underlying algorithms/techniques can be continuously refined

[2]This may be a type of algorithms/techniques instead of a specific one.

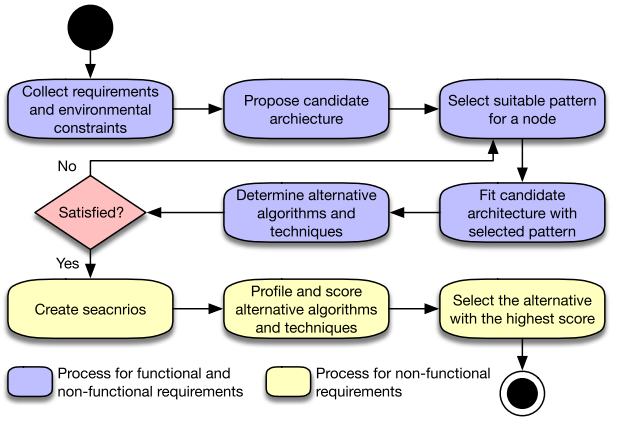


Fig. 4. ***Guideline of selecting self-awareness architectural pattern and the underlying algorithms/techniques.***

Table 2 Classification of the Representations on Domain Expertise With Possible Examples

Category	Example Expertise Representations
Methodology	RUP [53], agile [54], SSADM [55], SCURM [56], ...
Concept	technical debt [57], code smell [58], software entropy [59], feature creep [60], ...
Model	feature model [61], goal model [62], UML [63], Markov model [64], Petri net [65], queuing model [66], I* [67], viewpoints model [68], design patterns [69], ...
Documentation	SLA [70], requirement documents [71], user manual, configuration files, API documents, software and system specifications, ...
Program	source code of one (or more) programming language, library invocation and dependency, ...
Assumption	past problem instances and experiences, insights from peer and users discussions, ...

based on the profiling results. The final outcome for each node, after a satisfied number of iterations, would be the instantiation of a selected self-awareness architectural pattern with chosen underlying algorithms/techniques for self-awareness. Due to limited space, we advise interested readers to our handbook [8] for detailed information.

V. DBASES FOUNDATIONS

A. Representations of Engineers' Domain Expertise

As mentioned in Section II-A, for domain information, it is important to distinguish between problem nature and domain expertise; the former is not necessarily equivalent to the latter. Domain expertise, particularly, that from the software and system engineers, can be represented in various forms. For the simplicity of exposition, we use the following terminology to explain this concept in DBASES.

1) *Expertise Representation:* Expertise representation is generally abstract, which can be further refined and customized for expressing the domain expertise that captures domain knowledge for a specific case. These are often the general skills and tools that are familiar to a software and system engineer. For example, feature model is a representation of the expertise, which is commonly used by software and system engineers. It can be applied to a wide range of application domains within each of which the representation would be specialized into a particular design instance.
2) *Category of Expertise Representation:* This refers to a group of expertise representations that share similar nature, for example, the feature model, unified modeling language (UML) models, and the goal model are all design models.

Clearly, an expertise representation can be specialized into different instances that share the same structure, rules, and semantics, but each can capture/be tailored to handle different knowledge about the domain. Drawing on the recent survey about what expertise knowledge has been considered in practically engineering self-awareness [27], [28], DBASES is underpinned by a classification, as shown in Table 2, to categorize the most commonly used expertise representations when engineering software systems.[3] Each of the categories is explained as follows.

[3]The examples here do not intend to be exhaustive, but they serve as intuitive illustrations of the concepts.

1) Methodology: This refers to the systematic specification and analysis methods that are applied to abstract the expertise and represent it to aid the development of software systems. An expertise representation can be considered in this category if all of the following criteria are met.

1) It covers all or nearly all the phases in engineering a software system.
2) It contains specific methods, rules, postulates, procedures, or processes to manage a software or system project.
3) It involves description about the roles of different stakeholders in the engineering process, for example, analysts, designers, and testers.

2) Concept: This includes the intents, drivers/forces, and motivations that derive the knowledge/expertise capture and representation. An expertise representation stands as a `Concept` if all of the following criteria are met.

1) It represents an abstract idea or generic notion in mind that captures some common and justifiable phenomena of different instances in software and system engineering.
2) It aims to describe an idea or notion in a "plain" way that is intuitive and close to the general understanding of human.
3) It is a widely recognized practice and truth in the engineering process.

3) Model: This involves the standard for abstracting the expertise; it can systematically capture at least certain aspects of a software system, which are mainly utilized during the analysis and design phases. An expertise representation belongs to `Model` if all of the following criteria are met.

1) It contains a formal notation or language to describe how knowledge about the software system can be captured.
2) It can represent certain aspects of the software system and the relationships between them.
3) It is a more formal way of representing concept(s).
4) It is often illustrated in a graphical manner.

4) Documentation: This refers to artifacts that document and express the metadata for the representations of expertise, specifying scope, constrains, uses, anti-uses, etc., with an aim to be understandable for different stakeholders

(e.g., end users, managers). An expertise representation belongs to `Documentation` if all of the following criteria are met.

1) It contains metadata provided on digital or analog media.
2) It aims to illustrate data or represent agreement between parties for the software system.
3) It is entirely (or mostly) based on "plain" textual language of human.

5) Program: This involves the expertise representations that actually enable the software system to run. An expertise representation is related to `Program` if the following criterion is met.

— It is related to the source code that enables the execution of the software system.

6) Assumption: This refers to the expertise representations that are directly derived from the subjective beliefs and experience of the software and system engineers, which may not be well-justifed. An expertise representation can be considered in this category if all of the following criteria are met.

1) It is a general belief about the software system derived from specific instances.
2) It represents the sense of expectation on certain aspects of the software system, which is not guaranteed to be true.

The above classification in DBASES does not aim to be exhaustive, but they serve as a general guideline that covers majority of the cases, and thereby it can be flexibly extended. It may be possible that a given representation of expertise can fit more than one categories, in which case it is the engineer's decision on which one is more suitable. Similarly, it is also possible that a representation cannot be fit into any category above. In such a case, the representation can form an additional category (e.g., `Other` category), which can then be considered under the criteria of structurability and tangibility that we will elaborate as follows.

B. Structurability and Tangibility

The expertise representation expresses knowledge can be a result of one's experience, which would be in various forms. Therefore, it is also important to understand whether these representations are structural and tangible, as studied in human cognition research [71]. In a software and systems engineering domain, it is not uncommon to see that more structural representations can be more beneficial [72], and more tangible ones are better tools to express knowledge [73]. Therefore, given the variety of different expertise representations, explicitly recognizing their category in terms of structurability and tangibility is important. A structural representation means that the organization of its information follows specific rules, or semantics; otherwise, it is said to be nonstructural.

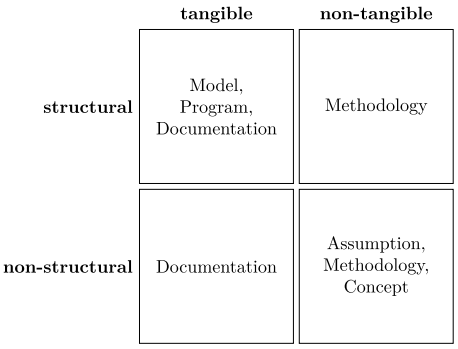


Fig. 5. ***Confusion matrix on the taxonomy of the expertise representations with respect to structurability and tangibility.***

Specifically, a given representation of expertise is structural if all of the following criteria can be satisfied.

1) Its organization and arrangement of the internal elements (and their relations) form some repeatable patterns.
2) It can be specialized into case-dependent variants, which, although different, can still be derived from the same core.
3) It contains explicit, step-by-step, information about how itself can be "assembled."

A tangible representation refers to the expertise representation that is perceptible by directly interacting and/or observing; or otherwise it is nontangible. Again, a representation of expertise is tangible if all of the following criteria can be satisfied.

1) It can be directly seen or touched to understand the information it holds.
2) It comes with a digital or analog media.

In Fig. 5, we further taxonomize the aforementioned six categories of expertise representations depending on their nature with respect to the above criteria of structurability and tangibility, as part of DBASES. The taxonomy provides a more intuitive way for the engineers to understand how a category can be linked to these two properties. However, it is worth noting that any given expertise representation can be assigned using the above criteria.

It is clear that expertise representations in the category of `Model` and `Program` are both structural and tangible, as they can easily meet all the criteria mentioned above. On the other extreme, representations in the category of `Assumption` and `Concept`, as the name suggests, are both nonstructural and nontangible. Because they cannot be derived from the same pattern, and are difficult to be seen or interacted with directly, which have failed to meet the criteria for being structural and tangible.

`Documentation` contains expertise representations that can be directly seen and comes with a media, thereby they are tangible, but could be structural or nonstructural. For example, service level agreement (SLA) and application program interface (API) documents also satisfy the three criteria of being structural. In contrast, requirement

Table 3 Possible Synergies Between Expertise Representation and Self-Aware Capabilities

Category	Example Expertise Representations	Self-Aware Capabilities
Methodology	SSADM [55]	stimulus-, time-, interaction-, goal- and meta-self-awareness
	SCURM [56]	stimulus-, time-, interaction-, goal- and meta-self-awareness
	...	...
Concept	technical debt [57]	time- and goal-awareness
	code smell [58]	stimulus-, time- and goal-awareness
	software entropy [59]	time- and goal-awareness
	feature creep [60]	stimulus-, time- and goal-awareness
	...	...
Model	feature model [61]	stimulus-, time- and goal-awareness
	goal model [62]	stimulus-, time- and goal-awareness
	UML [63]	stimulus-, time-, interaction-, goal- and meta-self-awareness
	Petri net [65]	stimulus-, time-, interaction- and goal-awarness
	Markov model [64]	stimulus-, time-, interaction- and goal-awarness
	queuing model [66]	stimulus-, time- and goal-awareness
	design pattern [69]	stimulus- and goal-awareness
	...	...
Documentation	SLA [70]	stimulus-, time- and goal-awareness
	requirement documents [71]	stimulus-, time-, interaction-, goal- and meta-self-awareness
	API	stimulus- and goal-awareness
	...	...
Program	source code	stimulus-, time-, interaction- and goal-awareness
	library invocation and dependency	stimulus-, time- and goal-awareness
	...	...
Assumption	past problem instances and experiences	stimulus-, time-, interaction-, goal- and meta-self-awareness
	insights from peer and users discussions	stimulus-, time-, interaction-, goal- and meta-self-awareness
	...	...

documents and user manuals are nonstructural, whose content is documented by natural language without specific rules. Thereby they fail to meet the criterion that there are variants which can be derived from the same common ground.

The category of `Methodology` would have expertise representations that are nontangible as they cannot be directly observed. Yet again, they could be structural or nonstructural. For instance, the structured systems analysis and design method (SSADM) is a rather structural methodology and it satisfies all three criteria. In contrast, Scrum, which is a form of Agile methodology, does not contain explicit, step-by-step, information about its internal structure due to the need of being flexible. Therefore, Scrum is said to be nonstructural.

C. Relationship Between Expertise Representations and Capabilities of Self-Awareness

Expertise representations can be possibly synergized to inform, enrich, and/or refine the capabilities of self-awareness depending on the domains, and guided by the specific design of expertise representations. Drawing on the work reviewed by recent survey on engineering self-awareness [27], [28] and our understandings form the EPiCS project[4] [49], in Table 3, we illustrate some examples of the possible synergies with respect to the categories presented previously.

Given the openness of certain categories of expertise representations (e.g., `Methodology` and `Assumption`), the domain expertise can be potentially synergized to benefit all the possible capabilities of self-awareness. For example, the SCURM methodology can help to better understand the engineering process of the algorithms that realize certain self-awareness. In addition, the methodology also covers the management between incremental development and operation phase. This, for instance, can assist the meta-self-awareness to collect suitable data about the applicability of other self-awareness as the software system runs, and thereby providing readily available information to be discussed again in the next phase of incremental development.

For other cases, on the other hand, domain expertise knowledge can be only useful to certain capabilities. For example, domain knowledge expressed using feature models would be useful for stimulus-, time-, and goal-awareness but can be of limited help for interaction and meta-self-awareness. This is because it neither expresses information on the interactions between nodes of software systems nor provides foundations to reason about the needs of different self-awareness capabilities.

Another example is related to goal modeling. In particular, goal modeling and its various refinements can be synergized with the benefit of goal-awareness. The aim, for example, is to dynamically analyze the satisfaction of that goal, areas, and traces within the model that requires refinements and further elaboration to meet the goal. This can be supported by synergizing the goal model with the stimulus- and time-awareness which would enable better goal reasoning. However, the goal model itself does not often express information on interaction.

As mentioned, capturing and modeling the knowledge, expressed via domain expertise can take forms of structured or unstructured and tangible or nontangible, which

[4]http://epics.uni-paderborn.de/

is heavily influenced by the available representations of domain expertise for the engineering of the self-aware and self-adaptive software system. Arguably, the structured and tangible expertise representations are often more systematic means and disciplined approaches, while unstructured and nontangible ones can be naturally flexible for probing, learning, and cross-fertilization of expertise. In this regard, the structurability and tangibility can largely affect the design and maintenance difficulty of synergy, as we will discuss in Section V-F.

It is worth noting that the examples here are merely for the guideline on the possible synergies in DBASES, they do not mean to restrict one to follow a specific synergy if both the expertise representation and the related capability of self-awareness are available. Whether a synergy is needed, as well as the level and form of such synergy (as we discuss in the following), is highly domain-dependent.

D. Levels of Domain Expertise Synergy

Generally, the information possessed by an expertise representation can be synergized with a capability of self-awareness at different extents. However, given the complexity of expertise representation, as well as the underlying algorithms/techniques for self-awareness, the synergy of expertise with self-aware software system may require to be automatic depending on the level.

In DBASES, we propose and distinguish four hierarchical levels of expertise synergy with a self-aware capability, which can be flexibly selected and reasoned about given the requirements. The hierarchical levels are derived from our experience on working with industry practitioners form the EPiCS project [49], together with the work in recent survey on engineering self-awareness [27], [28]. It is known that hierarchical analysis is highly beneficial for classifying concepts in engineering software systems, especially when dealing with requirements of the engineering problems [74]. Specifically, inspired by the work of Berry *et al.* [75], we describe each level according to the aspects listed as follows.

1) *Motivation:* A scenario where the level is required.
2) *Criteria:* A set of criteria classifies the synergy to a particular level.
3) *Description:* A general elaboration of the characteristics of the level.
4) *Example:* An instance where the level has been used.

The levels are structured in an incremental way, that is, level 2 would retain all the properties of level 1 and level 0.

1) Level 0 of Synergy:

a) Motivation: This is the level such that there is no actual domain expertise synergy but could merely utilize the necessary information about the problem nature to achieve the most basic specialization of the AI algorithms. This is often the case when standard AI algorithms are directly applied.

b) Criteria: Since this is the most basic level of synergy (i.e., no synergy at all), and thus there are no criteria for this level, as in essence, any realization of the self-awareness is at least level 0.

c) Description: Here, the engineers may not (or only trivially) reason about the problem and thus there may be no expertise representations. The underlying algorithm and technique that realize a capability of self-awareness does not use any information derived from the domain expertise. At this level, the synergy is a manual process.

d) Example: Considering a distributed system, where there is a machine learning algorithm that learns what are the important nodes to be tuned, but if the nodes are simply taken from whatever nodes that are currently running, then here, information of the problem nature (the available nodes) is used in stimulus-awareness. However, there is a lack of human reasoning involved (thus no domain expertise). Therefore, in such a case, we still have level 0 of domain expertise synergy.

2) Level 1 of Synergy:

a) Motivation: Apart from the problem nature, which is often naturally intuitive with the problems, software and system engineering involves many cases where the detailed information is not obvious, which can only be made available through the expertise of engineers together with various tools and methods.

b) Criteria: Specifically, the synergy is at level 1 if the following criterion is met.

1) The expertise representation is specialized through in-depth reasoning according to the software system to be built.

c) Description: This is the most common level where there is a limited synergy between domain expertise and self-awareness. Here, the engineers do reason about the problem and there are certain expertise representations. However, there is no, or only trivial, machine reasoning on the reasoned expertise representation that aims to extract more meaningful information for a capability of self-awareness (and the underlying algorithm/technique), which is the key step to sufficiently synergize the expertise. At this level, the synergy can be either a manual or automatic process.

d) Example: For example, the produced feature model design is a representation of expertise after careful human reasoning, but if the goal-awareness simply embed all the features from the model to optimize, then it is clearly a level 1 of domain expertise synergy, as some information about the human reasoning is used (the features) while there is no further, nontrivial reasoning about the feature model itself.

3) Level 2 of Synergy:

a) Motivation: The expertise representation produced by extensive human reasoning is likely to be complicated and large, which may be an inevitable result for the software system that is built and evolved over years. In such a case, the useful information contained in the expertise representation is blur and difficult to be used directly.

b) Criteria: The synergy is at level 2 if all of the following criteria are met.

1) The expertise representation is specialized through in-depth human reasoning according to the software system to be built.
2) There is a nontrivial automatic process that extracts information from the expertise representation for the software system.

c) Description: In this level, the engineers are required to reason about the problem and produce certain representations of their expertise. There is also a need of further automatic machine reasoning, which extracts and synergizes the useful information of the reasoned expertise representation with the underlying algorithm and techniques for realizing self-awareness. However, the underlying algorithms and techniques do not need to be aware of the information about the expertise; they may operate as if there is no such information.

d) Example: For example, an engineer may reason about and produce a feature model, then, the model would be further reasoned and extracted, such that the irrelevant features for optimizing the software system are ruled out in the capability of goal-awareness. However, from the perspective of the search algorithm, it does not aware that the given features to tune have been tailored by the experts' specialized knowledge; it would merely operate as if those features were selected arbitrarily.

4) Level 3 of Synergy:

a) Motivation: Although most algorithm/techniques would work without changing their internal structure, it is often the case that when their internal components are tailored specifically with the extracted domain expertise, the expected results can be largely improved. Such a process is not essential, but desirable.

b) Criteria: In particular, the synergy is at level 3 if all of the following criteria are met.

1) The expertise representation is specialized through in-depth human reasoning according to the software system to be built.
2) There is a nontrivial automatic process that extracts information from the expertise representation for the software system.
3) The internal components of the algorithm are tailored, such that it can actively and directly exploit the information extracted from the expertise representation.

c) Description: This is the highest level of domain expertise synergy. Here, both human reasoning and automatic machine reasoning on the representation of expertise are needed. In addition, the underlying algorithm and technique for realizing self-awareness need to be tailored in a way that they can be aware of the experts' specialized knowledge, and thus promote more explicit reactions and exploitation of the expertise. This often implies a nontrivial consolidation to the internal components of the algorithm and techniques, which would make them less general but being more specific to the given problem.

d) Example: Considering a queuing model, which is analyzed and designed by the engineers, used to synergize with a tailored machine learning algorithm to offer better awareness of goal. In this case, the queen model has some parameters that can be tuned automatically. More importantly, the machine learning algorithm is aware of the expertise expressed in the model, such that the training and updating mechanism can be tailored by the queuing model, which will clearly influence the accuracy of learning.

It is worth noting that for all levels, self-awareness and self-adaptation are still achieved through the underlying algorithms and techniques, but their behaviors are guided by varying the amount of information about the engineers' domain expertise, as constrained by the corresponding level of domain expertise synergy.

E. Benefit Score on Synergy

Generally, it is expected that a higher level of synergy would lead to better quality of self-awareness, and eventually better results of self-adaptation. This is because the underlying algorithm and technique can be guided, or even consolidated, with the information of domain expertise to fit with the domain problem better. To support quantitative reasoning on the potential benefit for different levels of synergy in DBASES, each level can be assigned a numeric score as follows.

1) *Level 0:* benefit score $= 1.25$.
2) *Level 1:* benefit score $= 1.5$.
3) *Level 2:* benefit score $= 1.75$.
4) *Level 3:* benefit score $= 2$.

where the values are normalized into the range between 1 and 2 to assure numeric stability. Noteworthily, the scores are fairly flexible as they serve as indication on the relative rank between different levels. Hence, the above scores are default settings in DBASES where the margin between different levels is equivalent. It is, however, perfectly acceptable to ask the stakeholders and engineers to assign the relative benefits score depending on the needs, similar to what have been done in CBAM [41], as long as the ranking remains unchanged. For example, if one considers that level 3 is likely to obtain much higher benefits than the others, then one may assign the benefit scores from level 0 to level 3 as: 1.1, 1.2, 1.3, and 2, respectively.

Indeed, the proficiency would have a definite impact on the likely benefit, as immature expertise, for example, that from a naive or inexperienced engineer, would likely to mislead the algorithms and techniques for self-awareness and self-adaptation. To reflect on this, within the methodology we introduce in Section VII, the engineers are asked to weight the proficiency on the expertise representation and the underlying algorithms for self-awareness, based on which a more informed-decision of the synergy can be made.

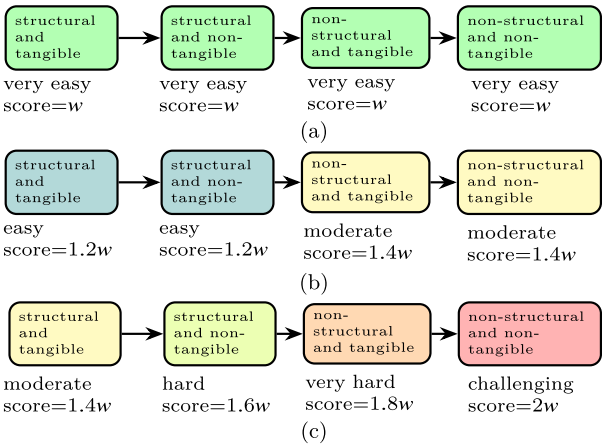


Fig. 6. ***Design difficulty and the related score of expertise synergy with respect to the levels, structurability and tangibility. (w denotes the weight (w ∈ [1, 2]) that distinguishes the difficulty between general and specific form of synergy for all cases. Level 0 always has a difficulty score of 1). (a) Level 1. (b) Level 2. (c) Level 3.***

F. Difficulty Score on Designing Synergy

In DBASES, the design of the synergizing domain expertise with a self-aware capability can be of either specific or general forms. In the specific case, one needs to analyze and reason about a particular instance of expertise representation (e.g., a design of feature model), and synergize it with a specific algorithm/technique (or any algorithms/techniques of the same type) that realizes self-awareness and self-adaptation. In the general case, the synergy needs to operate on different instances of expertise representation, for example, it works on any design instance of the feature model, and any algorithms/techniques of the same type. Undoubtedly, these forms do not applied on the level 0 of synergy.

It is clear that designing the general synergy would impose greater difficulty than the specific one, as wider range of the possible instances under the expertise representation needs to be considered. Here, the difficulty also serves as a general indicator of the cost in terms of labor, time, and resource for both implementation and maintenance, therefore, it is a crucial factor to consider when synergizing domain expertise. Within each of the two forms of synergies, the relative degrees of design difficulty vary depending on the levels of expertise synergy, as well as the structurability and tangibility of the expertise representation involved. Depending on different situations, the relative level of difficulty and the associated numeric scores have been illustrated in Fig. 6.[5] Note that the design difficulty for level 0 of synergy is constantly set as 1, that is, they are at most as hard as level 1 synergy even considering different forms, since there is no actual synergy at all.

More specifically, in Fig. 6, the difficulty is ranked in a way that it is consistent with the general understanding. At level 1, the synergy shares similar design difficulty regardless of the structurability and tangibility because at this level the main difficulty is related to the human reasoning of the domain knowledge, which is part of the tasks that the engineers have to do regardless whether there is a synergy. In particular, the algorithms and techniques for realizing self-awareness and self-adaptation are directly exploited to the domain, rendering the actual synergy relatively straightforward. At level 2, the synergy becomes more difficult in general. Particularly, the design difficulty becomes higher as the related expertise representation turns into a nonstructural form but remain unchanged with respect to the tangibility. This is because here, the underlying algorithms and techniques do not require to be aware of the domain expertise, thus the tangibility is less important. However, machine reasoning on the given expertise representation is necessary, therefore, the domain expertise needs to be made structural for the automatic reasoning and synergy to take place. Such an extra processing of structuring could impose additional design difficulty. Finally, at level 3, the expertise representation needs to be both structural and tangible, and thereby for expertise representation that belongs to the category of `Assumption` or `Concept`, additional efforts need to be conducted on both structuralization and tangibilization, rendering it as the most difficult case of synergy. Relatively, structuralization is more complex and difficult than tangibilization, as the former often requires in-depth and high proficiency on the expertise representation, whereas the latter, can be as simply as translating and documenting the concepts.

Based on the ranking, each case is assigned a numeric score to add quantitative values in the design process. The scores have been normalized in the range between 1 and 2, which can be used directly in the methodology discussed in Section VII. w is the normalized weight (between 1 and 2) that distinguishes the difficulty between general and specific form of synergy (e.g., 2 for general and 1.5 for specific) as provided by the engineers. Such a weight is applied to all possible synergy under consideration when engineering a self-aware and self-adaptive software system.

Noteworthily, similar to the benefit scores, the default margin between the difficulty scores of different cases is almost identical. However, it is perfectly possible that if one considers a case to be more difficult than the others and hence amend the margin, as long as the ranking is preserved.

Indeed, the actual synergy approach is highly domain dependent, relying on the selected underlying algorithms/techniques for self-awareness, the expertise representation that is available and the other constraints as well as requirements. Nevertheless, given the information about the expertise representation and the expected level of domain expertise synergy, the degree of design diffi-

[5]The illustration shows only relative degrees of design difficulty, i.e., a "very easy" does not means it is easy in an absolute sense, but it is relatively easier comparing with the others. Similarly, a "very easy" in the general synergy form is not equivalent to the "very easy" in the specific synergy form.

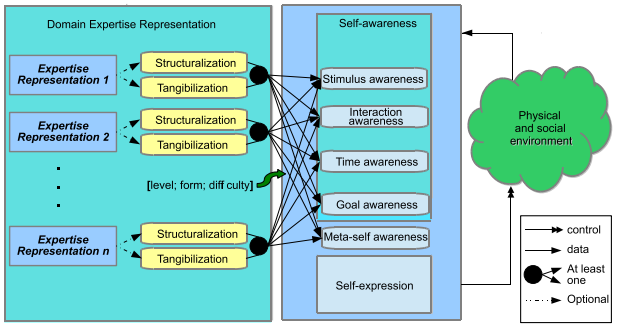


Fig. 7. ***Possible capabilities of self-awareness in the self-awareness architectural patterns with explicit synergy between domain expertise and self-awareness.***

culty offers the engineers with intuitive guidelines and information on the likely barriers, in addition to the likely benefits. This gives rise to the opportunity for them to rethink and even refine the level of expertise synergy at the design stage, considering the tradeoff between efforts and the expected quality. To demonstrate such details, in Section VIII, we elaborate examples of the synergy approaches within the contexts of three diverse case studies.

VI. ENRICHED SELF-AWARENESS PATTERNS IN DBASES

We now illustrate how the notions of domain expertise representations and their synergies in DBASES's foundation can be embedded with the capabilities of self-awareness, which are collectively expressed using the self-awareness patterns.

A. Capabilities of Self-Awareness in the Patterns With Explicit Domain Expertise

The proposed self-awareness architectural patterns, as discussed in Section IV, can be enriched based on the proposed synergy framework in Section V. Fig. 7 shows the general capabilities of self-awareness, which underpins the self-awareness architectural patterns, with explicit links to different expertise representations. Such a general enrichment can be instantiated into diverse instances, depending on the available expertise representation, the selected pattern, and the required synergy. Clearly, for a particular domain, there can be more than one expertise representation (from the same or different categories), but only one specialized instance of an expertise representation exists at a time. Those expertise representations, depending on their categories, may or may not undergo structuralization and tangibilization. Importantly, an expertise representation needs to be synergized with at least one capability of self-awareness (e.g., time and goal) and its underlying algorithm/technique. On the other hand, there is no cap on the maximum number of self-awareness capabilities that it can synergize with; it is possible that an expertise representation may be synergized with all the capabilities of self-awareness. According to Fig. 6, each synergy expresses the expected level involved, as well as the form and the design difficulty, which are separated by a semicolon.

Noteworthily, it is important to distinguish level 0 of synergy and no domain information is required. The former has no synergy but the information of the problem nature may still be used. The latter refers to no information is used in a self-aware capability at all. With the enriched pattern, level 0 is still expressed, but without showing the selection of form and the difficulty level. This becomes much more intuitive when instantiating the enriched self-awareness pattern with explicit domain expertise, which we elaborate in Section VI-B.

B. Examples of Instantiating the Patterns With Explicit Domain Expertise

In Fig. 8, we illustrate an example where the information sharing pattern and the related algorithms and techniques have been chosen. Then, following the general pattern from Fig. 7, the information sharing pattern can be instantiated with explicit domain expertise and the related synergies in different ways, among which Fig. 8 is one candidate. In this example, the expertise representation is a design of the Petri net that contains rich domain expertise about the concurrence and transitions between conditions, etc. This is particularly useful for the interaction-awareness and the underlying algorithm/technique, which enables a level 2 synergy between domain expertise and self-awareness. Specifically, the actual synergy can vary, for example, suppose a machine learning algorithm underpins the interaction-awareness to learn the likely under-utilized node for assigning more workloads. Here, the designed Petri net provides strong domain expertise about the features (conditions), which can be further parsed automatically to form a more relevant set of features. Finally, the resulted feature set is learned by the machine learning algorithm. This is clearly a level 2 of synergy, as there are both human and machine reasoning on the expertise representation, yet the machine learning algorithm itself does not know the fact that the given feature set was derived from domain expertise. There is no link between

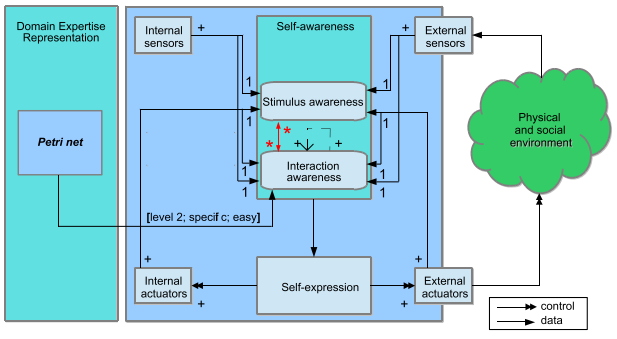


Fig. 8. ***Instantiating information sharing pattern with synergy between Petri net and self-awareness.***

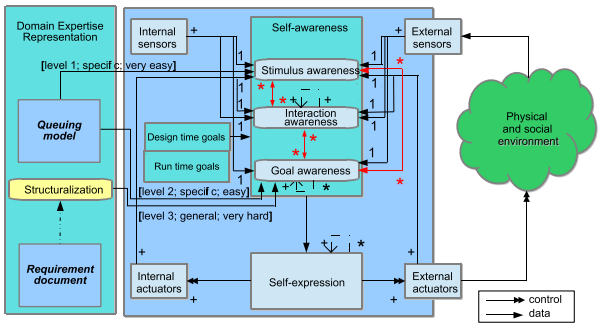


Fig. 9. *Instantiating goal-sharing pattern with synergies of queuing model and requirement document.*

a design of the Petri net and stimulus awareness, which means no information has ever been used for stimulus.

As expressed in the figure, the form of synergy is specific, which means only a design of Petri net needs to be synergized with the capability of self-awareness. Given that the Petri net belongs to the category of `Model` which is both structural and tangible (as seen from Fig. 5), there is no additional structuralization and tangibilization. The design difficulty of synergy is therefore "easy" according to Fig. 6. In contrast, if the expected level of domain expertise synergy was level 1 or level 3, then the design difficulty would become "very easy" or "moderate," respectively.

Fig. 9 shows a slightly more complicated example, in which the goal-sharing pattern and the related algorithms and techniques have been selected. In this example, the goal-sharing pattern can be instantiated with two aspects of domain expertise that are of different expertise representations and from distinct categories. Again, there could be different ways of synergies depending on the form and level, within which Fig. 9 illustrates only one candidate. Specifically, the design of queuing model is clearly a type of model while the requirement document belongs to the `Documentation` category. There are three synergies of domain expertise, each of which belongs to a different level. At the simplest form, the queuing model can create a level 1 of synergy with stimulus-awareness. This can be, for example, the feature components of the model serves directly as the detection points of any stimulus from the software systems, and therefore no extra reasoning and analysis conducted on the produced queuing model. Another synergy is between the queuing model and the goal-awareness, which can be of level 2. Here, certain parameters in the designed queuing model may be changed dynamically, either by a deterministic or machine learning algorithm. The tailored model, in turn, acts as the function to evaluate an adaptation solution within a search algorithm that optimizes toward the optimality of a goal. Such extra reasoning conducted on the queuing model has promoted the synergy to level 2.

In this example, the requirements document requires a relatively more complex, level 3 synergy with the goal-awareness. For example, the negotiated requirement document may be further analyzed using techniques for natural language processing, then, the results are synergized with the internal structure of a search algorithm, for example, to form tailored operators. In this way, the synergized expertise is fully aware by the underlying algorithm that realizes goal-awareness, which can explicitly react to the knowledge of expertise. This is aligned with the criteria of level 3. Again, those missing links between an expertise representation and a capability of self-awareness implies that there is no information to be used at all.

In this case, the queuing model can be linked with specific form of synergies while the requirements document requires the general form, in which case any given formats and designs of the requirements document need to be synergized with the self-awareness capability, and thus it is relatively harder. The relative design difficulty for all three synergies can be distinguishable using Figs. 5 and 6. A queuing model is both structural and tangible, and thus no extra processes are needed, therefore the level 1 synergy has a design difficulty of "very easy" whereas the level 2 one is classified as "easy." The synergy related to the requirements document is more complex, as it belongs to the `Documentation` category and it is tangible but nonstructural. As a result, given the required synergy of level 3, the relative design difficulty is "very hard." Note that since the requirements document requires general forms for its two synergies, they are likely to be more difficult than the specific one for the queuing model.

VII. METHODOLOGICAL ANALYSIS IN DBASES

Drawing on the aforementioned notions and enriched patterns, in this section, we codify a detailed methodology, as part of DBASES, that can assist the quantitative design on the synergy when engineering self-aware and self-adaptive software systems. In a nutshell, the methodology contains the steps below, the details of which will be explained in Sections VII-A–VII-E.

1) *Patterns and Algorithms:* Selecting patterns and algorithms.
2) *Representations of Expertise:* Determining the available representations of expertise.
3) *Candidates Creation:* Creating design candidates by instantiating the selected enriched pattern with synergy between domain expertise and self-awareness.
4) *Difficulty and Benefit Scores:* Calculating the overall difficulty and benefit scores for all the candidate synergies of expertise under the chosen pattern.
5) *Further Investigation:* Selecting the suitable candidate(s) for further investigation.

A. Patterns and Algorithms

The first step is to determine which is the suitable architectural pattern for self-awareness and the underlying

algorithms/techniques[6] that realize the self-aware capabilities. As mentioned in Section IV, we have proposed a handbook, together with a comprehensive guideline to guide the engineer to make such selections. A more thorough explanation and case studies can be found in the handbook [8].

B. Representations of Expertise

The actual representation of domain expertise is highly dependent on the case, and thus their diversity can vary. However, arguably any given software and systems engineering would require at least one formal representation of expertise. In this step, we ask the engineers to create a list of all available representation of the expertise based on their existing knowledge, some of which could be taken from the examples in Table 2.

C. Candidates Creation

According to the available representations of expertise identified in step 2, this step aims to answer the following questions for each of these representations.

1) Which category does the expertise representation belong to? (using the criteria in Section V-A)
2) If such a representation structural? Is it tangible? (using the criteria or classification in Section V-B)
3) The expertise representation can be synergized with which algorithm/technique that realizes the self-aware capability? What are the possible levels of synergy? (using the criteria in Section V-D)
4) What is the possible form for each synergy?
5) What is the difficulty level for each synergy? (using the Fig. 6)

Note worthily, the different synergies of expertise representations and their combinations form the possible alternative instantiations of the enrichment for the selected pattern, as shown in Section VI. In this way, step 3 aims to create a candidate set of instantiations for the enriched patterns with information about all possible ways of synergies. For example, suppose that there are two expertise representations and the chosen pattern is information sharing pattern, which has two self-aware capabilities. If both representations need to be synergized with all self-aware capabilities while the synergy can be at all levels and under both forms, then considering all possible combinations, the outcomes of step 3 would be $2 \times 4^4 = 512$ candidates. The final selection would be made based on the quantitative scores on both the difficulty and benefits for all the alternative candidates.

D. Difficulty and Benefit Scores

In this step, we aim to visualize the difficulty and benefits score for all the candidates identified from step 4 using the synergy framework. In particular, the overall difficulty of a candidate C_n that has a total of n synergies is calculated as

$$\mathbb{D}_{C_n} = \sum_{i=1}^{n} \frac{d_i}{p_i} \tag{1}$$

whereby d_i is the original difficulty score for synergizing the corresponding expertise representation in the ith synergy. As mentioned in Section V-F, the original difficulty score has been predefined according to the structurability and tangibility of the representation. The w is a normalized weight given by the engineers and it is applicable to all other synergies. p_i is the proficiency on the ith synergy (normalized between 1 and 2), which covers both the expertise representation and the underlying algorithms/techniques that realize the corresponding self-aware capability. The higher the proficiency, the less difficulty for achieving the synergy.

The overall expected benefit of a candidate C_n can be computed as

$$\mathbb{B}_{C_n} = \begin{cases} \sum_{i=1}^{n} p_i \times b_i, & \text{at level 0} \\ \sum_{i=1}^{n} w \times p_i \times b_i, & \text{otherwise} \end{cases} \tag{2}$$

where b_i is the original expected benefit score for the ith synergy, as discussed in Section V-D. Again, w and p_i are the actual forms (i.e., general or specific) of the synergy and the proficiency, respectively. The higher proficiency, the larger the expected benefit.

As mentioned, the values of w and p_i entirely depend on the domain, and therefore it is difficult to draw any general guidelines. However, their relative settings can be discussed case by case. For example, the relative w between general or specific form of synergy may be small for structural and tangible expertise representation, as it is more straightforward to generalize it from specific cases; in some situations, the w may be identical as the two forms may not differ too much, such as queuing model. In contrast, for nonstructural and/or nontangible expertise representations, their margin of w can be amplified. As regards p, the category of domain expertise representation and the selected algorithms, together with the engineer's own experience, can provide indication about how its value for different synergies can be relatively set. For example, an engineer who works on software variability management and machine learning algorithms for years would likely to rank a high proficiency for the synergy between feature model and learning algorithm, but for other synergies, the proficiency can be given a relatively low value.

E. Further Investigation

As we can see from the example of benefit/difficult plot in Fig. 10, each candidate is an instantiation of the selected

[6]One may only need to decide the type of algorithms, rather than a specific one.

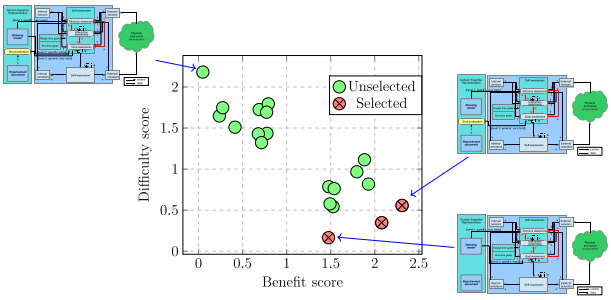


Fig. 10. ***Example of visualizing the difficulty and benefit scores for all candidates.***

and enriched patterns with a particular way of synergizing domain expertise. While some of the candidates are clearly dominated by the others, there can be a tradeoff between the difficulty and the expected benefit.

Indeed, to physically validate whether the achieved benefits and incurred difficulty (in terms of both implementation and maintenance) by the candidates are truly acceptable, it is an ideal case that if all the candidates can be subject to further investigation and profiling, that is, the actual implementation, profiling, and evaluation. However, given the time/resource constraint in real-world software and system development scenario, it is often the case that only a handful of them can be prototyped [41]. This is in fact what we seek to provide with the engineers: an intuitive and principled guideline to extract the candidates for further investigation. In DBASES, the intuitive visualization of benefit/difficult plot provides the necessary foundation for the engineers to select only the most desirable ones, ruling out those that are clearly unneeded and thus saving the valuable human efforts in investigating them. As an example, Fig. 10 shows three selected candidates for further investigation.

Noteworthily, despite there may be more than one way to implement the prototype of a particular candidate, it is generally possible to use a representative in the comparison process during further investigation, as what has been done in the domain of architecture profiling [41].

After further investigation, the final selection for production would inevitably involve not only the engineers, but also other stakeholders of the software systems. However, the methodology in DBASES, supported by the framework about synergy between domain expertise and self-awareness, their levels of difficulty and the enriched patterns, have enabled a more intuitive and quantitative visualization of all the possible alternatives in the tradeoff. This, in turn, provides better informed decision making when synergizing domain expertise with the self-awareness in software systems.

VIII. CASE STUDIES: PRACTICAL APPLICATIONS OF DBASES FRAMEWORK

In this section, we illustrate the practical applications of DBASES framework on three tutorial case studies, each of which is recent research effort that seeks to engineer self-awareness into different types of software systems. Those studies are the collaboration in a team of researchers and engineers from China and United Kingdom, under the funding grants from their research councils.

As part of the methodology in DBASES, for all case studies, a set of desirable and representative candidates was selected for further investigation. This includes the actual prototype implementation of these candidates, deploying the resulted self-aware and self-adaptive system(s), running them and measuring their behaviors according to various quality indicators with real-world benchmarks. Eventually, the most promising one with the verified results would be chosen for production.

Each case study covers a type of software systems that may be applied to different scenarios, for example, a highly constrained software system may run as a web-based systems or service-based systems. Therefore, for the candidates that are subject to further investigations, their prototypes were run on one or more scenarios. All the quantitative experiments are done in real environments, using the actual software system that fit under a given scenario. The data and source code used for all three case studies are publicly available on GitHub.[7]

A. Self-Awareness and Self-Adaptation for Highly Constrained Software Systems

Context: Self-adaptive software systems often have several nonfunctional quality attributes (e.g., latency and throughput), which are difficult to manage due to the changing environment, such as workload. Those software systems are centralized, but structurally complex, that is, there is a large number of features and complex dependence constraints. A typical example could be the multilayered web applications, in which the actual software often relies on a stack of third party libraries and frameworks, each of which has its own different adaptable features that can interplay together to influence the behaviors of the entire software system.

Problem: The aim of the first case study is, at runtime, to achieve more effective multiobjective optimization on the nonfunctional qualities of software systems. Clearly, in such context, self-awareness offers stronger capability for a software system to conduct more informed optimization and reasoning.

Challenges: The challenges here are twofold.

1) It is difficult to effectively and systematically convert the design of self-adaptive systems, expressed as a feature model, to the context of a search algorithm while considering the right encoding of features in the solution representation. This is even more complex in the presence of feature dependence constraints, e.g., the cache size can only be adapted when the cache feature has been "turned on," or, the size of a thread

[7]https://github.com/taochen/ssase/tree/master/experiments-data

pool needs to be equal to or greater than the number of spare threads in the pool.

2) Optimizing multiple conflicting objectives and managing their tradeoffs are complex and challenging in self-adaptive systems, especially at run time. This is attributed to the huge number of alternative adaptation solutions that can vary with their quality for the said requirements. Moreover, the dynamic and uncertain nature of self-adaptive systems further complicates the conflicting relations between objectives, rendering the tradeoff surface difficult to explore.

1) Patterns and Algorithms: After analyzing the requirements and following the handbook [8], it has been identified that there is no need to have knowledge about the interactions. This is because the target software system was not aimed for distributed environment, and that it is considered as satisfactory to optimize the local goal for a single self-adaptive system. Furthermore, the environment is not expected to actively react on the adaptation of the software system, and thus no interaction between it and the environment. There is also no need for meta-self-awareness, because the extra overhead on reasoning about the different capabilities of self-awareness is unnecessary, as the requirements on the capabilities are clear. In contrast, goal-awareness is the essential part as it permits capability to reason about goal and search toward an optimal (or near-optimal) solution. Time-awareness is also important in the modeling of goal, which consolidates the capability to thoroughly evaluate, and even predict, the effectiveness of a solution during the optimization process. As a result, these have led to the conclusion that the temporal goal aware pattern is the most appropriate pattern for the design. The pattern has been illustrated in Fig. 3.

The primary goal is to optimize nonfunctional quality, and thus a vast of search algorithms is available. However, there may be an explosion of the search space for the self-adaptive system, which renders the problem as intractable. Furthermore, it is difficult, if not possible, to obtain a precise understanding on the nature of the optimization problem beforehand and there are often multiple conflicting quality to be optimized. Drawing on these and as guided by the handbook [8], it has been concluded that the metaheuristic algorithms, particularly the evolutionary algorithms, are promising to realize the capability of goal-awareness in the software systems. However, given the wide range of possible domains, it is expected that the solution does not tie to a specific evolutionary algorithm, rather, it should support a diverse set of evolutionary algorithms. In addition, machine learning algorithms and other modeling techniques can be used to support the knowledge of time, which form the objective model that is essential in the reasoning of goal. Finally, stimulus-awareness, which is the simplest capability of self-awareness, can be realized by periodic detection. A complete list of the algorithms and techniques that realize the capabilities of self-awareness involved are shown in Table 4.

Table 4 Algorithms and Techniques That Realize the Capabilities of Self-Awareness for the First Case Study

Self-Awareness	Algorithms and Techniques
stimulus-awareness	periodic detection
time-awareness	machine learning/analytical model
goal-awareness	evolutionary algorithm

2) Representations of Expertise: In this case, the only available representation of expertise is the feature model [76], which is expressed as the tree structure. Such a model is widely used for software and systems engineers to represent the functional variability of software. In the context of self-adaptive software systems, the inherited concept of a feature model allows it to define the extent to which the software system is able to adapt at runtime (i.e., a range of variations that the software system can achieve). In particular, there is no definite constraint about the level that the feature model can cover, that is, the features define the prominent or distinctive aspects between different variations of a software system [60], which range from high-level architectural elements (an entire component) to low-level configurations (a specific parameter). Fig. 11 shows an example of the feature model, where there are four in-branch dependences and two cross-branch dependences.

1) OPTIONAL refers to the feature that might be "turned off."
2) MANDATORY denotes core features that cannot be "turned off."
3) XOR represents the feature in a group such that exactly one group member can be "turned on."
4) OR means a group in which at least one group member needs to be "turned on."
5) F_i REQUIRE F_j means the former can only be "turned on" if the latter is "turned on."
6) F_i EXCLUDE F_j denotes two features that are symmetrically mutually exclusive.

3) Candidates Creation: At this step, all the possible ways of synergy can be created by instantiating the enriched self-awareness architectural pattern. In particular, answers to the questions presented in Section VII are shown as follows.

1) Which category does the expertise representation belong to?

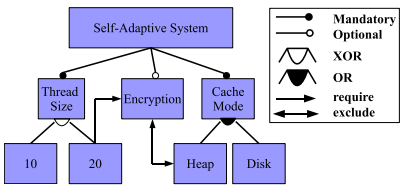


Fig. 11. ***Feature model.***

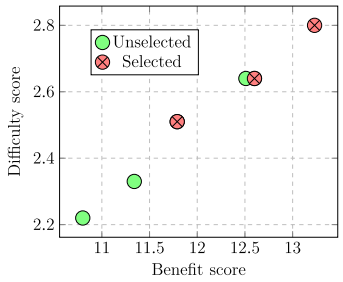


Fig. 12. ***Difficulty and benefit scores for all candidates in the first case study.***

— *Answer:* Feature model belongs to the `Model` category.

2) If such a representation structural? Is it tangible?
   — *Answer:* It is both structural and tangible.
3) The expertise representation can be synergized with which algorithm/technique that realizes the self-aware capability? What are the possible levels of synergy?
   — *Answer:* It needs to be synergized with all three self-aware capabilities in the enriched temporal goal aware pattern. However, the synergy can only be at level 1 to the stimulus awareness but level 1 and level 2 are allowed for time awareness. For goal awareness, all levels except level 0 are possible, but level 2 and level 3 would require the synergy with time awareness to be at level 2.
4) What is the possible form for each synergy?
   — *Answer:* Only the synergy with goal awareness can be of both specific or general form. The others are to be realized in a general form.
5) What is the difficulty level for each synergy?
   — *Answer:* According to Fig. 6, the difficulty level ranges between very easy to moderate.

The above answers have led to six different candidates of synergizing domain expertise represented as the enriched temporal goal aware pattern.

4) Difficulty and Benefit Scores: For all the six candidates, their overall scores with respect to both the difficulty and benefit are illustrated in Fig. 12. In particular, the w between specific and general form of synergy is set as 1.2 and 1.4, respectively, the proficiency is set as 1.8 for all synergies in a candidate.

5) Further Investigation: As shown in Fig. 12, after discussions, three candidates have been selected for further investigation, as they are either desirable or serve as representatives for the others. Briefly, each of the candidates is specified as follows.

1) C_1 ($\mathbb{B}_{C_1} = 13.23$, $\mathbb{D}_{C_1} = 2.8$): As shown in Fig. 13(a), the candidate automatically extracts only important features to create synergies in time- and goal-awareness, at level 2 and level 3, respectively. In particular, dependence constraints are also injected and synergized with the evolutionary algorithms that underpin the goal-awareness. No machine reasoning is required for stimulus-awareness, which senses directly on the features at level 1 of synergy.
2) C_2 ($\mathbb{B}_{C_2} = 12.6$, $\mathbb{D}_{C_2} = 2.64$): The candidate, illustrated in Fig. 13(b), achieves the synergy of the feature model with goal-awareness at level 2 (general form), such that the evolutionary algorithm is not aware of the dependence constraints; all the other synergies remain the same as that of C_1.
3) C_3 ($\mathbb{B}_{C_3} = 11.79$, $\mathbb{D}_{C_3} = 2.51$): Fig. 13(c) illustrates the candidate in which the feature model is synergized with time- and goal-awareness at level 1 (general form), that is, all possible features are selected

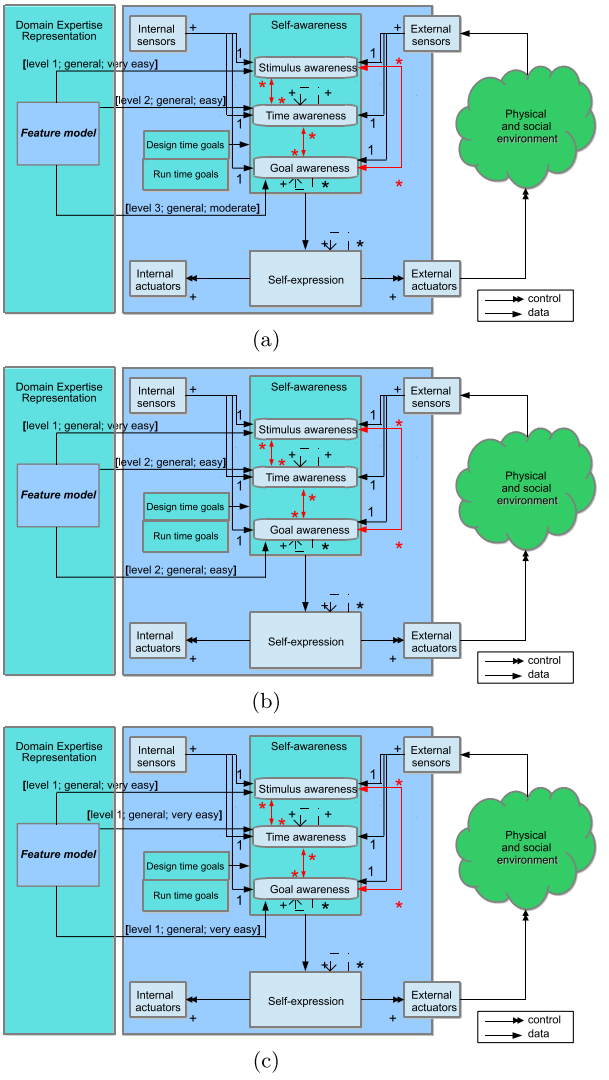


Fig. 13. ***Possible candidates selected for further investigation in the first case study. (a) Candidate C_1. (b) Candidate C_2. (c) Candidate C_3.***

Table 5 Real Subject Software Systems for the Experiments of the First Case Study

	Objective	#Feat.	#Dep.	Env.	Space
RUBiS	latency; power	1,151	89,736	workload	1.3×10^{16}
SOA	throughput; cost	221	255	services	5.6×10^{18}

* Feat. denotes features; Dep. denotes dependencies; Env. denotes environment; Space denotes search space.

to be tuned without further parsing of the feature model, and no dependence constraint is captured by the evolutionary algorithm. All the other synergies remain the same as that of C_1.

More technical details on the actual synergy approaches can be found in [23].

6) Further Investigation Setup: Since all the synergies are in general form, the candidates are evaluated on two different real subject software systems, namely RUBiS [77] (a web system) and SOA [33] (a service system), under two distinct categories of evolutionary algorithms, that is, Nondominated Sorted Genetic Algorithm II [78] (NSGA-II) and Indicator-Based Evolutionary Algorithm [79] (IBEA), for realizing goal-awareness that optimizes different conflicting quality objectives. The details of the two subject software systems can be found in Table 5. Given that the optimization occurs at runtime, the setup of both algorithms has been carefully tuned, such that the mutation rate is 0.1 and crossover rate to be 0.9, with 100 population size for ten generation. Each experiment is repeated 100 turns to cater for the stochastic nature of the optimization. For the time-awareness, machine learning model [13], [80], [81] is used for RUBiS and the analytical model [25], [33] is adopted for SOA. The results are statistically significant as confirmed by the Wilcoxon Signed Rank test (p <0.05) with nontrivial effect sizes (ESs), following the guideline provided by Kampenes *et al.* [82].

The quality indicators for benefit and difficulty are shown in Table 6. As can be seen, the benefits are assessed by various performance attributes, which are scenario-dependent, as well as the percentage of valid solutions found for adaptation. The difficulty is evaluated by using lines-of-code (LOC) of the implemented prototypes for the candidates, as it is a common metric to measure the complexity in software engineering. A higher LOC implies higher complexity in implementation and maintenance, hence higher difficulty.[8]

[8]Note that we do not include LOC for any third party libraries.

Table 6 Quality Indicators for Benefit and Difficulty for the First Case Study

Attribute	Quality Indicators
Benefit	latency, power, throughput, cost and % of valid solutions
Difficulty	Lines-Of-Code (LOC)

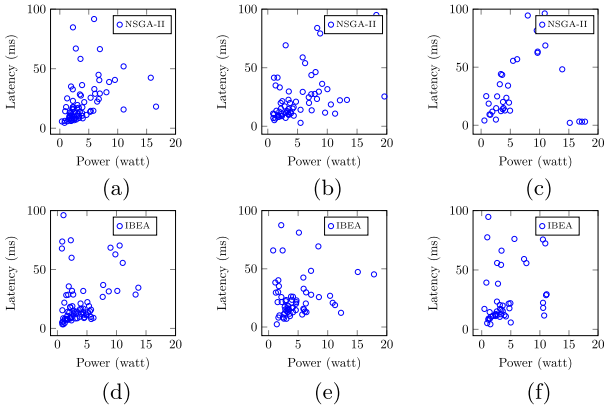


Fig. 14. ***Benefits on* RUBiS *under further investigated candidates over all runs. (a) C_1 (NSGA-II). (b) C_2 (NSGA-II). (c) C_3 (NSGA-II). (d) C_1 (IBEA). (e) C_2 (IBEA). (f) C_3 (IBEA).***

7) Results: As shown in Figs. 14 and 15, clearly, we see that for all cases, in contrast to C_3, C_2 finds more solutions that are condensed to the bottom-left (top-left for SOA) corner of the objective space. This means that more advanced synergy helps to enable more promising results in the optimization. When comparing C_1 and C_2, the solutions are even more condensed to the ideal corner under C_1, and is of particular significance in the case of SOA due to its stronger extents of conflicts. This proves that allowing the underlying algorithm for goal-awareness to be aware of the domain expertise, although impose higher design difficulty, can be very beneficial in terms of the results.

Fig. 16 illustrates the mean percentage of valid solutions found, and we see that the C_1 achieves 100% valid solution as the evolutionary algorithm is aware of the expertise about the dependence during the evolution, which promotes the ability to actively repair the solutions that violate dependence. C_2, on the other hand, do not have such benefits but it is more likely to result in valid solutions than

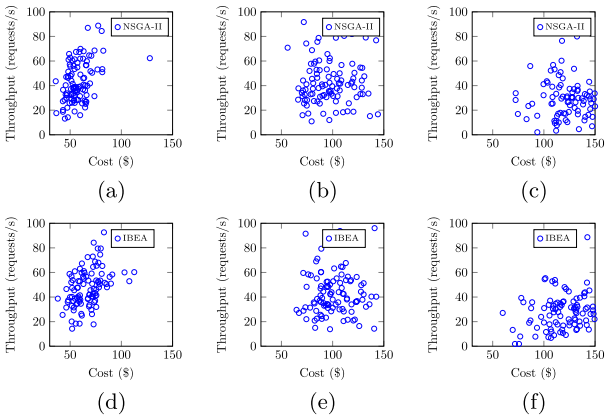


Fig. 15. ***Benefits on* SOA *under further investigated candidates over all runs. (a) C_1 (NSGA-II). (b) C_2 (NSGA-II). (c) C_3 (NSGA-II). (d) C_1 (IBEA). (e) C_2 (IBEA). (f) C_3 (IBEA).***

that of C_3. This is because C_2 encodes a set of automatically extracted and more elitist features to be tuned. C_3, in contrast, encodes all the features in the feature model, which can hardly find valid solutions given the high number of features in the subject software systems.

For the difficulty shown in Table 7, as expected, C_1 has the highest LOC which implies higher difficulty in implementation and maintenance. C_2 is ranked the second but its differences to C_3 are small, which suggests that the difficulty related to automatically extracting the important features is considerably low.

Final choice for the first case study:
According to the verified results from the further investigation, the team has decided that C_1 is a more preferred and promising choice for production.

B. Self-Awareness and Self-Adaptation for Change Expensive Software Systems

Context: Self-adaptive software systems may subject to financial contracts with respect to its performance and resource consumption to perform adaptation. For example, a software system deployed on the Cloud Computing platform are charged on the amount of resources it consumes, and it may incur monetary penalty (or reward) for violating (or exceeding) some agreed threshold of performance. In particular, the adaptations in the target self-adaptive software systems are often expensive, or the reasoning process related to the adaptation is resource consuming, and therefore in certain cases, it could be more beneficial to not adapting.

Problem: In the second case study, the aim is to dynamically determine when and whether to adapt those critical software systems for which adaptations can impose expensive cost. This again exhibits a strong requirement of self-awareness.

Challenge: The key challenge is how to model and reason about the dynamic and uncertain cost–benefit between adapting the software system and not adapting it, then deciding on when and whether to adapt. It is required to measure the software systems not only on the achieved quality of nonfunctional attributes, but also, in terms of the monetary values that it generates, or carry as debts.

1) Patterns and Algorithms: After analyzing the requirements and following the handbook [8], it has been concluded that there is no interaction awareness required, as the target software system was not an aim for distributed environment, and that it is considered as satisfied to

Table 7 LOC for the First Case Study

Candidate	LOC
C_1	80,718
C_2	74,092
C_3	73,466

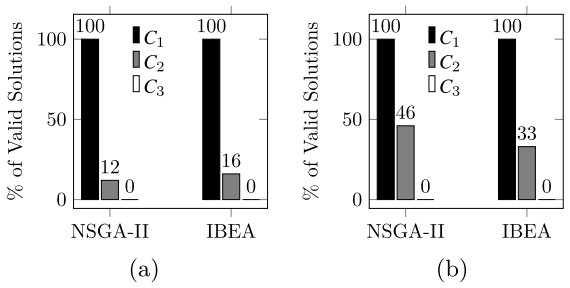


Fig. 16. ***Mean percentage of valid solutions found under further investigated candidates over all runs. (a)* RUBiS. *(b)* SOA.**

optimize the local goal for a single self-adaptive system. Furthermore, the environment is not expected to actively react on the adaptation of the software system, and thus no interaction between it and the environment. There is also no need for meta-self-awareness, because the requirements on the required capabilities are clear and that the problem itself aims to reduce the extra computations involved in the self-adaptation process. Therefore, the overhead produced by meta-self-awareness, which could be potentially high, should be better avoided. Indeed, the self-adaptive software system itself is often goal-aware due to the need of explicitly reasoning on the goals and objectives. However, for the problem that should be dealt with (i.e., when and whether to adapt), extensive reasoning on the goals is not the key purpose; rather, it is more related to track and make a binary decision: to adapt or not to adapt, drawing on insights about their time-varying cost-benefits. As a result, these have led to the conclusion that the temporal knowledge aware pattern is the appropriate pattern for the design. The pattern is illustrated in Fig. 2.

In this case study, the primary goal is to model the time-varying cost–benefit on the decision of adapting and not adapting the software system. Therefore, by following the steps in the handbook [8], machine learning algorithm has been identified as the promising way to handle the problem. This is because they are often effective in producing fast prediction in acceptable time, given that sufficient amount of past samples. Since there are only two decisions to model, the problem can be rendered as a binary classification problem, where, given a set of features (e.g., software system status and environment changes), the model aims to predict whether it is better to adapt or not. Again, given the generality of the target software system, the solution should not be specific to a particular machine learning algorithm, and thus it should support a wide range of the types, allowing for better flexibility on customization. As for the stimulus-awareness, it can be easily realized by periodic detection. A complete list of the algorithms and techniques that realize the capabilities of self-awareness involved are shown in Table 8.

2) Representations of Expertise: Here, there are two representations of expertise, namely the SLA and the technical debt concept.

Table 8 Algorithms and Techniques That Realize the Capabilities of Self-Awareness for the Second Case Study

Self-Awareness	Algorithms and Techniques
stimulus awareness	periodic detection
time-awareness	machine learning

In general, SLA is a formal legal binding negotiated between the software company and the end users before the software system is built [83]. An example fragment of the typical SLA, derived from the well-known WS-Agreement [84], is shown in Fig. 17, which states the rate of reward and penalty on the mean latency ($/s) and rate of CPU time of planning ($/s) for a software system. Specifically, the SLA states that the rate for the cost of adaptation is $0.345 per CPU second; and an adaptation that utilizes 2 s would lead to a total cost of $0.69. Similarly, the SLA may contain a penalty rate of mean latency violation as $0.043/s for a requirement of 2 s. Therefore, if there is a mean latency of 2.5 s for a period, then the penalty for it would be $(2.5 - 2) \times 0.043 = \$0.0215$.

Technical debt for software engineering was coined by Cunningham [56], to help deciding whether to improve the software, considering the costs and benefits of improvement versus that of not improving it. In general, when software faces bugs or requires improvement, the engineers have two options: 1) improve the software, in which case the quality of the software may be improved, but extra rework cost would needs to be paid for the human and resources spent or 2) leave it as it is, and thereby the software remain as flawed, which could accumulate the interests incurred by the bugs. The benefit of technical debt concept is that it offers an intuitive way for software and system engineers to make decision about whether to improve or not, and to track the debt over time.

3) Candidates Creation: At this step, the team creates all the possible ways of synergy by instantiating the enriched self-awareness architectural pattern. In particular, they answer the questions presented in Section VII as follows.

```
<wsag:GuaranteeTerm Name="Latency">
  <wsag:ServiceScope ServiceName="Adaptive System"/>
  <wsag:QualifyingCondition>
      {"function" : "AVG EVERY 100s"}
  </wsag:QualifyingCondition>
  <wsag:ServiceLevelObjective>
      <wsag:KPITarget>
        <wsag:KPIName>MeanTime</wsag:KPIName>
        <wsag:CustomServiceLevel>
            {"constraint" : "MeanTime LESS THAN 0.12s"}
        </wsag:CustomServiceLevel>
      </wsag:KPITarget>
  </wsag:ServiceLevelObjective>
  <wsag:BusinessValueList>
      <wsag:Penalty>
        <wsag:AssessmentInterval>
            <wsag:TimeInterval>100s</wsag:TimeInterval>
        </wsag:AssessmentInterval>
        <wsag:ValueUnit>USD_PER_SECOND</wsag:ValueUnit>
        <wsag:ValueExpression>2.76</wsag:ValueExpression>
      </wsag:Penalty>
      <wsag:Reward>
        <wsag:AssessmentInterval>
            <wsag:TimeInterval>100s</wsag:TimeInterval>
        </wsag:AssessmentInterval>
        <wsag:ValueUnit>USD_PER_SECOND</wsag:ValueUnit>
        <wsag:ValueExpression>1.13</wsag:ValueExpression>
      </wsag:Reward>
  </wsag:BusinessValueList>
</wsag:GuaranteeTerm>
```

```
<wsag:GuaranteeTerm
      Name="CPUTime">
  <wsag:ServiceScope
      ServiceName="Engine"/>
  <wsag:ServiceLevelObjective>
      <wsag:KPITarget>
          <wsag:KPIName>
              CPUTime
          </wsag:KPIName>
          <wsag:CustomServiceLevel>
              {"constraint" :
                  "CPUTime LESS THAN 0s"}
          </wsag:CustomServiceLevel>
      </wsag:KPITarget>
  </wsag:ServiceLevelObjective>
  <wsag:BusinessValueList>
      <wsag:Penalty>
          <wsag:AssessmentInterval>
              <wsag:Count>1</wsag:Count>
          </wsag:AssessmentInterval>
          <wsag:ValueUnit>
              USD_PER_SECOND
          </wsag:ValueUnit>
          <wsag:ValueExpression>
              0.0788
          </wsag:ValueExpression>
      </wsag:Penalty>
  </wsag:BusinessValueList>
</wsag:GuaranteeTerm>
```

Fig. 17. ***Fragment of an SLA.***

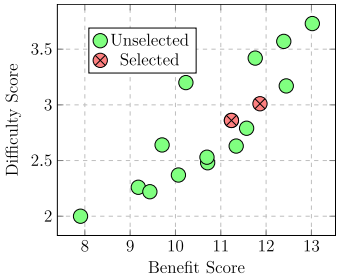


Fig. 18. ***Difficulty and benefit scores for all candidates in the second case study.***

1) Which category does the expertise representation belong to?
   — *Answer:* SLA belongs to the `Documentation` category but technical debt belongs to the `Concept`.
2) If such a representation structural? is it tangible?
   — *Answer:* SLA is both structural and tangible while technical debt is neither structural nor tangible.
3) The expertise representation can be synergized with which algorithm/technique that realizes the self-aware capability? What are the possible levels of synergy?
   — *Answer:* SLA needs to be synergized with both self-aware capabilities in the temporal knowledge aware pattern, but information and expertise related to technical debt need to be used with the time awareness only. The synergy between SLA and stimulus awareness needs to be at level 1, while for time awareness, it can be of any level (including level 0). Similarly, the technical debt can be synergized with time awareness at any level (including level 0).
4) What is the possible form for each synergy?
   — *Answer:* All the synergies need to be realized in a general form.
5) What is the difficulty level for each synergy?
   — *Answer:* According to Fig. 6, the difficulty level ranges between very easy to challenging.

The above answers produce 16 different candidates of synergizing domain expertise represented as the enriched temporal knowledge aware pattern.

4) Difficulty and Benefit Scores: For all the 16 candidates, their overall scores with respect to both difficulty and benefit are shown in Fig. 18. In this context, the ω between specific and general form of synergy is set to 1.2 and 1.4, respectively. For each candidate, the proficiency is set as 1.8 for all synergies related to SLA and 1.5 for those related to technical debt.

5) Further Investigation: Two candidates, as illustrated in Fig. 19(a), have been selected due to its superiority on the expected benefit over most other candidates, while

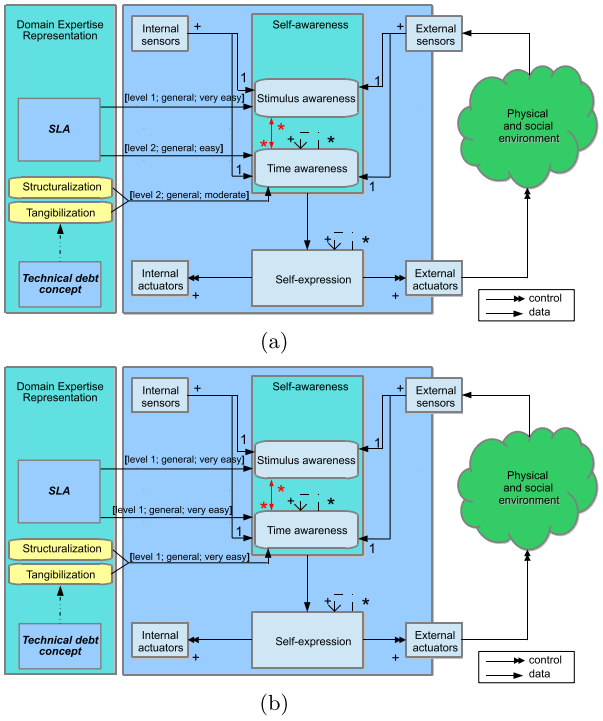


Fig. 19. *Possible candidates selected for further investigation in the second case study. (a) Candidate C_1. (b) Candidate C_2.*

causing an acceptable degree of difficulty. In a nutshell, they are discussed as follows.

1) C_1 ($\mathbb{B}_{C_1} = 11.865$, $\mathbb{D}_{C_1} = 3.01$): As shown in Fig. 19(a), the candidate automatically converts the technical debt concept into structural and tangible knowledge, which can be synergized with time-awareness at level 2. The SLA has also been parsed to extracted meaningful information to consolidate understanding regarding time as level 2 of synergy. The stimulus-awareness, however, directly use the information from SLA without additional machine reasoning at level 1 of synergy.
2) C_2 ($\mathbb{B}_{C_2} = 11.235$, $\mathbb{D}_{C_2} = 2.86$): For the candidate in Fig. 19(b), the synergy of domain expertise on the SLA and technical debt concept with the time-awareness are realized at level 1. In this way, the time awareness merely predicts the occurrence of an event, that is, violation of performance requirement (the only information from SLA), without further parsing on the SLA and technical debt. The prediction results are then further analyzed by statistical inference; thus only the significant, reliable, and persistent violations would trigger adaptation. All the other synergies remain the same as those of C_1.

More technical details on the actual synergy approaches can be found in [24].

6) Further Investigation Setup: The candidates are evaluated using RUBiS (detailed in Table 5) as the subject software system deployed and run in the real cloud environment, with the negotiated SLA shown in Table 9. Two distinct machine learning algorithms are run in parallel, that is, Naive Bayes [85] (NB) and multilayer perceptron [86] (MLP), each of which is of different complexities. The goal is to optimize the latency and power of the cloud-based software system, and thus both multiobjective planner (MOP) and single-objective planner (SOP), in which all objectives are combined in an equally weighted aggregation, are applied. The actual objective function to be optimized is trained based on machine learning [13], [80], [81]. However, it is worth noting that optimization is not a concern of the designed software system that is self-aware, as it is not part of selected the pattern. The number of repeated runs is 100 for the experiments, based on which the mean is reported. The results are confirmed by Wilcoxon signed rank test ($p < 0.05$), following the ESs categorization in [82].

Similar to the previous case study, the quality indicators used to assess both benefit and difficulty are shown in Table 10.

7) Results: From Table 11, we see that for all cases, the C_1 under all algorithms outperforms the C_2 on both quality attributes, with statistical significance and nontrivial ES on at least one attribute. In Fig. 20, we also observe

Table 9 Cloud-Based Software System for the Experiments of the Second Case Study

Attribute	Setup
Latency	threshold is 0.05s; reward and penalty rate are \$3.5 per unit
Power	threshold is 5 watt; reward and penalty rate are \$0.5 per unit
CPU time of adaptation	charge rate is \$0.01 per unit
Enviroment	workload

Table 10 Quality Indicators for Benefit and Difficulty for the Second Case Study

Attribute	Quality Indicators
Benefit	latency, power, debt, number and cost of adaptation
Difficulty	Lines-Of-Code (LOC)

Table 11 Benefits in Terms of Latency and Power, Their Statistical Significance and Effect Sizes (ESs) Over All Runs (Statistically Significant Comparison Between C_1 and C_2 Is Highlighted in Gray)

MOP	Latency (ms)		Power (watt)	
	Mean	p value (ES)	Mean	p value (ES)
C_2	2.69	-	3.58	-
C_1 (NB)	**0.19**	.003 (large)	4.10	.810 (trivial)
C_1 (MLP)	0.32	.017 (small)	3.42	.576 (trivial)
SOP				
C_2	3.32	-	5.06	-
C_1 (NB)	**0.19**	<.001 (large)	**3.22**	<.001 (large)
C_1 (MLP)	0.18	.108 (small)	3.64	<.001 (large)

Table 12 LOC for the Second Case Study

Candidate	LOC
C_1	69,343
C_2	65,671

that C_1 has led to less debt, meaning that the monetary value generated by the software system, after synergizing the domain expertise with automatic machine reasoning, is higher than the case when the synergy is limited. We can also note that such benefit is achieved by using remarkably smaller amount and cost of adaptation.

To gain a better understanding about the total debt, we plot the debt throughout for an entire run. Fig. 21(a) shows the cumulative distribution of debt for different levels of synergy, when using MOP. We can see clearly that, in contrast to others, C_1 with the two machine learning algorithms reduces the debt quicker as their slopes are much steeper than the C_2. Yet, the superiority of C_2 on debt reduction is much more obvious when the debt is greater than about \$9. Fig. 21(b) compares the cumulative debt of approaches when using SOP. Here, we see that C_1 is again significantly outperforms the case when there is no actual synergy, with faster reduction on the debt.

The difficulty in terms of LOC is shown in Table 12. As can be seen, C_1 requires higher LOC than C_2, which implies higher difficulty in implementation and maintenance. This is predictable based on the benefit/difficult plot. However, we did not expected that the margin is as little as 3672 lines, suggesting that the difficulty difference is in fact negligible given the much better benefits brought by C_1.

Final choice for the second case study:
According to the verified results from the further investigation, C_1 is deemed as more suitable and thus it is chosen for production.

C. Self-Awareness and Self-Adaptation for Rapidly Composed Software Systems

Context: Service systems, unlike the others, do not have the actual implementation. Instead, they have a set of abstract services, each of which can be adapted to select different concrete services published in the Internet, according to a given workflow with different predefined connectors (sequential or parallel) [25], [33], [87], [88]. Such a process, namely service composition, is the key to enable rapid realization and integration of different functionalities that are required by the stakeholders. This is also a benefit of service systems, such that they share some similarities which make the exploitation of past problem instances and experiences possible.

Problem: In the third case study, the aim is to conduct multiobjective optimization for rapidly composing self-adaptive service systems at runtime, leveraging the benefits from the capabilities of self-awareness.

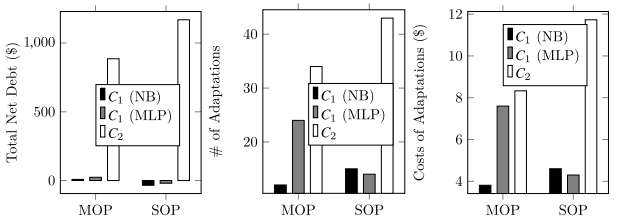


Fig. 20. *Total debt, number, and cost of adaptation under further investigated candidates over all runs.*

Challenge: The challenge here is that there is often a large number of services to fulfill the same functional requirement, but come with different levels on some possibly conflicting nonfunctional quality-of-service (QoS) attributes, for example, latency, throughput, and cost. Thereby optimizing and finding the good service composition plans, that is, a set of selected concrete services, and their tradeoffs becomes a complex and challenging problem which is known to be NP-hard [33], [89]. In addition, given the potentially rapid needs of composing the services, the optimization requires fast convergence to ensure the effectiveness of the optimized composition plan.

1) Patterns and Algorithms: Following the procedures from the handbook [8], it has been concluded that the requirements in this case study do not involve interaction awareness because there is no way to know in advance what are the concrete services available, thus there is often a service broker that act as a centralized point to compose a service system. Furthermore, the environment is not expected to react on the adaptation of the software system, hence no interaction between it and the environment. The meta-self-awareness has been ruled out as the requirements on the required capabilities is clear, and no need to introduce extra overhead. Goal-awareness is again essential in the optimization and time-awareness is also crucial for self-adaptive service systems because the currently available concrete services, as well as their QoS values, could change over time, and thereby requiring a model that cope with such a change. As a result, these have

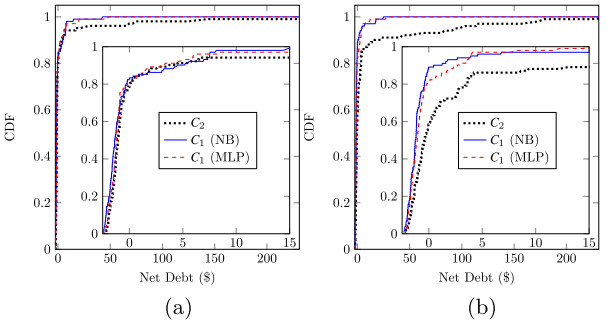


Fig. 21. *Cumulative distribution function of debt under further investigated candidates over all runs. (a) MOP. (b) SOP.*

Table 13 Algorithms and Techniques That Realize the Capabilities of Self-Awareness for the Third Case Study

Self-Awareness	Algorithms and Techniques
stimulus awareness	event driven detection
time-awareness	analytical model
goal-awareness	evolutionary algorithm

led to the conclusion that the temporal goal aware pattern as the appropriate choice for the design. The pattern has been illustrated in Fig. 3.

Given the NP-hard problem with an explosion of the search space and the nature of multiobjectivity for the self-adaptive services systems, the handbook [8] has suggested that the metaheuristic algorithms, particularly the evolutionary algorithms, are promising to realize the capability of goal-awareness in the software systems. Yet, given the high diversity of the workflow structures, it is expected that the solution does not tie to a specific evolutionary algorithm, rather, it should support a wide range of evolutionary algorithms. The time-awareness is supported by an analytical model, which tracks the available set of concrete services and their QoS values, and is capable of evaluating the aggregated QoS value for the workflow. The stimulus-awareness can be realized by event driven detection, such that the stimulus is captured through passive detection. A complete list of the algorithms and techniques, with respect to the capabilities of self-awareness involved, are shown in Table 13.

2) Representations of Expertise: There are two fundamental representations of the expertise in this case: the workflow structure of the service composition and past problem instances/experience about the optimization when composing services.

As shown in Fig. 22, where we can see that the workflow is represented as a graph and each vertex represents an abstract service. The edge denotes the connector between vertices, for example, they can be either sequential where the users' requests are proceed in strict order or parallel such that different users' requests are handled by simultaneously.

Another important representation of expertise is past problem instances and experience about the service composition. In the context of service composition, adaptation is required when change occur, for example, the QoS of concrete service changes or some concrete services becomes unavailable. These changes, albeit can occur rapidly, often occur in relatively small extents. As a result, past problem instances and experience can still provide useful information for the scenario after changes occur. For example, changes on the QoS for a few concrete services may not affect the search and objective space significantly. Furthermore, composition plans for service composition with similar workflow structure can also be rather useful.

3) Candidates Creation: At this step, the team considers all the candidates of synergy by instantiating the enriched self-awareness architectural pattern. In particular, they answer the questions as presented in Section VII as follows.

1) Which category does the expertise representation belong to?
   — *Answer:* Workflow structure belongs to the `Model` category but past problem instances/experience belongs to the `Assumption`.
2) If such a representation structural? is it tangible?
   — *Answer:* Workflow structure is both structural and tangible while past problem instances/experience is neither structural nor tangible.
3) The expertise representation can be synergized with which algorithm/technique that realizes the self-aware capability? What are the possible levels of synergy?
   — *Answer:* The workflow structure needs to be synergized with both stimulus- and time-awareness at level 1; its synergy with goal-awareness is also required, but can be at any level except level 0. The past problem instances/experience needs to be synergized with goal-awareness only, at all levels, including level 0.
4) What is the possible form for each synergy?
   — *Answer:* The workflow structure can be synergized with goal awareness in either specific or general form. All other synergies need to be realized in a general form.
5) What is the difficulty level for each synergy?
   — *Answer:* According to Fig. 6, the difficulty level ranges between very easy to challenging.

The above answers produce 24 different candidates of synergizing domain expertise represented as the enriched temporal goal aware pattern.

4) Difficulty and Benefit Scores: For all the 24 candidates, their overall scores with respect to both the difficulty and benefit are shown in Fig. 23. Here, the w between specific and general form of synergy is set to 1.3 and 1.5, respectively. For each candidate, the proficiency is set to 1.8 for all synergies related to workflow structure and 1.3 for those related to past problem instances/experience.

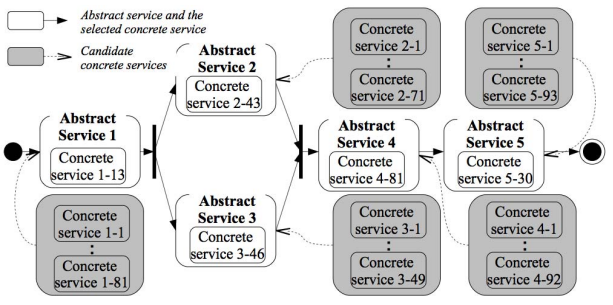


Fig. 22. ***Example model of workflow structure for a service composition.***

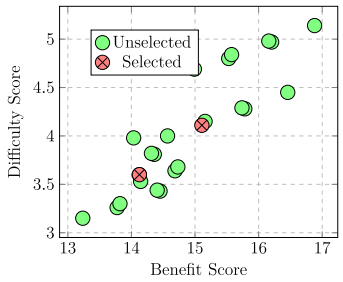


Fig. 23. ***Difficulty and benefit scores for all candidates in the third case study.***

5) Further Investigation: After discussion with the team, two candidates, as shown in Fig. 24(a), are selected since they appear to achieve a relatively good balance between the likely difficulty and the expected benefit. In brief, each of them is discussed as follows.

1) C_1 ($\mathbb{B}_{C_1} = 15.108$, $\mathbb{D}_{C_1} = 4.11$): The candidate in Fig. 24(a) automatically converts the knowledge of past problem instances and experience into structural and tangible representation. This is then used directly synergized with the goal-awareness to expedite the optimization process at level 2. The workflow model, which is a result of human reasoning, is directly utilized by the stimulus-, time- and goal-awareness at level 1 of synergy.

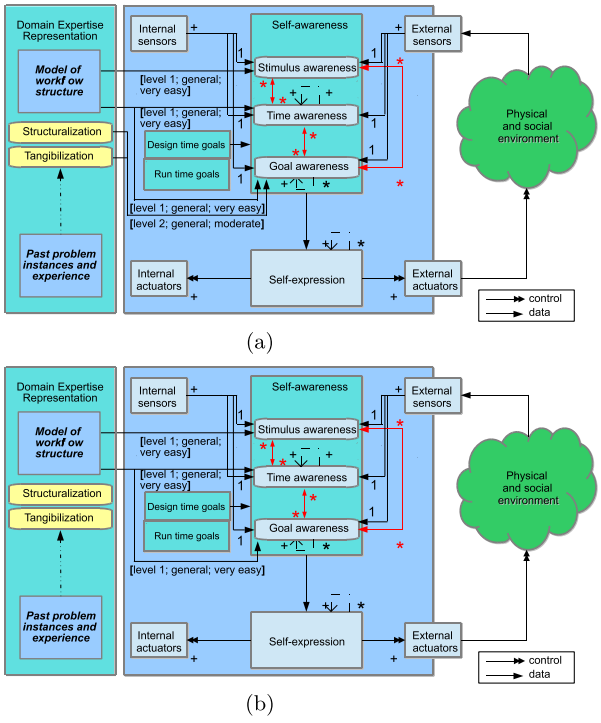


Fig. 24. ***Possible candidates selected for further investigation in the third case study. (a) Candidate C_1. (b) Candidate C_2.***

Table 14 Subject Service-Based Systems for the Experiments of the Third Case Study

System	Objective	#AS	#CS	Env.	Space
5AS	latency; throughput; cost	5	510	services	1.1×10^{10}
10AS	latency; throughput; cost	10	1,033	services	2.3×10^{20}
15AS	latency; throughput; cost	15	1,490	services	3.0×10^{30}
100AS	latency; throughput; cost	100	12,200	services	3.4×10^{200}

* AS denotes abstract services; CS denotes concrete services; Env. denotes environment; Space denotes search space.

2) C_2 ($\mathbb{B}_{C_2} = 14.125$, $\mathbb{D}_{C_2} = 3.6$): As shown in Fig. 24(b), the candidate has no synergy between the past problem instances/experience and goal-awareness. In other words, the evolutionary algorithm realizes goal-awareness, supported by the time aware analytical model, without any additional information on the past problem instances and experiences. All the other synergies remain the same as those of C_1.

More technical details on the actual synergy approaches can be found in [26].

6) Further Investigation Setup: The investigation is conducted by using the real-world WS-DREAM data set [90], which contains QoS values for 4500 services. Four distinct workflow structures of the software systems are randomly generated, each with 5, 10, 15, and 100 abstract services, respectively. As shown in Table 14, the number of concrete services and their QoS values on latency, throughput, and cost[9] are randomly selecting form the data set, resulting in a range between 510 and 12 200 possible concrete services with a search space over one million. NSGA-II [78] and IBEA [79] are used as the underlying evolutionary algorithm for goal-awareness, which are set a mutation rate of 0.1 and a crossover rate of 0.9, with 100 population size for 50 generation (300 generations for the case of 100 abstract services). As mentioned, for time-awareness, standard analytical models for service compositions are used [89]. All experiments were repeated 30 times and the mean values are reported. Again, the quality indicators used to assess both benefit and difficulty are shown in Table 15.

7) Results: From Table 16, clearly, C_1 leads to at least the same results for a quality objective when comparing to the case of C_2. In particular, it has also resulted in better Hypervolume (HV) value[10] [91]. All the

[9]The cost values are generated in a way that a concrete service with better latency would also have higher cost.

[10]HV measures the region from the nondominated solutions to a nadir point, which in this case is a vector of the worst possible objective values found. The larger the HV value, the better.

Table 15 Quality Indicators for Benefit and Difficulty for the Third Case Study

Attribute	Quality Indicators
Benefit	latency, throughput, cost and HV
Difficulty	Lines-Of-Code (LOC)

Table 16 Benefits Under Further Investigated Candidates Over All Runs

System	Metric	C_1	C_2
NSGA-II			
5AS	latency	**0.113**	**0.113**
	throughput	**0.048**	**0.048**
	cost	**7.151**	12.476
	hypervolume	**9.803E-01**	9.781E-01
10AS	latency	**0.113**	**0.113**
	throughput	**0.050**	0.048
	cost	**16.346**	24.832
	hypervolume	**9.871E-01**	9.705E-01
15AS	latency	**0.113**	**0.113**
	throughput	**0.065**	0.048
	cost	**43.433**	56.330
	hypervolume	**9.749E-01**	9.460E-01
100AS	latency	**0.113**	**0.113**
	throughput	**0.098**	0.047
	cost	**259.126**	404.584
	hypervolume	**9.860E-01**	9.521E-01
IBEA			
5AS	latency	0.276	**0.115**
	throughput	**0.047**	**0.047**
	cost	**7.151**	**7.151**
	hypervolume	**9.572E-01**	9.443E-01
10AS	latency	0.251	**0.132**
	throughput	**0.051**	0.050
	cost	16.148	**16.088**
	hypervolume	**9.669E-01**	9.541E-01
15AS	latency	0.170	**0.118**
	throughput	**0.065**	0.048
	cost	**45.304**	52.832
	hypervolume	**9.656E-01**	9.351E-01
100AS	latency	0.164	**0.115**
	throughput	**0.082**	0.053
	cost	**278.873**	447.491
	hypervolume	**9.755E-01**	9.435E-01

comparisons, except those equivalent ones, are statistically significant according to the Wilcoxon Signed Rank test (p <0.05), with nontrivial ESs. In particular, the improvement tends to be amplified as the number of abstract services increases, implying that the more complex the scenario, the better benefit that the domain expertise on past problem instances can offer when combined with the self-awareness. In Figs. 25 and 26, we see that on all cases, C_1 achieves higher HV value than that of the C_2 throughout, meaning that it exhibits faster convergence. Again, the improvement is more obvious under more complex scenarios, for example, when there are 100 abstract services.

In Fig. 27, the team examines how the behaviors of the software systems change when the underlying algorithm that realizes self-awareness is simplified. To this end, the crossover operator in NSGA-II is omitted, based on which the results can be compared to the cases when it is present for both approaches. Clearly, we see a considerable

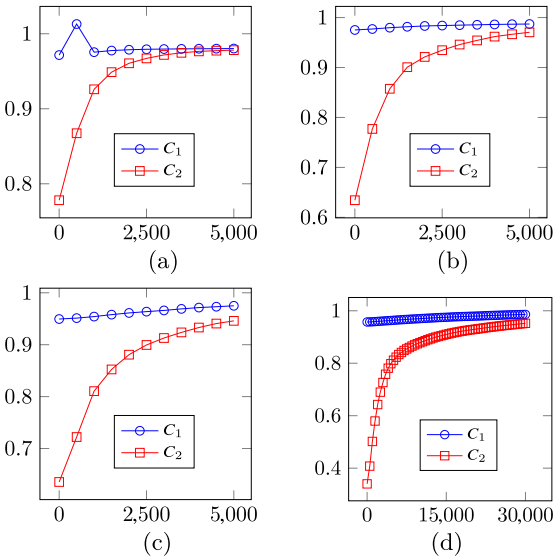


Fig. 25. *Changes of mean HV (y-axis) when using NSGA-II on all runs with respect to the number of evaluations (x-axis) under further investigated candidates. (a) 5AS. (b) 10AS. (c) 15AS. (d) 100AS.*

reduction on the HV values when the crossover operator is removed, suggesting that a simplified version of the underlying algorithm that realizes self-awareness may negatively affect the performance. Furthermore, the more complex the service system, the greater the reduction. However, we see that C_1 is more resilient than C_2, which again proves that the domain expertise of past problem instance can be beneficial in guiding the algorithm that achieves self-awareness for even better results.

As shown in Table 17, C_1 requires higher LOC than C_2, which is as anticipated. Yet, their margin of difficult

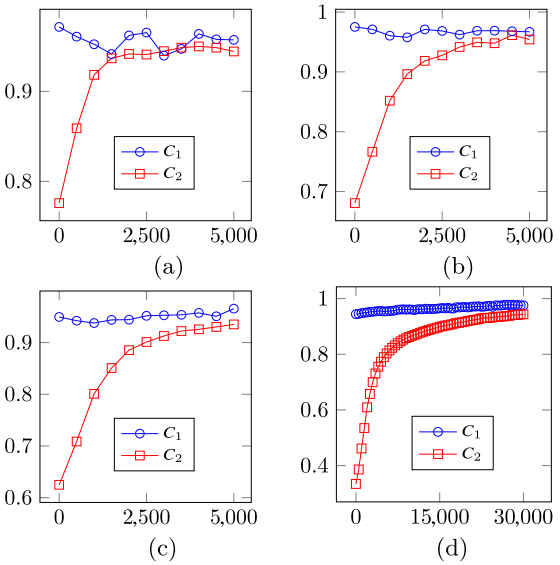


Fig. 26. *Changes of mean HV (y-axis) when using IBEA on all runs with respect to the number of evaluations (x-axis) under further investigated candidates. (a) 5AS. (b) 10AS. (c) 15AS. (d) 100AS.*

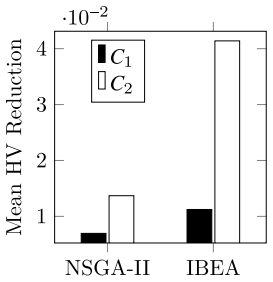


Fig. 27. ***Mean HV reduction (on all systems and runs) for simplified algorithm under further investigated candidates.***

is remarkably small (635 lines) and hence it is wiser to choose C_1 for the actual deployment.

Final choice for the third case study:
According to the verified results from the further investigation, the team has decided that C_1 can better fit the needs.

IX. DISCUSSION

A. How DBASES Can Help?

In summary, there are three aspects based on which DBASES can help to engineering synergy between domain expertise and self-awareness in a principled way, as follows:

1) Encapsulates various notations and classification that helps to understand, analyze, and reason about the information and knowledge that the engineers have for achieving self-awareness.
2) Visualizes the possible synergies, in a form of the enriched self-awareness architectural patterns, to provide intuitive understanding of the candidates.
3) Provides a methodology, building on the above two points, that offers step-by-step guidance on how to engineering self-awareness with explicit consideration of domain expertise.

It is worth noting that, although DBASES aims to help the engineers to finally select a single candidate of synergy, within the engineering process, it is by no mean that we restrict them to select only one or to choose all of the possible candidates. In this respect, DBASES is similar to CBAM [41], which is a successful architecture selection methodology that also helps to reveal and quantify the cost, benefit, and risk of design options in software development. CBAM also provides some visualizations similar to the way we do, but it is irrelevant to the explicit categories of domain expertise and their synergies to self-awareness. In fact, the precise description of which synergy candidate(s) chosen for implementation is irrelevant to the point of the framework and the message of this article. Our key point is that we have given the engineers a principled, repeatable method for making architectural choices of candidate(s), and understanding the consequences of these choices in terms of difficult and benefit, This method has been successful in that it guided the engineers to consider many ways of synergy that they would have otherwise overlooked, for two reasons.

1) Taking the difficult/benefit plot in Fig. 10 as an example, some candidates have extremely high benefits scores whilst relatively easy to realize and hence bear some of the highest desirability.
2) Some candidates have relatively low benefit scores but are still quite difficult to realize, primarily due to low proficiency. Therefore, they can be ruled out from consideration.

It is perfectly normal that more than one candidate are selected for further investigation and profiling, as what we have done in the tutorial case studies. But our framework provides such an opportunity to intuitively localize which are the prefer ones and which candidates should be ruled out, thereby saving the valuable human effort in investigating them.

B. Threats to Applicability

When the number of possible candidate increases, the engineers are likely overwhelmed with identifying and discussing them all. However, the fact that there are many combinations is not uncommon when making architectural design decisions [40]; this is in fact a more general problem in Operational Research that how can one makes proper tradeoff decision when there is a large number of alternatives. Visualization and quantification seem to be a promising solution, which is what DBASES provides. In this way, a designer can have more intuitive information on the relative difficulty and benefits on the alternative, and thus making informed decision.

Of course, when the number of points is too many to conduct analysis visually, it is possible to improve DBASES by incorporating some forms of preferences so that only a particular region of points that is of interest can be focused. This is, however, subject to future work.

C. Threats to External Validity

It is known that methodological work is extremely difficult to be evaluated and ensure its generality. The reported case studies, as the name suggested, aim to replicate what would have happen when DBASES is used in a diverse scenarios, in which case it is likely that only some desirable ones can be selected for further investigation while ruling out the others which are of no interests. This is the reason why we have chosen a subset of the candidates in the experiments. Of course, it is indeed possible to evaluate all of the synergy candidate, but this would consume a large

Table 17 LOC for the Second Case Study

Candidate	LOC
C_1	70,429
C_2	69,794

amount of time/resource, which we plan to investigate as part of future work.

Another threat is related to whether the industry practitioners will find that the DBASES is practical enough at a real-world industrial scale. Indeed, while this is important, it cannot be achieved without expensive surveying process, which will be extremely time-consuming. Therefore, we see this work as a first step to promote engineering synergy between domain expertise and self-awareness, and a more thorough evaluation with industrial stakeholders is part of our ongoing research.

X. CONCLUSION AND FUTURE WORK

Architectural patterns and methodology for self-awareness have proven to be effective in guiding the systematic design, knowledge representation, and reasoning for software systems that demand self-adaptation. However, when domain expertise needs to be synergized with the capabilities of self-awareness, current patterns, and methods lack guidelines about which domain expertise can be synergized, the extents of synergy and what are the tradeoffs involved.

This article is the first attempt that highlights the importance of synergizing domain expertise with the self-awareness in software systems, relying on well-defined underlying approaches. As part of the contributions, we present a holistic framework, dubbed DBASES, that offers a principled guideline for the engineers to perform difficulty and benefit analysis for synergizing domain expertise and self-awareness.

Using three tutorial case studies from distinct domains, we describe how DBASES can help to assist in making design decision on the synergy of domain expertise with self-aware capabilities, particularly on selecting candidates for further investigation with quantitative profiling.

The notion of synergy in DBASES is a genuine attempt toward keeping domain experts and architects in the loop, a branch of a larger vision that relate to keeping "engineers-in-the-loop" for self-adaptive software systems, in which human (i.e., software and system engineers for our case) can control the behaviors of the underlying algorithms and techniques that realize the self-awareness at least to certain extents. This will consequently offer greater intuition and transparency into the awareness processes of the self-adaptive software system, improving its interpretable and explainable appeal.

Drawing on the foundation provided in this article, future research shall investigate how exactly the human can be placed into the loop with DBASES, considering the timeliness and reliability of their expertise. Those problems will open up a full range of new research directions, drawing on the findings and proposals derived from this article. This is one of our ongoing research investigation that is evolving into a specialized topic by its own for the discipline of engineering self-aware and self-adaptive software systems. ■

REFERENCES

[1] U. Zdun and P. Avgeriou, "Modeling architectural patterns using architectural primitives," in *Proc. 20th Annu. ACM SIGPLAN Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, New York, NY, USA, 2005, pp. 133–146, doi: 10.1145/1094811.1094822.

[2] D. C. Schmidt and C. O'Ryan, "Patterns and performance of distributed real-time and embedded publisher/subscriber architectures," *J. Syst. Softw.*, vol. 66, no. 3, pp. 213–223, Jun. 2003.

[3] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, vol. 2. Hoboken, NJ, USA: Wiley, 2013.

[4] M. Stal, "Using architectural patterns and blueprints for service-oriented architecture," *IEEE Softw.*, vol. 23, no. 2, pp. 54–61, Mar. 2006.

[5] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, and T. Newling, *Patterns: Service-Oriented Architecture and Web Services*. New York, NY, USA: IBM Corporation, 2004.

[6] A. Computing *et al.*, *An Architectural Blueprint for Autonomic Computing*, vol. 31. New York, NY, USA: IBM, 2006, pp. 1–6.

[7] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, no. 10, pp. 46–54, Oct. 2004.

[8] T. Chen *et al.*, "The handbook of engineering self-aware and self-expressive systems," 2014, *arXiv:1409.1793*. [Online]. Available: http://arxiv.org/abs/1409.1793

[9] H. Giese, T. Vogel, A. Diaconescu, S. Götz, and S. Kounev, *Architectural Concepts for Self-Aware Computing Systems*. Cham, Switzerland: Springer, 2017, pp. 109–147, doi: 10.1007/978-3-319-47474-8_5.

[10] P. R. Lewis *et al.*, "Architectural aspects of self-aware and self-expressive computing systems: From psychology to engineering," *Computer*, vol. 48, no. 8, pp. 62–70, Aug. 2015.

[11] T. Chen, F. Faniyi, and R. Bahsoon, "Design patterns and primitives: Introduction of components and patterns for SACS," in *Self-Aware Computing Systems An Engineering Approach* (Natural Computing Series), P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, Eds. Springer, 2016, pp. 53–78, doi: 10.1007/978-3-319-39675-0_5.

[12] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting self-aware software systems," in *Proc. IEEE/IFIP Conf. Softw. Archit.*, Apr. 2014, pp. 91–94.

[13] T. Chen and R. Bahsoon, "Self-adaptive and online QoS modeling for cloud-based software services," *IEEE Trans. Softw. Eng.*, vol. 43, no. 5, pp. 453–475, May 2017.

[14] T. Chen and R. Bahsoon, "Self-adaptive trade-off decision making for autoscaling cloud-based services," *IEEE Trans. Services Comput.*, vol. 10, no. 4, pp. 618–632, Jul. 2017.

[15] T. Chen and R. Bahsoon, "Toward a smarter cloud: Self-aware autoscaling of cloud configurations and resources," *Computer*, vol. 48, no. 9, pp. 93–96, Sep. 2015.

[16] N. Dutt, A. Jantsch, and S. Sarma, "Self-aware cyber-physical systems-on-chip," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2015, pp. 46–50.

[17] Z. Guettatfi, P. Hubner, M. Platzner, and B. Rinner, "Computational self-awareness as design approach for visual sensor nodes," in *Proc. 12th Int. Symp. Reconfigurable Commun.-Centric Syst. Chip (ReCoSoC)*, Jul. 2017, pp. 1–8.

[18] B. Rinner *et al.*, "Self-aware and self-expressive camera networks," *Computer*, vol. 48, no. 7, pp. 21–28, Jul. 2015.

[19] D. Binkley, D. Lawrie, and C. Morrell, "The need for software specific natural language techniques," *Empirical Softw. Eng.*, vol. 23, no. 4, pp. 2398–2425, Aug. 2018.

[20] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Inf. Softw. Technol.*, vol. 76, pp. 135–146, Jun. 2016, doi: 10.1016/j.infsof.2016.04.017.

[21] T. Menzies, "The five laws of SE for AI," *IEEE Softw.*, vol. 37, no. 1, pp. 81–85, Jan. 2020.

[22] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms," in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, San Francisco, CA, USA, May 2013, pp. 522–531, doi: 10.1109/ICSE.2013.6606598.

[23] T. Chen, K. Li, R. Bahsoon, and X. Yao, "FEMOSAA: Feature-guided and knee-driven multi-objective optimization for self-adaptive software," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, pp. 1–50, Jul. 2018.

[24] T. Chen, R. Bahsoon, S. Wang, and X. Yao, "To adapt or not to adapt?: Technical debt and learning driven self-adaptation for managing runtime performance," in *Proc. ACM/SPEC Int. Conf. Perform. Eng. (ICPE)*, New York, NY, USA, 2018, pp. 48–55, doi: 10.1145/3184407.3184413.

[25] T. Chen, M. Li, and X. Yao, "On the effects of seeding strategies: A case for search-based multi-objective service composition," in *Proc. Genetic Evol. Comput. Conf. (GECCO)*, New York, NY, USA, 2018, pp. 1419–1426, doi: 10.1145/3205455.3205513.

[26] T. Chen, M. Li, and X. Yao, "Standing on the shoulders of giants: Seeding search-based multi-objective optimization with prior knowledge for software service composition," *Inf. Softw. Technol.*, vol. 114, pp. 155–175, Oct. 2019, doi: 10.1016/j.infsof.2019.05.013.

[27] T. Chen, R. Bahsoon, and X. Yao, "A survey and taxonomy of self-aware and self-adaptive cloud autoscaling systems," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–40, Jul. 2018.

[28] A. Elhabbash, M. Salama, R. Bahsoon, and P. Tino, "Self-awareness in software engineering:

A systematic literature review," *ACM Trans. Autonom. Adapt. Syst.*, vol. 14, no. 2, pp. 1–42, Dec. 2019.

[29] T. Chen, "All versus one: An empirical comparison on retrained and incremental machine learning for modeling performance of adaptable software," in *Proc. IEEE/ACM 14th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, M. Litoiu, S. Clarke, and K. Tei, Eds, Montreal, QC, Canada, May 2019, pp. 157–168.

[30] T. Chen, M. Li, and X. Yao, "How to evaluate solutions in Pareto-based search-based software engineering? A critical review and methodological guidance," *CoRR*, vol. abs/2002.09040, Jun. 2020.

[31] K. Li, Z. Xiang, T. Chen, S. Wang, and K. C. Tan, "Understanding the automated parameter optimization on transfer learning for CPDP: An empirical study," in *Proc. 42nd Int. Conf. Softw. Eng. (ICSE)*, Seoul South Korea, May 2020.

[32] A. Agrawal and T. Menzies, "Is AI different for SE?" *CoRR*, vol. abs/1912.04061, Jun. 2019.

[33] H. Wada, J. Suzuki, Y. Yamano, and K. Oba, "E^3: A multiobjective optimization framework for sla-aware service composition," *IEEE Trans. Services Comput.*, vol. 5, no. 3, pp. 358–372, Jun. 2012.

[34] S. Kumar, R. Bahsoon, T. Chen, K. Li, and R. Buyya, "Multi-tenant cloud service composition using evolutionary optimization," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Singapore, Dec. 2018, pp. 972–979.

[35] T. Chen and R. Bahsoon, "Symbiotic and sensitivity-aware architecture for globally-optimal benefit in self-adaptive cloud," in *Proc. 9th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, G. Engels and N. Bencomo, Eds., Hyderabad, India, Jun. 2014, 2014, pp. 85–94.

[36] T. Chen, M. Li, K. Li, and K. Deb, "Search-based software engineering for self-adaptive systems: One survey, five disappointments and six opportunities," *CoRR*, vol. abs/2001.08236, Aug. 2020.

[37] G. G. Pascual, R. E. Lopez-Herrejon, M. Pinto, L. Fuentes, and A. Egyed, "Applying multiobjective evolutionary algorithms to dynamic software product lines for reconfiguring mobile applications," *J. Syst. Softw.*, vol. 103, pp. 392–411, May 2015, doi: 10.1016/j.jss.2014.12.041.

[38] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016.

[39] P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, *Self-Aware Computing Systems: An Engineering Approach* (Natural Computing). New York, NY, USA: Springer, 2016, doi: 10.1007/978-3-319-39675-0.

[40] C. Paul, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA, USA: Addison-Wesley, 2002.

[41] R. Kazman, J. Asundi, and M. Klein, "Quantifying the costs and benefits of architectural decisions," in *Proc. 23rd Int. Conf. Softw. Eng. (ICSE)*, Toronto, OH, Canada, 2001, pp. 297–306.

[42] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992.

[43] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. Hoboken, NJ, USA: Wiley, 2004.

[44] P. Oreizy *et al.*, "An architecture-based approach to self-adaptive software," *IEEE Intell. Syst.*, vol. 14, no. 3, pp. 54–62, May 1999.

[45] D. Weyns *et al.*, "On patterns for decentralized control in self-adaptive systems," in *Software Engineering for Self-Adaptive Systems*. New York, NY, USA: Springer, 2013, pp. 76–107.

[46] E. Gat, R. P. Bonnasso, and R. Murphy, "On three-layer architectures," *Artif. Intell. Mobile Robots*, vol. 195, p. 210, Dec. 1998.

[47] J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *Proc. Future Softw. Eng. (FOSE)*, Washington, DC, USA, May 2007, pp. 259–268, doi: 10.1109/FOSE.2007.19.

[48] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A general and extensible framework for self-aware computing," MIT, Cambridge, MA, USA, Tech. Rep. MIT-CSAIL-TR-2011-046, 2011.

[49] T. Becker *et al.*, "EPiCS: Engineering proprioception in computing systems," in *Proc. IEEE 15th Int. Conf. Comput. Sci. Eng.*, Dec. 2012, pp. 353–360.

[50] F. Buschmann, K. Henney, and D. Schmidt, *Pattern Oriented Software Architecture: On Patterns and Pattern Languages*. Hoboken, NJ, USA: Wiley, 2007.

[51] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. I. Verkamo, "Software metrics by architectural pattern mining," in *Proc. Int. Conf. Softw., Theory Pract.*, 2000, pp. 325–332.

[52] P. Kruchten, *The Rational Unified Process: An Introduction*. Reading, MA, USA: Addison-Wesley, 2004.

[53] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, 2002.

[54] C. M. Ashworth, "Structured systems analysis and design method (SSADM)," *Inf. Softw. Technol.*, vol. 30, no. 3, pp. 153–163, 1988.

[55] K. Schwaber, "Scrum development process," in *Business object Design Implemention*. New York, NY, USA: Springer, 1997, pp. 117–134.

[56] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, Apr. 1993, doi: 10.1145/157710.157715.

[57] M. Tufano *et al.*, "When and why your code starts to smell bad," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, May 2015, pp. 403–414.

[58] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Redwood City, CA, USA: Addison-Wesley, 2004.

[59] J. M. Sullivan, "Impediments to and incentives for automation in the air force," in *Proc. Int. Symp. Technol. Soc. Weapons Wires, Prevention Saf. Time Fear*, 2005, pp. 102–110.

[60] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Softw. Eng. Inst., Carnegie Mellon Univ., Pittsburgh, PA, Tech. Rep. CMU/SEI-90-TR-021, 1990. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=11231

[61] L. Liu and E. Yu, "Designing information systems in social context: A goal and scenario modelling approach," *Inf. Syst.*, vol. 29, no. 2, pp. 187–203, Apr. 2004.

[62] G. Booch, *The Unified Modeling Language User Guide*. London, U.K.: Pearson, 2005.

[63] S. Fine, Y. Singer, and N. Tishby, "The hierarchical hidden Markov model: Analysis and applications," *Mach. Learn.*, vol. 32, no. 1, pp. 41–62, 1998.

[64] W. Reisig, "Petri nets and algebraic specifications," in *High-Level Petri Nets*. New York, NY, USA: Springer, 1991, pp. 137–170.

[65] D. G. Kendall, "Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded Markov chain," *Ann. Math. Stat.*, vol. 24, no. 3, pp. 338–354, Sep. 1953.

[66] E. S. Yu, "Social modeling and i*," in *Conceptual Modeling: Foundations and Applications: Essays in Honor of John Mylopoulos*, A. T. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. Yu, Eds. Berlin, Germany: Springer, 2009, pp. 99–121, doi: 10.1007/978-3-642-02463-4_7.

[67] D. T. Ross and K. E. Schoman, "Structured analysis for requirements definition," *IEEE Trans. Softw. Eng.*, vols. SE–3, no. 1, pp. 6–15, Jan. 1977.

[68] K. Beck, "Using pattern languages for object-oriented programs," in *Proc. OOPSLA*, 1987.

[69] A. Keller and H. Ludwig, "The WSLA framework: Specifying and monitoring service level agreements for Web services," *J. Netw. Syst. Manage.*, vol. 11, no. 1, pp. 57–81, 2003.

[70] K. Wiegers and J. Beatty, *Software Requirements*. London, U.K.: Pearson, 2013.

[71] J. H. Bradley, R. Paul, and E. Seeman, "Analyzing the structure of expert knowledge," *Inf. Manage.*, vol. 43, no. 1, pp. 77–91, Jan. 2006.

[72] P. N. Robillard, "The role of knowledge in software development," *Commun. ACM*, vol. 42, no. 1, pp. 87–92, Jan. 1999.

[73] P. B. Andersen and P. Nowack, "Tangible objects: Connecting informational and physical space," in *Virtual Space*. New York, NY, USA: Springer, 2002, pp. 190–210.

[74] B. H. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.* New York, NY, USA: Springer, 2009, pp. 468–483.

[75] D. M. Berry, B. H. Cheng, and J. Zhang, "The four levels of requirements engineering for and in dynamic adaptive systems," in *Proc. 11th Int. Workshop Requirement Eng. Found. Softw. Qual. (REFSQ)*, 2005, p. 5.

[76] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration using feature models," in *SPLC*, vol. 3154. New York, NY, USA: Springer, 2004, pp. 266–283.

[77] *Rice University Bidding Systems*, Rice Univ., Houston, TX, USA. Accessed: Nov. 17, 2019. [Online]. Available: http://rubis.ow2.org/

[78] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[79] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *Proc. 8th Int. Conf. Parallel Problem Solving Nature*, 2004, pp. 832–842.

[80] T. Chen and R. Bahsoon, "Self-adaptive and sensitivity-aware QoS modeling for the cloud," in *Proc. 8th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, May 2013, pp. 43–52.

[81] T. Chen, R. Bahsoon, and X. Yao, "Online QoS modeling in the cloud: A hybrid and adaptive multi-learners approach," in *Proc. IEEE/ACM 7th Int. Conf. Utility Cloud Comput.*, Dec. 2014, pp. 327–336.

[82] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of effect size in software engineering experiments," *Inf. Softw. Technol.*, vol. 49, nos. 11–12, pp. 1073–1086, Nov. 2007.

[83] J. Skene, F. Raimondi, and W. Emmerich, "Service-level agreements for electronic services," *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 288–304, Mar. 2010.

[84] A. Andrieux *et al.*, "Web services agreement specification (WS-agreement)," *Open Grid Forum*, vol. 128, no. 1, p. 216, 2007.

[85] G. H. John and P. Langley, "Estimating continuous distributions in Bayesian classifiers," in *Proc. 11th Conf. Uncertainty Artif. Intell.*, San Francisco, CA, USA: Morgan Kaufmann, 1995, pp. 338–345. [Online]. Available: http://dl.acm.org/citation.cfm?id=2074158.2074196

[86] S. S. Haykin, *Neural Networks: A Comprehensive Foundation*. Beijing, China: Tsinghua Univ. Press, 2001.

[87] S. Kumar, R. Bahsoon, T. Chen, and R. Buyya, "Identifying and estimating technical debt for service composition in SaaS cloud," in *Proc. IEEE Int. Conf. Web Services (ICWS)*, E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, and K. Oyama, Eds., Milan, Italy, Jul. 2019, pp. 121–125.

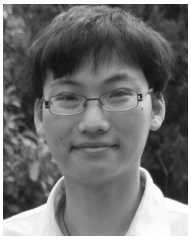
[88] S. Kumar, T. Chen, R. Bahsoon, and R. Buyya, "Datesso: Self-adapting service composition with debt-aware two levels constraint reasoning," in *Proc. 15th Int. Symp. Softw. Eng. Adapt. Self-Manag. Syst. (SEAMS)*, Seoul South Korea, 2020.

[89] A. Ramírez, J. A. Parejo, J. R. Romero, S. Segura, and A. Ruiz-Cortés, "Evolutionary composition of QoS-aware Web services: A many-objective perspective," *Expert Syst. Appl.*, vol. 72, pp. 357–370, Apr. 2017.

[90] Z. Zheng, Y. Zhang, and M. R. Lyu, "Investigating QoS of real-world Web services," *IEEE Trans. Services Comput.*, vol. 7, no. 1, pp. 32–39, Jan. 2014.

[91] M. Li, T. Chen, and X. Yao, "A critical review of: 'A practical guide to select quality indicators for assessing Pareto-based search algorithms in search-based software engineering': Essay on quality indicator selection for SBSE," in *Proc. 40th Int. Conf. Softw. Eng., New Ideas Emerg. Results, (NIER)*, A. Zisman and S. Apel, Eds., Gothenburg, Sweden, May/Jun. 2018, pp. 17–20.

ABOUT THE AUTHORS



Tao Chen (Member, IEEE) received the Ph.D. degree from the School of Computer Science, University of Birmingham, Birmingham, U.K., in 2016.

He is currently a Lecturer (an Assistant Professor) in computer science with the Department of Computer Science, Loughborough University, Loughborough, U.K. He has broad research interests on software engineering, including but not limited to performance engineering, self-adaptive software systems, search-based software engineering, data-driven software engineering, and computational intelligence. As the lead author, his work has been published in internationally renowned journals and conference such as the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, the *ACM Transactions on Software Engineering and Methodology*, the IEEE TRANSACTIONS ON SERVICES COMPUTING, the *ACM Computing Surveys*, and the IEEE/ACM International Conference on Software Engineering.

Dr. Chen regularly serves as a program committee (PC) Member for various conferences in his fields. He also serves as an Associate Editor for the *Services Transactions on Internet of Things*.

Rami Bahsoon received the Ph.D. degree in software engineering from the University College London, London, in 2006, investigating stability of software architectures. He was an MBA Fellow in Technology at London Business School (2003–2005).

During his sabbatical year in 2018, he was a Visiting Scientist with the Software Engineering Institute (SEI), Carnegie Mellon University, Pittsburgh, PA, USA, investigating the problem of runtime evaluation of IoT software architectures and was the 2018 Melbourne School of Engineering (MSE) Visiting Fellow of The School of Computing and Information Systems, the University of Melbourne, Melbourne, VIC, Australia, investigating technical debt management in cloud-based architectures. He is currently a Senior Lecturer (an Associate Professor) with the School of Computer Science, University of Birmingham, Birmingham, U.K. He is also a Founding Member of the Software Engineering Research Group, Birmingham University. He co-edited four books on software architecture, including *Economics-Driven Software Architecture*, *Software Architecture for Big Data and the Cloud*, *Aligning Enterprise*, *System*, and *Software Architecture*. He conducts research in the fundamentals of self-adaptive and managed software architectures and its application to emerging paradigms, such as cloud, microservices, IoT, and CPS. His investigations have also looked at self-aware software architectures, economics-driven software architectures, and technical debt management in services and cloud software engineering.

Dr. Bahsoon is a Fellow of the Royal Society of Arts. He is also an Associate Editor of *IEEE Software*.

Xin Yao (Fellow, IEEE) received the B.Sc. degree from the University of Science and Technology of China (USTC), Hefei, China, in 1982, the M.Sc. degree from the North China Institute of Computing Technologies, Beijing, China, in 1985, and the Ph.D. degree from USTC in 1990.

He is currently a Chair Professor of computer science with the Southern University of Science and Technology (SUSTech), Shenzhen, China, and a part-time Professor of computer science with the University of Birmingham, Birmingham, U.K. His current research interests include evolutionary computation, machine learning, and their real-world applications, especially to software engineering.

Dr. Yao was a recipient of the prestigious Royal Society Wolfson Research Merit Award in 2012, the IEEE Computational Intelligence Society (CIS) Evolutionary Computation Pioneer Award in 2013, and the IEEE Frank Rosenblatt Award in 2020. His work won the 2001 IEEE Donald G. Fink Prize Paper Award, the 2010, 2016, and 2017 IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION Outstanding Paper Awards, the 2011 IEEE TRANSACTIONS ON NEURAL NETWORKS Outstanding Paper Award, and many other best paper awards. He was the President of IEEE CIS from 2014 to 2015 and the Editor-in-Chief of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION from 2003 to 2008. He was a Distinguished Lecturer of IEEE CIS.