A New Software Architecture for the Wise Object Framework: Multidimensional Separation of Concerns

Sylvain Lejamble[1,2][a], Ilham Alloui[2][b], Sébastien Monnet[2][c] and Flavien Vernier[2][d]
[1]*SaGa Corp, Paris, France*
[2]*LISTIC, Université Savoie Mont Blanc, Annecy, France*

Keywords: Self-adaptive Systems, Separation of Concerns, Modularization, Wise Object, Event-driven Architecture.

Abstract: Adaptive systems represent an appropriate solution to the increasing complexity of software-intensive systems. We constructed a Wise Object Framework to develop self-adaptive software systems we name "Wise systems". Those consist of distributed communicating software objects (Wise Objects) able to autonomously learn on how they behave and how they are used while demanding little attention from their users. A WO is either delivering a service (Awake state) or simulating its operation to learn behavior that has not occurred yet (Dream state). In its first version, WO architecture has been designed on the basis of a single component embedding built-in mechanisms for data monitoring and analysis. This architecture has major drawbacks we encountered when using WOF to develop new applications: (*1*) built-in mechanisms embedded within the WO do not allow using data by other components such as new analyzers, (*2*) raw data and data resulting from analysis in Awake or Dream states are not distinguished, (*3*) mandatory components to self-adaptation are missing especially those for action plan construction and execution. In this paper, we address those limitations through a MAPE-K compliant architecture, based on the Separation of Concerns(SoC) and an event-driven publish/subscribe mechanism. This is related to the general issue of wise system maintenance, reuse and evolution. Separation of Concerns is done according to different dimensions that are managed using hyperslicing techniques.

1 INTRODUCTION

Adaptive systems represent an appropriate solution to the increasing complexity of software-intensive systems. An ideal system is able to adapt to users by reducing the effort required for human-machine interaction. It must also be able to detect unusual situations (error, unexpected behavior...) and manage them (correct, alert a suitable person to its resolution...). Self-adaptive systems are an open area of research that address many of these issues with the goal to let IT professionals focus on business higher value-added tasks. For this purpose, we developed WOF, a Wise Object Framework to build self-adaptive software systems we name "Wise systems". A Wise system consists of a set of distributed communicating software objects (Wise Objects: WOs), each able to autonomously learn on how it behaves and how it is used while demanding little attention from its users. WOs are able to monitor their functioning in order to acquire experience on their operation and hence on the use of the system ("Real logs"). Logs analysis guides the system towards an action plan (self-correction, adaptation, optimization...). Each time a WO is requested for service delivery, it collects data on this operation using a monitoring built-in mechanism. When a WO is not delivering a service, it can disconnect from the rest of the system to learn on its behavior by launching operations and analyzing their effects and impact. We respectively refer to those super-states as "Awake state" and "Dream state". The WO ability to disconnect from the real world is a strong feature that distinguishes wise systems from other self-adaptive systems. WOs may execute some parts of the system without impacting the real operation (Business domain). These "simulations" have several goals: (*i*) learn more about rare (fewly used) operations on the system, (*ii*) create state-transition behavioral graphs of the system, and (*iii*) create credible/realistic data to help analyzing real functioning of the system. During this state, the system produces logs different from real

[a] https://orcid.org/0000-0003-1043-4745
[b] https://orcid.org/0000-0002-3713-0592
[c] https://orcid.org/0000-0002-6036-3060
[d] https://orcid.org/0000-0001-7684-6502

Lejamble, S., Alloui, I., Monnet, S. and Vernier, F.
A New Software Architecture for the Wise Object Framework: Multidimensional Separation of Concerns.
DOI: 10.5220/0011355000003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 567-574
ISBN: 978-989-758-588-3; ISSN: 2184-2833

ones, we call them "Dream logs". The existing WO architecture in the WOF (Alloui et al., 2018) does not meet the requirements regarding the adaptation of the framework to complex applications. Indeed, in its first version, WO architecture has been designed on the basis of a single component embedding built-in mechanisms for data monitoring and analysis. This architecture has major drawbacks we encountered when using WOF to develop new applications: (1) built-in mechanisms embedded within the WO do not allow using data by other components such as new analyzers, (2) raw data and data resulting from the analysis be them from Awake or Dream states are not distinguished, (3) mandatory components to self-adaptation are missing especially those for plan construction and execution. In this paper, we address those limitations through a MAPE-K compliant architecture, based on the Separation of Concerns and an event-driven publish/subscribe mechanism. This is related to the general issue of wise system maintenance, reuse and evolution. Separation of Concerns is done according to different dimensions that are managed using hyperslicing techniques. The knowledge acquisition system is refactored to separate the different components involved in the self-adaptation MAPE-K loop.

The paper is organized as follows. Section 2 introduces related work. Section 3 presents the previous work on the framework. Section 4 describes the limits of the current architecture. Section 5 explains the solution we propose to handle present limitations. Finally, Section 6 concludes the paper and opens some interesting perspectives.

2 RELATED WORK

2.1 Self-Adaptive Systems

To meet the growing demand for more complex problems, IT professionals are striving to build even more complex systems. These high-value-added systems are very efficient but are becoming increasingly difficult and expensive to maintain (Crow, 1990). Furthermore, the maintenance must be performed by professionals whose expertise is close to the system.

An adaptive system can have several goals: adapting in response to its changing environment (Brun et al., 2009), switching models in a multi-model system (Ravindranathan and Leitch, 1998), modifying its components without waiting for the next maintenance periods to limit human interactions (Naqvi, 2012) or even evaluating its performance and changing its strategy when it is not performing enough (Cheng et al., 2009).

Self-adaptive systems are an open area of research that can address many of these issues. If the system can solve a problem itself, then IT professionals can focus on higher value-added tasks for the business. Although a system is highly dependent on its application domain, all self-adaptive systems have similar components. These blocks are defined by IBM's 4-state loop "Monitor-Analyze-Plan-Execute" (MAPE-K) (Kephart and Chess, 2003). This loop has a fifth component named "Knowledge" that communicates with the other 4.

Monitor: The system must produce comprehensive operation logs and make them available to the other components. **Analyze**: The system must then analyze its past operations in the manner of an expert and draw a conclusion: is the system functioning appropriately/as usual? **Plan**: The system uses the analysis results to produce a strategy addressing the problem/unusual situation. **Execute**: The execution block applies this plan to make the system perform a corrective action/adapt to a changing usage. **Knowledge**: Knowledge is the link between all the software blocks. It stores what the system knows about itself: logs, analyzes, metrics, topological information, actions performed...

2.2 Separation of Concerns

Separation of Concerns (SoC) (Dijkstra and Edsger, 1982) becomes essential while developing very complex software. In a system, each element should have an exclusive purpose. Ideally, while doing the Separation of Concerns, no software element should share responsibilities with another part of the system. We must set boundaries that encompass the full range of responsibilities along dimensions. They are a lot of types of Separations of Concerns[1]: (*i*) horizontal separation (separation through functionalities), (*ii*) vertical separation (dividing the application into modules), (*iii*) aspect separation...

MAPE-K provides the ability, by defining distinct blueprints, to separate priorities and thus to accelerate and simplify software development (Parnas, 1972). The multidimensional separation of priorities is interesting when a system may have to deal with new concerns (Ossher and Tarr, 2002). It may also be necessary when several concerns overlap just a small part of another. The separation can have a huge impact on maintenance when one part of the software has many diverse dependencies (Moreira et al., 2005). (Tarr et al., 1999) highlighted the multidimensional Separation of Concerns while the literature

[1] http://aspiringcraftsman.com/2008/01/03/art-of-separation-of-concerns/

was focusing on an orthogonal separation. They believe that the future of software development requires a "simultaneous separation of overlapping concerns in multiple dimensions". They also show that their work was already partially done in Subject-Oriented Programming (Harrison et al., 1993)(subjects are hyperslices), Aspect-Oriented Programming (Kiczales et al., 1997)(aspects are hyperslices), Adaptive Programming (Gouda et al., 1991)(propagation patterns are hyperslices)... Aspect Oriented Programming (AOP) may also help a lot during the Separation of Concerns process because it highlights non-functional concerns that might be separated. (Makabee, 2012) use an Event-driven approach to spot entangled concerns and separate them from other functionalities. He also compares Event-driven and Aspect-Oriented approaches and show that Event-driven programming has several advantages over AOP (more reusable, allowing inheritance and concurrent execution...).

3 PREVIOUS ARCHITECTURE

Wise Object. A WO (Alloui et al., 2015) is a software component that is given the ability to learn by itself from its past experiences. It can also learn about its environment through the interactions it has with it.

The WO aims at proxifying an existing object (physical or logical) from the system to add capabilities (wisdom) to it (Figure 1). In the first version of the concept, the proxy was implemented in an intrusive way in the business classes. The object also had to inherit from the WiseObject class in order to expose its monitoring functions. This caused several problems, especially in the inheritance chain. To fix them, we imagined a dynamic proxy that lets process the data upstream. It avoids unnecessary intrusions in the business code. The WO will therefore intercept all the system requests related to the object and create operation logs.

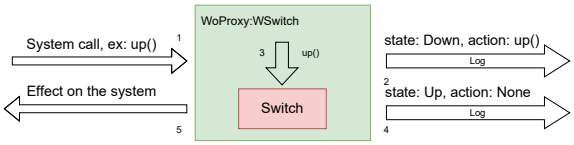

Figure 1: Example of a wise version of a switch. The switch has 2 states Up/Down and 2 functions up()/down() changing the business state of the object. WO Proxy intercepts calls (up() or down()) (1) and monitors them (2). It asks the object to act (3). As the state changes, it monitors it again (4). It sends back the response according to the function call (5).

Wise Object Framework. A framework has been developed around the notion of WO (Alloui et al., 2018). This framework opens up the possibility of using the WO concept in a classical computer application. It allows the use of a simple @WiseObject annotation in the declaration of the target object to monitor all its status changes. By doing so, a proxy is auto generated wrapping the original object. It also connects the proxy (logger) to the knowledge database. This knowledge will be progressively filled in by the usage logs. The proxy grants complete control over the original object. Since it intercepts the calls, it can modify or cancel a call to avoid an unwanted behavior if the analysis detects the system call is unusual and will cause the object to fall into an unwanted state.

Within the framework, WOs are able to communicate with each other through a software bus (publish/subscribe). The communication is done through event condition action rules (ECA) established by a manager. By detaching components from this software bus, it is possible to work with small parts of the application without impacting the system.

Detaching WO. The WO has two high-level states: *Awake*/*Dream*. During the *Awake* state, the object acts normally, and serves the system. When the object is not busy with a system call, it enters the *Dream* state (Figure 2).

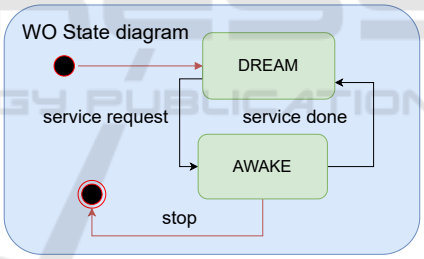

Figure 2: Original WO high-level states.

In the *Dream* state, we disconnect the WO from the bus in order to jump to a non-impacting phase. In order to guarantee this non-impact phase, the application must catch every instruction that leads to the writing of a file. The reading remains however functional. In this state the object will simulate system calls to explore its state transition graph. The goal is to discover as many as possible business states that have not yet been discovered during the actual operation of the system. For example, if our switch has never been turned on, it is theoretically impossible to know that the object has two states by analyzing the logs. As the object knows itself, it knows that it has several methods. So during the *Dream* it will randomly try to use up() and down() from its Down state. When using down(), the state keeps being Down, so an self-transition is created in the state diagram. However,

when using up(), the object discovers that it can enter a different state if it applies the procedure it just discovered. The use of *Dream* state improves the overall system perspectives. By analyzing produced logs, we are able to enrich the knowledge we have about the system. For example, we can confront the theoretical model of the system (designed initially by an expert) with the state transition graph discovered in the *Dream* state.

The framework makes possible the usage of interchangeable analyzers. These analyzers will scan the object's logs and draw a conclusion based on their expertise. For example, with a statistical analyzer, we can compute temporal metrics on the functioning of the object. By interpreting the results, it is possible to detect anomalies (e.g.: switch unusually *Up* on 16/03 from 6pm to 7am) and then warn the user.

Application. Extensive research (Alloui et al., 2018) has been conducted on the use of the WOF for IoT. The results proved the relevance of WOs in real world systems such as in classroom unusual use detection.

4 CONSTRAINTS & LIMITS

The precedent framework architecture has several drawbacks that led us to propose some changes.

WOs embedded Too Much Responsibilities. In the precedent architecture (Alloui et al., 2018), the knowledge is represented as a graph that is created during the monitoring process. Furthermore that knowledge was part of the WO. It shows that knowledge, its acquisition (monitoring) and the WO were mixed. The graph was also already an analysis since the logs are linked together and interpreted (no duplication of states in the graph...). We want the raw knowledge to be available in the memory so we can process it afterward. It is the main motivation for the horizontal Separation of Concerns.

Incoherent Dreams & Tangled Logs. In the previous work, the discovery of the unseen business states (in the dream) was done in a random way. We called state-change methods randomly from different initial states to see their impact on the states and we hoped at the end of the *Dream* that all possible business states and the transitions between them would be discovered. Since we only had a few possible alterations, the system never ends up in an incoherent state. If we make the system dream with parameterized methods in a random way(random function with random parameter), we risk finding impossible values when using the system in a business context. Furthermore, state change methods were hard-coded with restricting boundaries (we can call up when the object is at 100%, but it has no effect). Now let us consider OpenWindow(normal: from 0 to 100%) as an example. As humans we know it is absurd to use it with OpenWindow(-60%), except that the WO has no semantic and it is not intelligent, it does not know that -60% is not possible in the business context. The problem is therefore the following: we want to discover as many business states of the system as possible and at the same time we want these findings to be relevant. To overcome this problem, we have chosen to analyze the parameters before asking the Wise Object to use them (during *Dream* state). This allows the system to get closer to the human, i.e. the human dreams about his past experience and not randomly. In order to analyze these parameters, we need to be able to distinguish at the software level what really happened (real logs) from what we asked the system to do to discover its states (dream logs). This is not possible with the current architecture. This modification is the primary motivation for the Separation of Concerns in the WOF. By externalizing some parts of the application the Separation of Concerns grants a better adaptability and re-usability. A multidimensional separation seems to be necessary at the analysis level as we would like to stack non-vital part of the application. Parameter analysis will not be detailed in this paper but will surely be the subject of a more detailed study.

Missing MAPE Components. Finally, the WO had multiple roles but mandatory components of the MAPE loop as planning and execution were missing. It has been a huge problem while adapting to the framework to other business domains. To address this issue, we propose a basic architecture enabling the loop to be self-regulating.

5 CONTRIBUTION

We propose an upgrade of the current WOF architecture as well as an extension including the last elements of the MAPE loop. We built the meta-model presented in Figure 3. This model represents only the interactions between a single WO and its own feedback loop. We made the decision not to present the entire WO System because we did not change interactions between WOs. The meta-model is detailed

component by component in the next sections.

5.1 Introduction of the *Idle* Status

In the previous works, we had 2 possible states of the object: *Dream* and *Awake*. While the system is using the WO, it is in the *Awake* state. When the object is freed, it falls directly in the *Dream* state.

When it was not used, the object was therefore necessarily in the *Dream* state (even if it had nothing interesting to dream about). This has several disadvantages: consumption of unnecessary resources, production of redundant logs, higher complexity of the analysis. So we decided to add a third state *Idle*. This state allows the object not to perform any action when the system does not hold the object.

As shown in Figure 4, the *Dream* is now included in the new *Idle* state. The WO is not allowed to dream until it knows what it has to dream. The WO can dream when it is in the *Idle* state and receives a dream plan request. Once the dream is finished, the WO falls directly into the *Idle* state.

5.2 MAPE-K Compliance

The MAPE-K architecture has many advantages regarding the Separation of Concerns. Most of the MAPE component were present in the precedent work but we propose a new architecture to disentangle them. This is a huge step forward in terms of maintenance and re-usability.

5.2.1 Monitoring, Analysis & Knowledge

In the precedent architecture, the WO shared too much responsibilities with other parts of the system. The WO should only do his business job without worrying about the other components of the MAPE loop. As shown in Figure 5, we separate the Monitoring/Analysis/Knowledge steps because this would help along interoperability of components.

This job is carried out by a knowledge generator that monitors the WiseObject. At each interaction with the WO, the generator will log in the memory the methods and their effects on the WO states. When the object is not being used by a system call, it falls in the *Idle* state. *Idle* state is described in Section 5.1.

We chose to use a WiseEventManager from Figure 3 inside the memory. This manager has the role to store the data in an appropriate passive storage.

5.2.2 Planner

In the previous architecture, the planner did not exist. We want the planner to be standalone in order to deal with the Separation of Concerns. In the *Awake* state, the planner uses the results of the analysis in order to decide if the original call for the object has to be modified. If it notices an irregularity, it generates an action plan to correct it. In the *Dream* state, the planner decides to use the parameter analysis to send an action plan in order to discover new business states. The planner is mandatory for the WO to dream with a consistency close to the real use.

As explained in Figure 6, the planner listens to the analysis results. After consulting the policy, it makes a decision whether or not it should create an action plan. Policies (Kephart and Chess, 2003) are defined as a high-level configuration file. It contains business objectives that can be different if we want to use the same system in diverse environments. For example, if the policy is set with max_unusual_uptime=2h, the planner consumes this plan while looking for the analysis. A switch unusually *Up* on 16/03 from 6am to 7am will not generate a plan.

In our architecture, we choose to generate a planner in parallel to an analyzer, but our architecture is open enough to define a specific planner that takes into account the results from several analyzers.

5.2.3 Executor

The executor did not exist in the previous version. It joins all the plans made by the planner so that the object can understand what it has to do. Unlike many self-adaptive applications, the executor has an additional responsibility. It has to call/modify the calls made for the object according to the states (*Dream*/*Awake*) of the WO (Executor Manager Figure 3). Following the Event Driven Architecture of WOF, the executor is a listener of planner (see Figure 7). The executor parses the plans generated by the planner in order to assign them to the appropriate executor. The Reaction Executor has been introduced to apply *Awake* plans that have an impact on the real system. Meanwhile, the Dreamer only applies *Dream* plans. The Dreamer's actions have no impact on the system since in the *Dream* state the WO is disconnected from the bus.

5.3 Multidimensional SoC

We introduced IAPE-K (Alloui and Vernier, 2017) in our previous paper. IAPE-K is nothing else than an adaptation of MAPE-K when the object is in the *Dream* state. As we need to track the system states, we must store them in a different place depending on their high-level state. Dream logs and real logs are very similar. It makes sense to store them in the same data structure. The only difference will be a flag

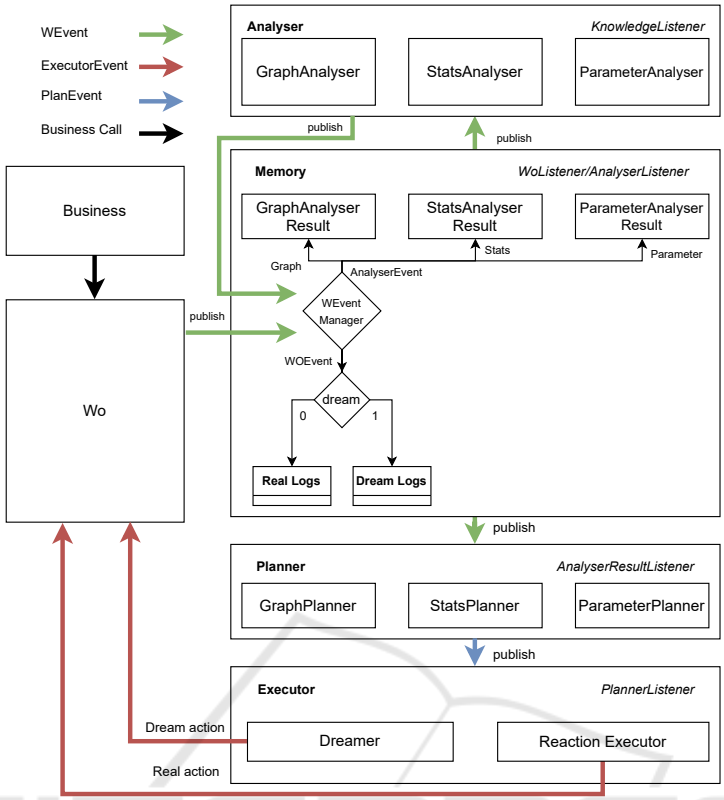

Figure 3: Meta-model of the new architecture.

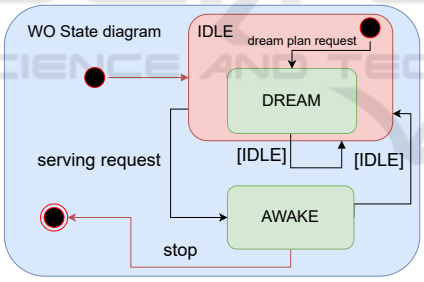

Figure 4: New WO high-level states.

containing the state of the object. Since the knowledge generator monitors all the states of the object, it knows where to store the usage logs.

A manager is thus necessary within the knowledge, it will have the role to allocate the logs to the right place of the knowledge. It receives data from diverse sources WO(*Awake*/*Dream*), Analyzers and put them in their own storage. Note that a manager is not mandatory in the Plan/Execute steps. As opposed to Memory, Planner/Executor are active elements. They can therefore dispatch the data themselves.

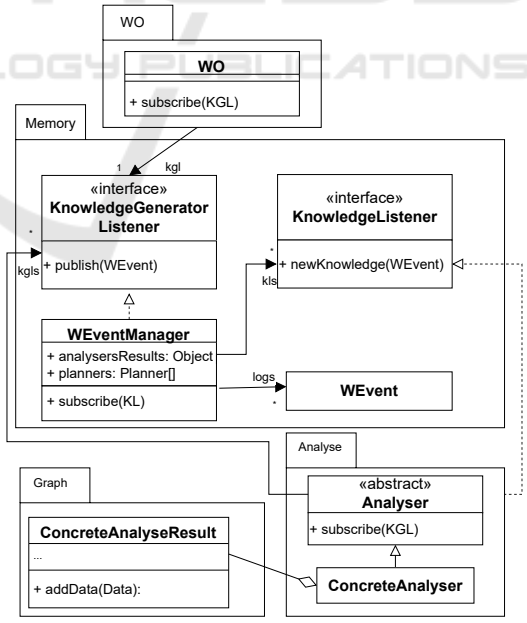

Figure 5: Static model of Monitor, Analysis & Knowledge.

5.3.1 Hyperslicing

In our framework, we need to separate the concerns horizontally (with the MAPE components) but we

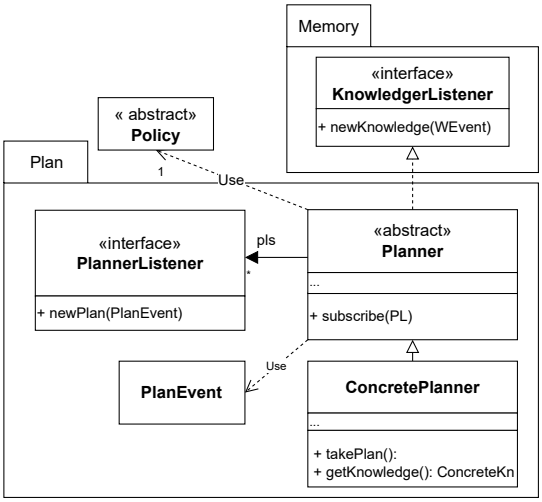

Figure 6: Static model of Knowledge & Plan.

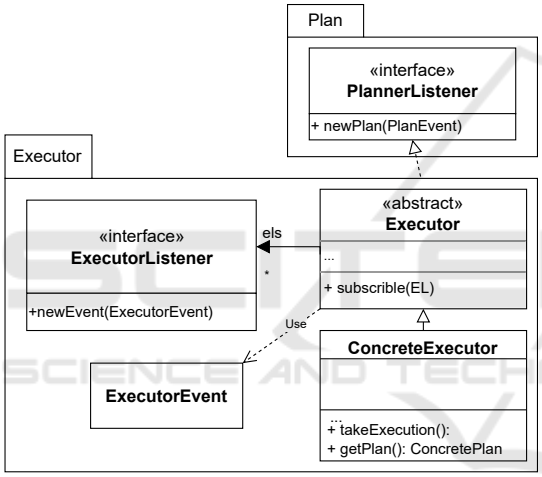

Figure 7: Static model of Plan & Execute.

also need to separate the concerns of interchangeable modules among different dimensions. (Tarr et al., 1999) proposes to split the system into "hyperslices". Hyperslices are hyper-planes that encapsulate concerns among multiple dimensions. We chose to use the hyperslices as a representation for patterns. For example, we encourage analyzer slices to have the same entry point i.e. Figure 8 *publish(WEvent)*. By doing so we have all the benefits of an Event-Driven system while keeping different functioning in the boxes below. For example, Analyzers primitive units (WEvent) are similar but how they update their knowledge will be different. In Figure 8, we represent an example of some MAPE loop components along the horizontal axis. Similar slices are also regrouped into hyper-modules where we can see feature decomposition along the vertical axis. The knowledge hyper-module can be seen as a data dimension. Even if the methods have the same names, knowledge slices store their *WEvent* at different places. It also publishes to different modules like Planners, Analyzers according to the type of *WEvent*. It is worth to note that there is no relation along the vertical axis between modules.

5.3.2 Analyzers

Analyzers are relatively similar in their operation. For each generation of knowledge, they receive associated data and then send back a conclusion. It therefore makes sense to imagine them in the same dimension of concerns. However, they must be modular so that new ones can be easily inserted without redesigning the whole knowledge block. The multidimensional Separation of Concerns is very useful in our analysis management. Each analysis will surely have different dependencies (statistics, deep learning, set theory...). Since they do not interact directly with each other, this makes modularization easier by creating several independent layers. We will use the notion of Hyper-Slices from (Tarr et al., 1999) in order to increase dimensions with each addition of analyzer. For the moment the analyzers do not interact with each other as the problem of knowledge fusion is a vast domain on which we have not yet worked.

6 CONCLUDING REMARKS

In this paper, we address the maintainability issue of self-adaptive wise systems. To overcome the problem of costly reuse and evolution of a monolithic architecture where a same entity (WO) embed several responsibilities, we propose a new architecture based on the separation of concerns and on an even-driven publish/subscribe mechanism. Separation of concerns is done according to several dimensions: MAPE-K loop model, Real knowledgevs. Dream knowledge, components with similar responsibilities. To complete the MAPE-K loop, we also implemented Plan construction and Execution components that were absent in the previous version of WOF.

This results in many organized slices that enable the system reuse, maintenance and evolution. At execution time, this architecture allows the system to easily swap among components with similar responsibility by stacking them in a same dimension. Furthermore, the publish/subscribe mechanism enables components to interact only with the relevant part of knowledge they need .

With this new architecture, we plan in the short term to develop Markovian and deep learning analyz-

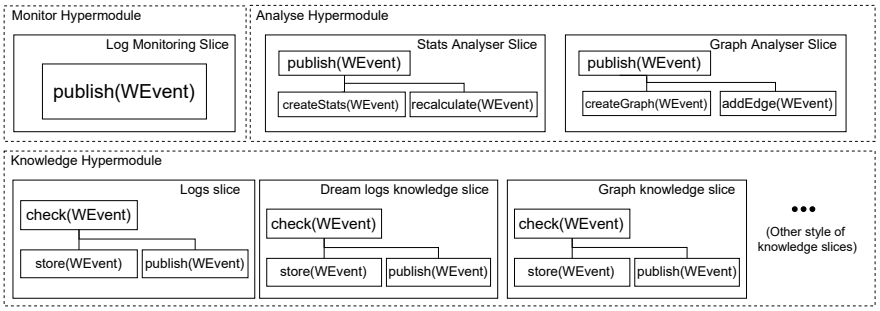

Figure 8: Example of components of the MAPE-K loop using hyperslicing visualization.

ers in addition to the existing graph and statistical analyzers. Developing different analyzers is crucial to the reliability of self-adaptive systems: analyzers may either discover faster new states of the monitored object or draw better conclusions about its functioning. Moreover analyzers of operation's parameters seems essential for both the credibility of the system and resource saving: we can develop a statistical analyzer that according to the parameter type generates new values by sampling in the distribution generated in past analyzes.

As we completed the MAPE-K model, we naturally will implement different Plan and Execute components dedicated to both Dream and Awake states of a WO. A first "dreamer" will be a smart dreamer that takes into account the WO experience (real data) instead of randomly simulating data. Regarding executors, the first one will simply authorize or not execution of the planned actions.

We also intend to study in future work how to improve the new architecture, by introducing a higher level where knowledge related to a WO can be merged, aggregated or simply used at system level.

REFERENCES

Alloui, I., Benoit, E., Perrin, S., and Vernier, F. (2018). Wiot: Interconnection between wise objects and iot. *13th International Conference on Software Technologies*.

Alloui, I., Esale, D., and Vernier, F. (2015). Wise objects for calm technology. *10th International Joint Conference on Software Technologies*.

Alloui, I. and Vernier, F. (2017). A wise object framework for distributed intelligent adaptive systems. *12th International Conference on Software Technologies*.

Brun, Y., Serugendo, G. D. M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Muller, H., Pezze, M., and Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. *Software engineering for self-adaptive systems*.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2009). Evaluating the effectiveness of the rainbow self-adaptive system. *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*.

Crow, L. (1990). Evaluating the reliability of repairable systems. *Annual proceedings on reliability and maintainability*.

Dijkstra and Edsger, W. (1982). On the role of scientific thought. In *Selected writings on computing: a personal perspective*, pages 60–66. Springer.

Gouda, G, M., Herman, and Ted (1991). Adaptive programming. *IEEE Transactions on Software Engineering*, 17(9):911–921.

Harrison, William, Ossher, and Harold (1993). Subject-oriented programming: a critique of pure objects. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 411–428.

Kephart, J. and Chess, D. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer.

Makabee, H. (2012). An event-driven approach for the separation of concerns. In *ENASE*, pages 122–127.

Moreira, A., Rashid, A., and Araújo, J. (2005). Multi-dimensional separation of concerns in requirements engineering. In *IEEE International Conference on Requirements Engineering*.

Naqvi, M. (2012). Claims and supporting evidence for selfadaptive systems – a literature review. page 47. Linnaeus University, School of Computer Science, Physics and Mathematics.

Ossher, H. and Tarr, P. (2002). Multi-dimensional separation of concerns and the hyperspace approach. *Software Architectures and Component Technology*.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Software Pioneers*.

Ravindranathan, M. and Leitch, R. (1998). Heterogeneous intelligent control systems. *IEE Proceedings-Control Theory and Applications*.

Tarr, P., Ossher, H., Harrison, W., and Sutton, S. (1999). N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering (ICSE)*.