IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 28 May 2024, accepted 27 August 2024, date of publication 9 September 2024, date of current version 18 September 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3456041

RESEARCH ARTICLE

AFD–An Architectural Language for Integral Modeling

STEFAN TUBIĆ, ZAHARIJE RADIVOJEVIĆ, SAŠA STOJANOVIĆ, AND MILOŠ CVETANOVIĆ
School of Electrical Engineering, University of Belgrade, 11000 Belgrade, Serbia

Corresponding author: Stefan Tubić (stefan.tubic@etf.bg.ac.rs)

This work was supported by the Ministry of Science, Technological Development, and Innovation of Serbia under Contract 451-03-65/2024-03/200103.

ABSTRACT Describing architectures of complex software systems using architectural languages is usually done through multiple viewpoints that enable the creation of views. While the creation of views enables the separation of stakeholders' concerns with the system and eases manageability, it raises the problem of inconsistencies among the views. This paper presents Annotated Functional Decomposition (AFD), an architectural language that provides integral modeling as a possible solution to this problem. Integral modeling creates a model by decomposing a system into its functions, which are annotated to simultaneously create multiple views. Having all created views available in the model at the same time facilitates inconsistency management. AFD supports automated inconsistency detection and manual inconsistency resolution. Moreover, AFD supports the automated translation of views to appropriate UML diagrams, which facilitates adaptation to other methodological approaches. According to the criteria used in the literature for the evaluation of 124 architectural languages, AFD provides nine out of 12 requirements that are important to practitioners.

INDEX TERMS Architectural language, viewpoints, consistency, integral modeling, UML.

I. INTRODUCTION

Developing large software systems often results in complex architectures, and over time, it has become a practice to design, build, maintain, and analyze architectural descriptions from multiple viewpoints [1]. Viewpoints enable different stakeholders to focus on details based on their concerns. Observing the system through viewpoints results in the creation of views. A large set of views may cause stakeholders to end up with architectural descriptions that are difficult to manage. Moreover, the description further requires ensuring consistency between many different views and thus hinders manageability. Ensuring consistency is a complex process encompassing activities ranging from detection through resolution, all the way to tracking of inconsistencies.

Architectural languages and their tools are used to describe architectures and address the complexity of software systems in different ways. Determining the weaknesses and strengths of each architectural language can be done in terms of requirements that are highly important to practitioners [2]. Requirements concerned with defining language syntax and semantics distinguish visual and textual languages. Similarly, how a language enables specifying, modifying, and maintaining architectural descriptions encompasses a requirement regarding support for multiple viewpoints. Additionally, the operational usage of languages depends on tool support, which among others encompasses a requirement for inconsistency management.

Even though a requirement for multiple viewpoints support is highly desirable as it enables the separation of concerns, it makes the problem of keeping views consistent more challenging. The creation of views is usually done independently in separate places, either through separate diagrams or textual descriptions. A greater number of views makes system design and understanding easier but inherently makes the problem of detection and resolution of inconsistencies among views more difficult.

This paper presents Annotated Functional Decomposition (AFD), a textual architectural language that enables the creation of architectural descriptions of complex software

The associate editor coordinating the review of this manuscript and approving it for publication was Fabrizio Messina.

systems using functional decomposition as a methodological design paradigm. In addition to functional decomposition, AFD introduces annotations to support multiple viewpoints and creation of views. Views in AFD are created together in the same textual description, leading to an integral model that should ease some inconsistency management activities. Additionally, this paper presents an AFD tool that supports the operational usage of AFD, not only for the creation of an integral model, but also for the separation of the integral model into views. Moreover, the AFD tool enables the transformation of an architectural description given in the AFD to the appropriate UML diagrams.

The remainder of this paper is organized as follows. The second section presents a related work. The third section introduces the AFD and its syntax and provides an example of integral modeling in AFD. The fourth section explains how viewpoints are represented in AFD and how AFD supports inconsistency management activities. The fifth section describes the AFD Tool, its role in facilitating the manageability of architectural descriptions, and depicts the generated UML for the example given in the third section. The sixth section evaluates the AFD in the context of other architectural languages. The seventh section concludes the paper.

II. RELATED WORK

Modeling complex software systems through multiple views represents an approach that praises a differentiated and complex scientific body of knowledge [3] in the domain of viewpoints. Contributions made in the last couple of decades have testified to the importance of multiple-viewpoint usage. In 1995. Soni et al. [4] introduced the conceptual, module, execution, and code viewpoints. In the same year, Kruchten [5] introduced four mandatory viewpoints: logical, process, physical, and development, and one optional viewpoint: scenarios. In 2000. IEEE created standard "IEEE 1471-2000" now known as "ISO/IEC 42010:2007" [6], which introduced the concept of viewpoints to capture common descriptive frameworks across many systems. Unlike approaches that prescribe a fixed set of viewpoints, this standard advocates creating a set of viewpoints that best serves the stakeholders and their concerns associated with a system. In 2002. Clements et al. [7] introduced 17 viewpoints categorized as module, component-and-connector, allocation, and hybrid styles. In 2002. Garland and Anthony [8] introduced 14 viewpoints categorized as conceptual and analysis, logical design, and environment/physical. In 2005. Rozanski and Woods [9] introduced context, functional, information, concurrency, development, deployment, and operational viewpoints. In 2009. Taylor et al. [10] introduced logical, physical, deployment, concurrency, and behavioral viewpoints. In 2011. IEEE created standard "ISO/IEC/IEEE 42010:2011(E)" [11] which is a revision of the previous standard and still does not prescribe a fixed set of viewpoints. In 2018. Ozkaya [2] introduced the logical, information, physical, deployment, behavior, concurrency, development, and operational viewpoints.

However, the usage of viewpoints and the creation of views raise the question of consistency among them. A couple of inconsistency management frameworks that have been defined throughout the years were unified by Spanoudakis and Zisman [12] as a set of activities that more accurately reflect the operationalization of the inconsistency management process by the various techniques and methods that have been developed to support it. These activities are the detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, tracking of inconsistencies, and specification and application of a management policy for inconsistencies, some of which can be performed using a certain technique or method. In their systematic literature review, Cicchetti et al. [3] identified a similar set of activities for the inconsistency management process, followed by their set of techniques and methods. Moreover, Cicchetti et al. argued that whenever multiple views are created separately, achieving consistency in architectural descriptions is based on resolution mechanisms specifically defined on pairs of views. The intricacy of achieving consistency is represented by the necessity of defining many binary consistency relations and corresponding restoring procedures, and possibly arising the ripple effect that might lead to a non-confluent process. The problem of n-ary consistency relations remains unresolved, as it poses severe difficulties in both theoretical and practical aspects.

In the analysis of existing architectural languages, Ozkaya [2] evaluated both visual and textual languages and showed that most of them provide support for multiple viewpoints. However, only a portion of the analyzed languages deal with inconsistency management. Most languages supporting multiple viewpoints create corresponding views separately, either by multiple diagrams or textual descriptions, which can hinder certain activities of inconsistency management. In contrast to existing languages, this paper proposes a language that supports multiple viewpoints but creates corresponding views by bringing them together as an integral model. Dealing with views on the integral model is expected to ease certain inconsistency management activities.

III. AFD

Annotated Functional Decomposition (AFD) is a textual architectural language that enables the creation of architectural descriptions of complex software systems. The idea behind AFD is to use functional decomposition as a methodological design paradigm to ease comprehension by seeing a system through its constituent parts. Furthermore, by introducing annotations, AFD extends functional decomposition and implements all four pillars of computational thinking [13] as a methodological problem-solving approach. To facilitate understanding, the annotations are represented as five levels of decomposition. The first level is mandatory, while the other four are orthogonal, and therefore optional. The first level describes the decomposition, the second describes the control flow, and the third describes the data flow,the fourth describes reuse, and the fifth describes implementation. Each level

```
1   Payment
2       Input
3       CreatePayment
4       TryPayment
5           GetOrderSum
6               GetItemsForOrderId
7                   GetDatabasePersistenceManager
8                   GetOrder
9                   GetItems
10              InitialSum
11              GetSumForItems
12                  GetItemPrice
13                  GetItemCount
14                  IncreaseSum
15          Transaction
16          TransactionDone
17              PaymentSuccesful
18                  WriteSuccessToLog
19                  ChangeToSuccesful
20              PaymentFailed
21                  WriteFailureToLog
22                  ChangeToFailed
23      RecordPayment
24          GetDatabasePersistenceManager
25          SavePayment
26          GetItemsForOrderId
27          UpdateInventory
28      Output
```

FIGURE 1. The first level of decomposition – Functions for Retail payment process example.

```
1   1 Payment
2       1 Input
3       2 CreatePayment
4       3 TryPayment
5           1 GetOrderSum
6               1 GetItemsForOrderId
7                   1 GetDatabasePersistenceManager
8                   2 GetOrder
9                   3 GetItems
10              2 InitialSum
11              3* GetSumForItems /item in items
12                  1 GetItemPrice
13                  2 GetItemCount
14                  3 IncreaseSum
15          2 Transaction
16          3 TransactionDone
17              1e PaymentSuccesful /status==200 AND (Initialized)
18                  1 WriteSuccessToLog
19                  2 ChangeToSuccesful
20              1e PaymentFailed /status!=200 AND (Initialized)
21                  1 WriteFailureToLog
22                  2 ChangeToFailed
23      4? RecordPayment /status==200
24          1 GetDatabasePersistenceManager
25          2p SavePayment
26          2p GetItemsForOrderId
27          3 UpdateInventory
28      5 Output
```

FIGURE 2. The second level of decomposition – Control flow for Retail payment process example.

```
1   Payment(=>PaymentInput,<=PaymentOutput)
2       Input(=>PaymentInput,<ordId,<CCNum)
3       CreatePayment(<payment,>ordId,>CCNum)
4       TryPayment(<>payment,<status)
5           GetOrderSum(>payment,<sum)
6               GetItemsForOrderId(>payment.ordId,<items)
7                   GetDatabasePersistenceManager(<dbPersistenceManager)
8                   GetOrder(>dbPersistenceManager,>ordId,<order)
9                   GetItems(>order,<order.items)
10              InitialSum(<sum)
11              GetSumForItems(>payment,>items,<>sum)
12                  GetItemPrice(>item,<price)
13                  GetItemCount(>item,<count)
14                  IncreaseSum(>payment,>price,>count,<>sum)
15          Transaction(>payment.CCNum,>sum,<status)
16          TransactionDone(<>payment,>status)
17              PaymentSuccesful(<>payment)
18                  WriteSuccessToLog(>payment)
19                  ChangeToSuccesful(<>payment)
20              PaymentFailed(<>payment)
21                  WriteFailureToLog(>payment)
22                  ChangeToFailed(<>payment)
23      RecordPayment(>payment)
24          GetDatabasePersistenceManager(<dbPersistenceManager)
25          SavePayment(>dbPersistenceManager,>payment)
26          GetItemsForOrderId(>payment.ordId,<items)
27          UpdateInventory(>items)
28      Output(>status,<=PaymentOutput)
```

FIGURE 3. The third level of decomposition - Data flow for Retail payment process example.

will be further explained in the example given in Figs. 1-5, depicting a simplified retail payment process.

The first level of decomposition represents the basic structure of a system by defining its functions (Fig. 1). The system's function can be further decomposed into subfunctions and represented by indenting relative to each other. Functions defined in this level can be further described with annotations that form other decomposition levels.

The second level of decomposition represents a set of annotations that defines the control flow of a system (Fig. 2). The control flow defines the execution order of the functions, conditional executions, loop executions, and parallel executions. The execution order of the functions is defined by writing ordinal numbers before function definition. The conditional and loop execution of a function are achieved by writing a condition after the function name, marking it for conditional or loop execution, respectively. A group of functions can be marked for exclusive execution, such as functions in lines 17 and 20, and for parallel execution, such as functions in lines 25 and 26.

The third level of decomposition represents a set of annotations that defines the data flow of a system (Fig. 3). The data flow defines the data input objects, data output objects, input streams, and output streams. The data object can be input to a function such as the payment data object in line 5, and the function can output a data object such as the data object sum in line 5. Data objects can have parts that are data objects, as depicted in line 6, where ordId is the data object and is a part of the payment data object. The input stream can be decomposed into a set of data objects as depicted in line 2, where the input stream PaymentInput is decomposed into ordId and CCNum data objects, whereas the output stream can be composed of a set of data objects as depicted in line 28, where the PaymentOutput output stream is composed of status data object. The input and output streams represent data that flow through a system at a higher level of abstraction in the decomposition.

The fourth level of decomposition represents a set of annotations that enables the reuse of already defined functions (Fig. 4). Function can be marked to be reused as a function in line 6. Other functions can be marked to reuse the reusable function such as function in line 26.

The fifth level of decomposition represents a set of annotations that defines the implementation aspect of a system. In addition to the implementation aspect, Fig. 5 shows the remaining four levels of decomposition for completeness of the example. The implementation aspect defines the executors of functions, states of data objects, types of data objects, components, nodes, resources, and actors. An executor is part of a system that is responsible for functionexecution and is defined inside square brackets after a function definition. The executor can be a method of a class, such as the method cPayment of a class Payment in line 3, a method of an object

```
1   Payment
2           Input
3           CreatePayment
4           TryPayment
5                   GetOrderSum
6                           GetItemsForOrderId#
7                                   GetDatabasePersistenceManager
8                                   GetOrder
9                                   GetItems
10                          InitialSum
11                          GetSumForItems
12                                  GetItemPrice
13                                  GetItemCount
14                                  IncreaseSum
15                  Transaction
16                  TransactionDone
17                          PaymentSuccesful
18                                  WriteSuccessToLog
19                                  ChangeToSuccesful
20                          PaymentFailed
21                                  WriteFailureToLog
22                                  ChangeToFailed
23          RecordPayment
24                  GetDatabasePersistenceManager
25                  SavePayment
26                  #GetItemsForOrderId
27                  UpdateInventory
28          Output
```

FIGURE 4. The fourth level of decomposition - Reuse for Retail payment process example.

like tryPayment of an object Payment in line 4, or a service like the bank service in line 15.

Data objects can define states and state transitions. The state is written in parentheses after the name of the data object. In line 3, a data object payment is in the state Initialized. After a transaction is done, the state of the payment data object is changed to Successful in line 19 when the transaction is successful, and the payment object is in the state Initialized. After a transaction is done, the state of the payment data object is changed to Failed in line 22 when the transaction is not successful, and the payment object is in the state Initialized.

The data type can be defined for a data object. Moreover, the hierarchy of data types can also be defined. The data type is written after a data object name and colon (:) symbol, and can be a primitive or class data type. In line 2, the order id and credit card number data objects are primitive-type integers. In line 3, the payment data object is a class-type Payment. In line 7, the persistence manager data object is of class type StoreDatabasePersistenceManager, which extends the class PersistenceManager. In line 9, items data object is a collection of objects of the class-type Item.

Data types can be grouped into system components. Components are logical groups of class types. A physical artifact such a file can manifest a logical component. After writing the class name, the component to which it belongs can be defined in parentheses. The component is defined by writing a physical name and logical name of the component and separating them by the colon symbol (:). In line 1, class Store belongs to a component Store that is physically stored in file store.jar. A component Store is a subcomponent of a component System. In line 8, class Order belongs to the component Persistence, whose physical name has not yet been defined.

Execution of the function is done on a node. A node is a generic machine in which artifacts are deployed, and functions are executed. A node can have an instance that represents the existing machine. A node is defined by writing at symbol (@), followed by a logical name for a node and a physical name for a node instance. The function in line 4 is marked to be executed on a FinanceServer node on the existing machine identified with the URL www.finance.com.

A function can be related to certain resources. The execution of a function may require access to a database or development of a function may require testing examples, or the operational usage of a function may require information related to installation and configuration. Resource usage is defined as the operation performed on a resource. A resource can have a defined type. Resource usage is written in curly brackets after a function definition. In line 25, the payment is stored in a table Payment using the SQL insert operation.

The function can have assigned actors. Actors are entities outside the system that can use the functions of the system. Actors that interact with a function can be defined by writing the circumflex symbol (^), followed by their names. The payment function in line 1 can be explicitly used by a system administrator, which implicates the usage of all its sub-functions. Similarly, the function in line 4, which tries a payment, can be used explicitly by the finance sector.

All previously described annotations are defined in the AFD through its context-free grammar, which consists of 41 rules. The rules and coverage of the decomposition levels by the rules are given in the appendix of this paper. An overview of some annotations used in the Retail payment process example is depicted in Table 1. For each level of decomposition, the table provides annotations and their location in the example, followed by brief explanations.

IV. INTEGRAL MODELING AND INCONSISTENCY MANAGEMENT IN AFD

AFD is characterized by its ability to model a complex software system by decomposing it through five levels of decomposition. Each level of decomposition is represented with certain AFD annotations. In decomposition, all annotations are written side by side, and therefore, an integral model of the entire software system is created. Having an integral model that contains all information about the system is not limiting, as it still supports the separation of concerns associated with the system because the introduced levels of decomposition may be mapped to appropriate viewpoints that are identified in the literature.

Over the years, various approaches have introduced different sets of viewpoints. The sets differ from each other in terms of the number, naming, and meaning of viewpoints. However, the existing standard does not prescribe a fixed set of viewpoints; therefore, to demonstrate the usage of viewpoints in AFD, an example viewpoint set contains

```
1  1 Payment(=>PaymentInput,<=PaymentOutput)[C:Store(store.jar:System.Store)]^SystemAdmin@StoreServer:www.store.com
2        1 Input(=>PaymentInput,<ordId:int,<CCNum:int)
3        2 CreatePayment(<payment(Initialized):Payment,>ordId,>CCNum)[C:Payment:cPayment]
4        3 TryPayment(<>payment,<status:int)[O:payment]^FinanceSector@FinanceServer:www.finance.com
5              1 GetOrderSum(>payment,<sum)
6                    1 GetItemsForOrderId#(>payment.ordId,<items)^DBAdmin,DBStoreOwner,DBPrivilegedUser
7                          1 GetDatabasePersistenceManager(<dbPersistenceManager:StoreDatabasePersistenceManager-
   >PersistenceManager(persistence.jar:Persistence))[C:PersistenceManagerFactory]
8                          2 GetOrder(>dbPersistenceManager,>ordId,<order:Order(:Persistence))[O:dbPersistenceManager]
9                          3 GetItems(>order,<order.items:{}Item(persistence.jar:Persistence))
10                   2 InitialSum(<sum)
11                   3* GetSumForItems(>payment,>items,<>sum:int) /item in items
12                         1 GetItemPrice(>item,<price:float)[O:item]
13                         2 GetItemCount(>item,<count:int)[O:item]
14                         3 IncreaseSum(>payment,>price,>count,<>sum)[O:payment]
15             2 Transaction(>payment.CCNum:int,>sum,<status:int)[S:BankService]@BankServer:www.bank.com
16             3 TransactionDone(<>payment,>status)[O:payment(payment.jar:System.Payment)]
17                   1e PaymentSuccesful(<>payment) /status==200 AND (Initialized)
18                         1 WriteSuccessToLog(>payment)[O:payment(payment.jar:System.Payment)]
19                         2 ChangeToSuccesful(<>payment(Succesful))
20                   1e PaymentFailed(<>payment) /status!=200 AND (Initialized)
21                         1 WriteFailureToLog(>payment)[O:payment:writeFailToLog]
22                         2 ChangeToFailed(<>payment(Failed))
23       4? RecordPayment(>payment) /status==200^DBAdmin,DBStoreOwner
24             1 GetDatabasePersistenceManager(<dbPersistenceManager:StoreDatabasePersistenceManager)[C:PersistenceManagerFactory]
25             2p SavePayment(>dbPersistenceManager,>payment)[O:dbPersistenceManager]{I-PAYMENT:TABLE@DatabaseServer:www.db.com}
26             2p #GetItemsForOrderId(>payment.ordId:int,<items)
27             3 UpdateInventory(>items)[C:Store]
28       5 Output(>status,<=PaymentOutput)
```

FIGURE 5. The fifth level of decomposition - Implementation for Retail payment process example.

TABLE 1. Overview of some annotations used in Retail payment process example.

Decomposition level	Annotations	Line in the example	Explanation
1st level - Decomposition	TryPayment	4	Function.
2nd level - Control flow	5	27	Ordinal number of a function.
	*	10	Function will be executed in a loop.
	?	22	Function will be conditionally executed.
	e	16	Function is exclusive.
	p	24	Function will be executed in parallel.
	/item in items	10	Loop condition for function execution.
	/status==200	22	Boolean expression as a condition for execution.
3rd level - Data flow	>ordId	3	Data input.
	<sum	5	Data output.
	<>sum	10	Data input and output.
	=>PaymentInput	1	Input stream.
	<=PaymentOutput	1	Output stream.
	payment.ordId	6	Data part.
4th level - Reusage	GetItemsForOrderId#	6	Definition of reusable function.
	#GetItemsForOrderId	25	Function reuse.
5th level - Resource flow	C:Payment:cPayment	3	Method of a class that executes the function.
	O:payment:writeFailToLog	20	Method of an object that that executes the function.
	Initialized	3	State of a data object.
	Payment	3	Type of data object.
	payment.jar:System.Payment	15	Physical and logical component for a type.
	StoreServer:www.store.com	1	Node and node instance.
	I-PAYMENT:TABLE	24	Operation done on a resource.
	DBAdmin	6	Actor.

the following viewpoints: functional, execution, information, implementation, data state, component, deployment, context, and resource. The functional viewpoint describes a system in terms of its basic functions and the relationships between them. The execution viewpoint describes the control flow of the system. The information viewpoint describes the data and relationships between the data and data flow. The implementation viewpoint describes the data types, relationships between data types, and parts of a system that executes functions. The data state viewpoint describes the state of the data and the transitions between data states. The component viewpoint describes a system as a set of logical components and their physical manifestations. The deployment viewpoint defines how the software components are mapped to the hardware. The context viewpoint defines external system actors, and how they interact with a system through use cases. The resource viewpoint defines operations on logical and physical resources that the system uses, and may also be used to describe nonfunctional requirements [14] [15] or represent resources needed for development and operational

TABLE 2. Mapping of example viewpoints to the viewpoints in literature.

Viewpoint set #	Viewpoints in literature	Functional	Execution	Information	Implementation	Data state	Component	Deployment	Context	Resource
1	Conceptual								X	
	Module	X								
	Execution		X	X	X	X				
	Code						X	X		X
2	Logical	X		X	X	X				X
	Process		X							
	Physical						X	X		X
	Development				X					X
	Scenarios								X	
3	Module styles	X		X					X	X
	Component-and-connector styles		X	X			X			X
	Allocation styles				X	X	X	X		X
	Hybrid styles	X	X	X	X	X	X	X	X	X
4	Conceptual and analysis viewpoints								X	
	Logical design viewpoints	X	X	X	X	X	X			X
	Environment/physical viewpoints						X	X		X
5	Context								X	
	Functional	X								
	Information			X	X	X				X
	Concurrency		X							
	Development		X	X	X	X	X			X
	Deployment						X	X		X
	Operational									X
6	Logical	X	X	X	X	X	X		X	X
	Physical							X		X
	Deployment						X	X		X
	Concurrency		X							
	Behavioral		X	X	X	X				
7	Logical	X					X		X	
	Information			X		X				
	Physical							X		
	Deployment							X		
	Behavioral		X							
	Concurrency		X							
	Development				X					X
	Operational									X

processes. The mapping between the example viewpoint set and the viewpoints in the literature is presented in Table 2. All viewpoint sets in Table 2 share similar concerns associated with a system.

The introduced viewpoints were mapped to the levels of decomposition and AFD annotations, as shown in Table 3. The existing relationships between the levels of decomposition are the cause of the relationships between viewpoints and the appropriate views. Creating an integral model starts with functions; therefore, a functional view of the system is created. Other views are created later, and they always directly or indirectly annotate the functional view, as depicted in Fig. 6. The data presented in the information view can be part of a condition defined in the execution view. The implementation view annotates the information view; however, the data presented in the information view can be a part of the executor defined in the implementation view. The data state view annotates the information view, and may present states that can be part of a condition defined in the execution view.

TABLE 3. Mapping the example viewpoint set to the levels of decomposition and AFD annotations.

Example viewpoints	1st level - Decomposition	2nd level - Control flow	3rd level - Data flow	4th level - Reusage	5th level - Executors	5th level - States	5th level - Types	5th level - Components	5th level - Nodes	5th level - Resources	5th level - Actors
Functional	X			X							
Execution		X									
Information			X								
Implementation					X		X				
Data state						X					
Component								X			
Deployment									X		
Context											X
Resource										X	

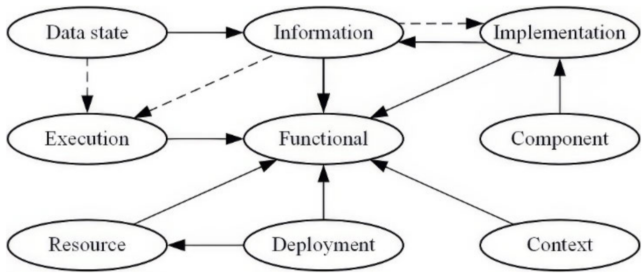

FIGURE 6. Relationships between Viewpoints. Full arrow – annotation relationship, dashed arrow – "part of" relationship.

The component view annotates the implementation view and the deployment view annotates the resource view.

Creating a model as a set of views could create consistency problems if the rules defined for the viewpoints are not satisfied. Therefore, the need for inconsistency management arises. Cicchetti et al. analyzed 40 research studies and defined a taxonomy for characterizing solutions for multi-view modeling [3] and listed activities regarding inconsistency management, such as specification of overlaps, inconsistency detection, and inconsistency resolution. Even broader set of activities is listed in the framework defined by Spanoudakis and Zisman [12], which besides specification and application of an inconsistency management policy includes the detection of overlaps, detection of inconsistencies, diagnosis of inconsistencies, handling of inconsistencies, and tracking of inconsistencies. By providing integral modeling, AFD facilitates activities related to the detection of overlaps, detection of inconsistencies, and handling of inconsistencies and uses some of the techniques and methods mentioned by Spanoudakis and Cicchetti. The diagnosis and tracking of inconsistencies are not supported by AFD.

The detection of overlaps in AFD is done using Similarity Analysis. According to Spanoudakis' terminology Similarity Analysis is performed using automated comparisons between

TABLE 4. Pairs of viewpoints that can have overlapping elements. Y – Yes, N – No, / - Not Applicable.

	Functional	Execution	Information	Implementation	Component	Data state	Context	Resource	Deployment
Functional	/	N	N	N	N	N	N	N	N
Execution		/	Y	N	N	Y	N	N	N
Information			/	Y	N	N	N	N	N
Implementation				/	N	N	N	N	N
Component					/	N	N	N	N
Data state						/	N	N	N
Context							/	N	N
Resource								/	N
Deployment									/

views. Elements in AFD are described mainly by their name, so similarity analysis is performed by comparison of element names in views that belong to viewpoints that can have overlapping elements. Example viewpoints that can have overlapping elements are presented in Table 4. Overlapping elements are a consequence of a "part of" relationships between viewpoints as shown in Fig. 6. According to Cicchetti's taxonomy, AFD provides implicit detection of overlaps established through the use of conventions, such as naming and so forth.

The detection of inconstancies in AFD is done by checking the satisfiability of specific consistency rules using Special forms of analysis. Spanoudakis identified consistency rule categories that can be defined by the language and can be, among others, in the category of well-formedness or description identity rule. Well-formedness refers to rules that must be satisfied by the views for them to be legitimate views of the language in which they have been expressed. The AFD defines well-formedness rules for each viewpoint as a set of semantic checks. Example of well-formedness rule for Functional viewpoint is: "Function that is reusable has unique name in decomposition." An example of well-formedness rule for the implementation viewpoint is: "There must be no cycle in class inheritance." As AFD is a language in which views are written side-by-side, well-formedness rules are also defined for combinations of views. An example of a well-formedness rule for the combination of functional and execution viewpoints is: "Function must have an ordinal number in the prefix." Example of well-formedness rule for combination of Functional and Information viewpoints is: "Function that references reusable Function must have the same number of data flows as reusable Function." Description identity rules require different overlapping elements of views to have identical descriptions. In the case of AFD, descriptions of overlapping elements are their names; therefore, if two elements overlap, their descriptions are the same. These rules are always satisfied and are not explicitly defined. According to Cicchetti's taxonomy, AFD provides automated inconsistency detection using operational semantics.

Handling of inconsistencies in AFD is done using Synoptic technique. According to Spanoudakis' terminology Synoptic technique defines that stakeholders are involved in the generation of solutions for handling inconsistencies. Inconsistencies in views created by AFD, which do not satisfy the consistency rules, can be resolved by stakeholders. Therefore, the resolution of inconsistencies is a manual process performed by stakeholders. According to Cicchetti's taxonomy, AFD provides manual inconsistency resolution, which is delegated to stakeholders.

V. AFD TOOL

The AFD Tool provides support for the practical use of AFD and is available online at https://afd.etf.bg.ac.rs/. The tool enables stakeholders to create, change, analyze, and manage the architectural description of a system. The architectural description represented as an integral model in AFD can be separated into views according to the viewpoints selected by a stakeholder. Eventual inconsistencies among the views are detected by AFD Tool and presented in the integral model to be easier understood and handled by a stakeholder. Besides that, AFD Tool can translate the integral model into appropriate UML diagrams, which can eventually be used in other tools.

The AFD Tool is implemented as a web application and is depicted in Fig. 7. A menu bar is located on the top of the window. The menu bar contains File, Edit, Viewpoints, UML, and Help buttons. A toolbar is located on the left side of the window. The toolbar consists of a search button, save file button, Viewpoints side panel button, UML side panel button, and Settings side panel button. Depending on the selected button on the tool bar, the corresponding side panel is shown. A status bar is located at the bottom of the window. The status bar shows the line and column numbers of the cursor and file encoding. The central panel of the tool window consists of a text editor in the middle and line numbers on the left side.

The text editor contains an architectural description written in AFD and colored such that each color represents a different viewpoint. In addition to coloring, viewpoints can be shown or hidden using the Viewpoints side panel. Considering that the architectural description can consist of a number of functions that stakeholders are not currently managing, the AFD Tool also enables folding certain functions by clicking the arrows located just after the line numbers. Clicking an arrow left to a certain function folds all its sub-functions. After hiding certain viewpoints and folding certain sub-functions, the AFD Tool enables stakeholders to copy only the visible text of the architectural description.

The AFD Tool provides support for the inconsistency management process by allowing the detection of inconsistencies and presenting them to a stakeholder. Consistency checks can be turned on or off using the Check consistency button at

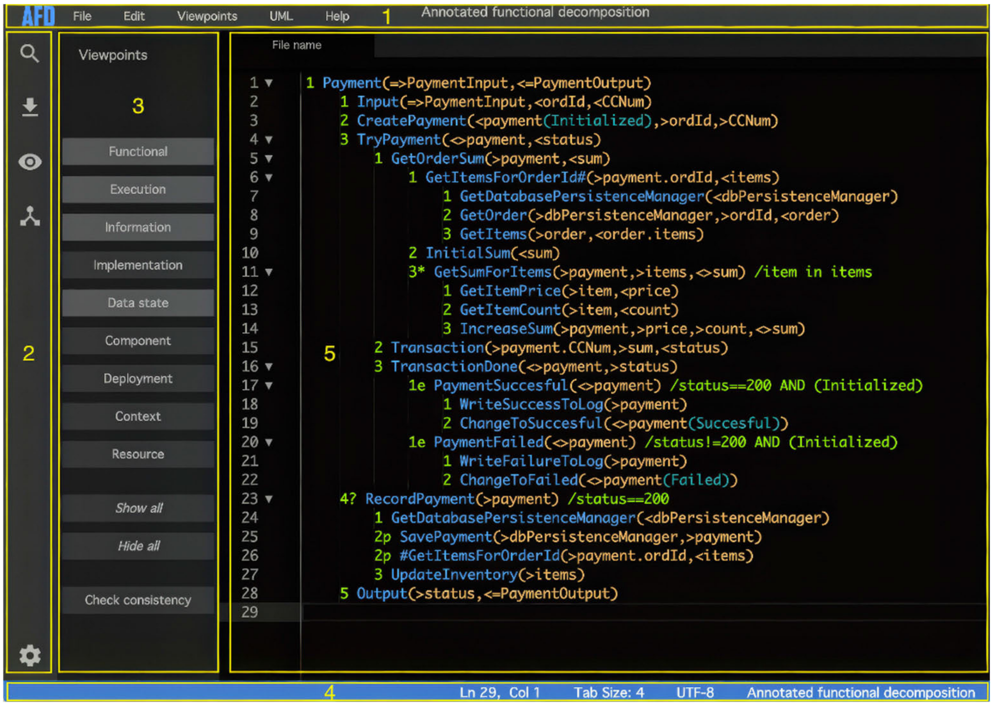

FIGURE 7. AFD Tool. 1 – Menu bar, 2 - Tool bar, 3 – Side panel, 4 – Status bar, 5 – Central panel.

the bottom of the Viewpoints side panel. Consistency rules that are not satisfied are presented to a stakeholder by the underlying parts of an architectural description that violate these rules and by marking lines in which violations are made. A description of a violated rule is visible after the mouse hovers over the underlined part of the architectural description. The detected inconsistencies are left to the stakeholders to handle them manually.

After handling all inconsistencies, if any, an architectural description can be translated by the AFD Tool into UML diagrams. The AFD Tool provides translation to seven UML diagrams: class, component, deployment, activity, sequence, state, and use-case diagram. Class, component, and deployment UML diagrams describe the structural design of a system, whereas activity, sequence, state, and use case UML diagrams describe the behavioral design of a system. Given an architectural description, not all views are necessary for the translation to a specific UML diagram. Table 5 lists the viewpoints used in the translation to a specific UML diagram type. The algorithms for the generation of UML diagram types are given in the appendix of the paper, while the diagrams generated for the Retail payment process example given in Fig. 5 are presented in Figs. 8-14. Figure 8 depicts the class diagram consisting of classes, their relationships, fields and methods. For example, line 7 contains StoreDatabasePersistanceManger->PersistanceManager, which is translated to the class diagram as classes StoreDatabasePersistanceManger and PersistanceManager, where the first one extends the second one. Figure 9 depicts component diagram consisting of components, their compositions, relationships and artifacts' manifestations of components. For example, lines 1 and 18 contain System.Store and System.Payment respectively which are translated to the component diagram into component System consisting of components Store and Payment. Figure 10 consists of two diagrams, the first depicting nodes and nodes' occurrences, and the second depicting artifacts' deployment on the identified nodes. For example, line 1 contains @StoreServer:www.store.com, which is translated to the node StoreServer and its occurrence www.store.com on the first deployment diagram. Line 1 contains store.jar:...StoreServer which is translated to the deployment of artifact store.jar to the node StoreServer on the second deployment diagram. Figure 11 consists of a set of activity diagrams defining system activities and its control

TABLE 5. Mapping of example viewpoints to UML diagrams.

	Class	Component	Deployment	Activity	Sequence	State	Use case
Functional	X	X	X	X	X	X	X
Execution				X	X	X	X
Information	X	X	X	X	X	X	
Implementation	X	X	X		X	X	
Component		X	X				
Data state						X	
Context							X
Resource			X				
Deployment			X				

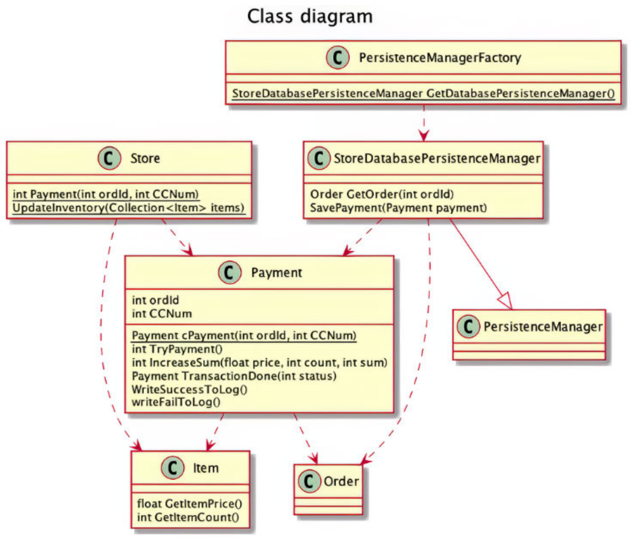

FIGURE 8. **Generated UML class diagram for Retail payment process example.**

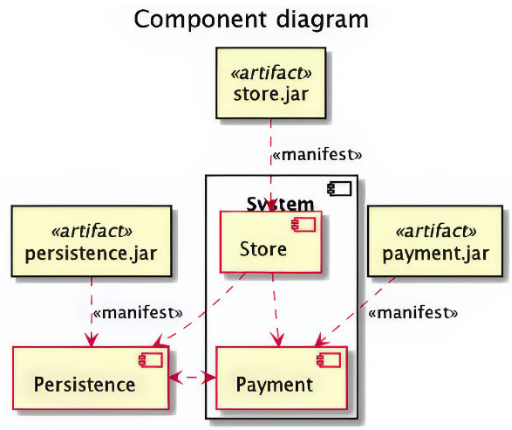

FIGURE 9. **Generated UML component diagram for Retail payment process example.**

flow. For example, line 11 contains GetSumForItems…/item in items which is translated on the second activity diagram to the loop condition represented by the decision node. Figure 12 depicts sequences of method calls in a set of generated sequence diagrams. For example, lines 4 and 12 contain TryPayment…[0:payment] and GetItemPrice…[0:item] respectively, which is translated on the sequence diagram to the method TryPayment of object payment calling the method GetItemPrice of object item. Figure 13 depicts an identified state machine with its states and state transitions. For example, line 20 contains /status!=200 AND (Initialized), which is translated on the state diagram to the transition from the state Initialized to some other state under condition status!=200. Figure 14 consists of system's use cases, their relationships and usage of use cases by actors. For example, line 4 contains TryPayment…^FinanceSector which is translated on the use case diagram to the actor FinanceSector which uses TryPayment use case.

The AFD Tool, even though created as a simple instrument for using AFD and not as a full-fledged IDE, demonstrates how AFD and integral modeling can ease managing of architectural descriptions. Creating an integral model that can be separated into views by color-coding viewpoints or hiding them, if necessary, can help comprehend the complexities of the modeled system. At the same time, having the integral model available all the time facilitates some inconsistency management activities. Moreover, the ability to translate an integral model into appropriate UML diagrams makes adoption of AFD and its possible integration with other methodologies more feasible.

VI. EVALUATION OF AFD IN CONTEXT OF OTHER ARCHITECTURAL LANGUAGES

The topic of architectural languages has been of great interest to the software architecture community and the number of languages has been increasing swiftly [2]. In the 2000s, Medvidovic et al.. provided a classification and comparison framework for architectural languages. The framework defines component, connector, architectural configurations, and tool support features, some of which are required and some can be optionally provided by languages [16]. In 2015. Lago et al. proposed a framework of language requirements [17] based on Malavolta et al. 's survey, which was conducted on 48 practitioners from 40 IT companies and aimed to understand practitioners' needs from architectural languages [18]. In 2018. Ozkaya used the requirements defined by Lago et al. and decomposed them into sub requirements according to Lago et al. and other seminal software architecture publications. Ozkaya analyzed existing architectural languages that were determined by Malavolta et al. with an aim to aid new architectural language developers in comparing existing languages and determining their weaknesses and strengths in terms of support for a number of requirements [2]. The requirements defined by Ozkaya are divided in three groups: language definition, language features, and tool support, which are presented in Table 6. In the remainder of this section, AFD will be critically evaluated in comparison with 124 other architectural languages, group by group, in accordance with each requirement defined.

Language definition requirement group consists of a notation set, nonfunctional requirements, and formal semantic requirements. The notation set of an architectural language can be either textual or visual, either of which is preferred by practitioners depending on their experience and needs. The AFD is an architectural language that uses a textual notation set to create an architectural description of a system. The AFD textual notation set is defined by its syntax rules. The textual notation set is used by 40% of architectural languages.

Nonfunctional requirements describe the quality requirements of a software system, such as performance and security requirements. The AFD provides the specification of nonfunctional requirements by informal notation, which is covered by the resource viewpoint. Nonfunctional requirements are supported by 21% of architectural languages, and

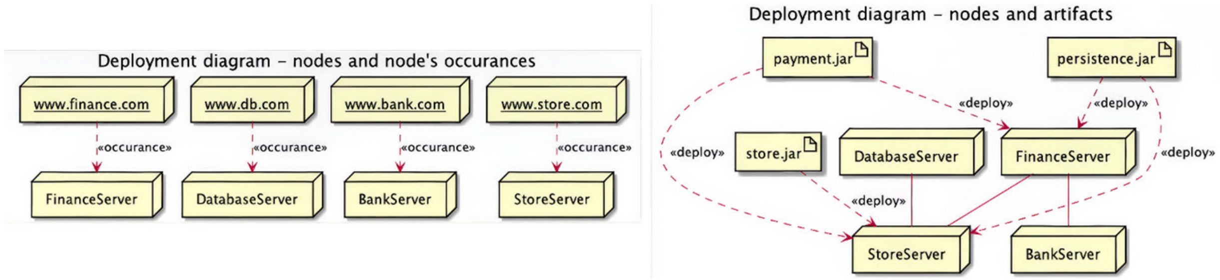

FIGURE 10. Generated UML deployment diagram for Retail payment process example.

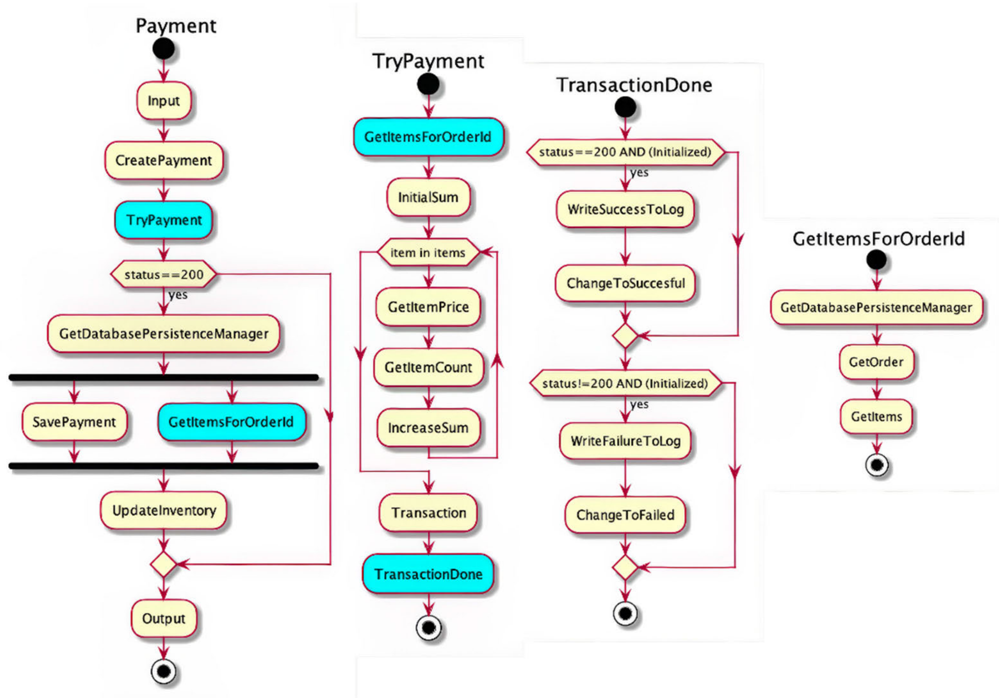

FIGURE 11. Generated UML activity diagrams for retail payment process example.

the specification of nonfunctional requirements by informal notation is supported by 13% of architectural languages.

Semantics of architectural language are defined either formally or informally. The formal semantics of an architectural language are defined by mathematically based formal methods, whereas the informal semantics of an architectural language are defined in plain English. The semantics of AFD is informally defined by the semantic rules. Informal semantics are supported by 52% of the architectural languages.

Language features requirement group consists of multiple viewpoints, extensibility, customization, and programming framework requirements. The multiple viewpoints considered by Ozkaya are the Logical, Information, Physical, Deployment, Behavior, Concurrency, Development, and Operational viewpoints [2]. Ozkaya viewpoints are mapped to example viewpoints defined by the AFD in Table 3. Example viewpoints which AFD provides are the Functional, Execution, Information, Implementation, Data state, Component, Deployment, Context, and Resource viewpoints. The example viewpoints are functional, component, context, information, data state, and execution are provided by 47% of architectural languages, while the deployment viewpoint is provided by only 15% of architectural languages.

Extensibility and customization are concerned with the ability to extend a language according to the requirements of interest. The extension of a language can be syntax or semantic. A syntax extension includes the ability to add new, modify, or remove existing architectural elements without changing the semantics of a language. Syntax extension can be achieved by introducing new, modifying, or removing the existing syntax rules of a language. Semantic extension includes the ability to add new viewpoints, new nonfunctional properties, interaction protocols, and connectors. Semantic extension can be achieved by introducing new syntax rules, and semantic rules into a language. AFD syntax can be extended by inheriting existing rules and adding new language rules. Extending the AFD syntax promotes the addition of new architectural elements, modifying existing or

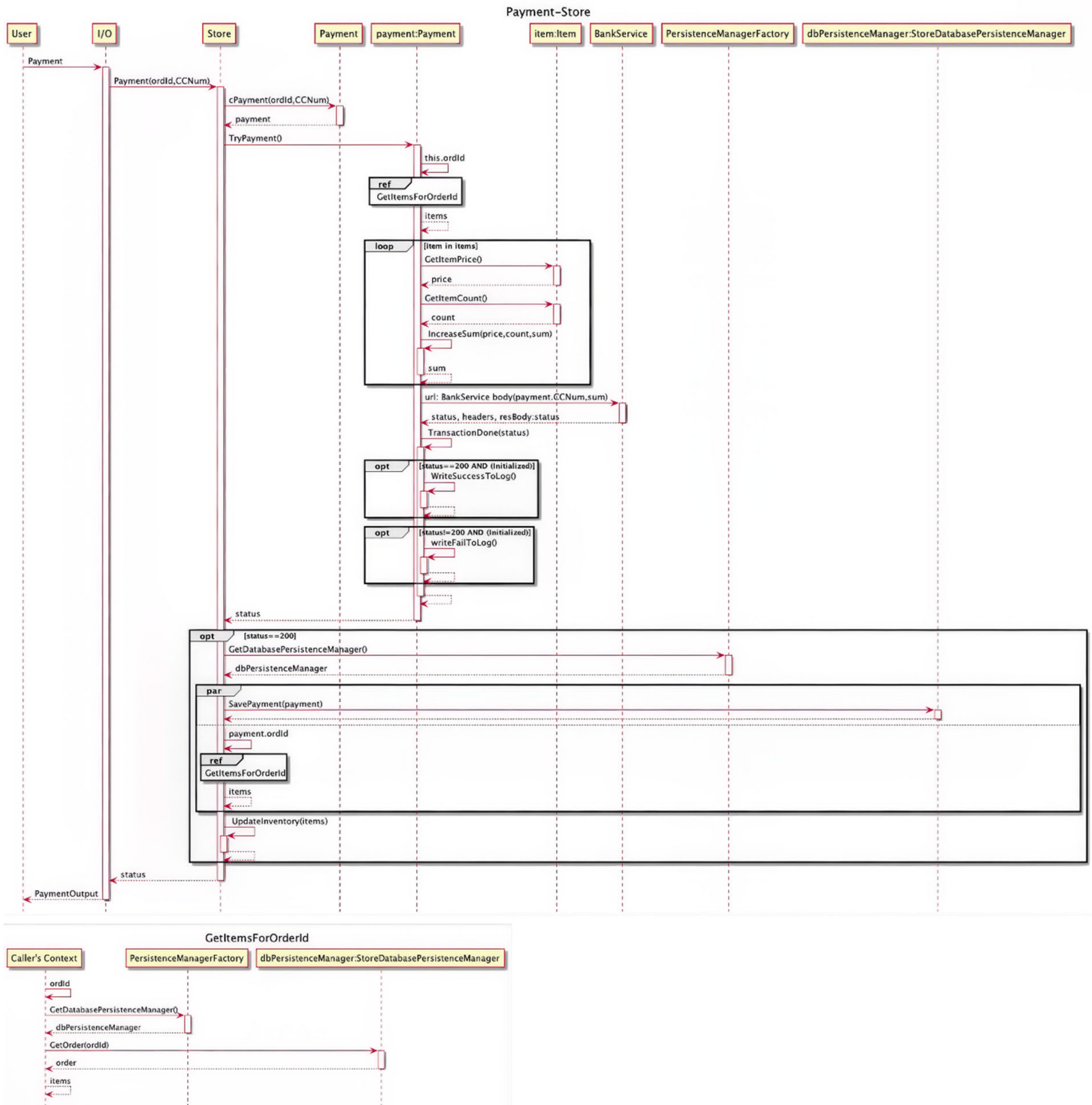

FIGURE 12. Generated UML sequence diagrams for retail payment process example.

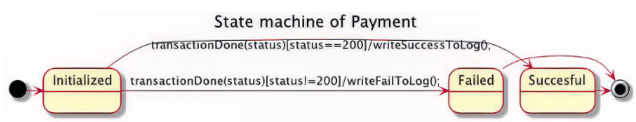

FIGURE 13. Generated UML state diagram for retail payment process example.

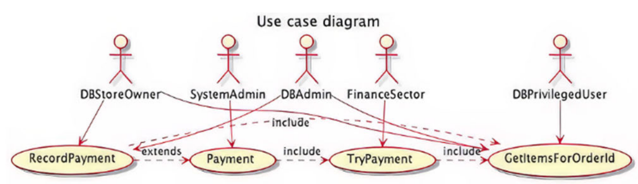

FIGURE 14. Generated UML use case diagram for retail payment process example.

removing undesired architectural elements without changing the AFD semantics. AFD semantics can be extended by inheriting existing rules, adding new language rules, writing new semantic rules, and using the resource viewpoint. Extending AFD semantics promotes the addition of new viewpoints, the creation of architectural elements inside new viewpoints, and new nonfunctional elements inside the resource viewpoint. Extensibility and customization are provided by 16% of architectural languages, where almost none of the architectural languages use language inheritance as a technique for syntax and semantic extension and customization.

TABLE 6. Requirements for architectural languages defined by Ozkaya [2] and their support by AFD. 124 architectural languages referred to in the table were determined by Malavolta et al.

Requirement Group	Requirement	Sub-requirements	Provided by AFD	Technique used by AFD	Percentage and number of languages that provide a requirement	Percentage and number of languages that provide a sub-requirement or requirement using the same technique as AFD[ab]
Language definition	Notation set	Textual, Visual	Textual	No specific technique	100% (124)	Textual 40% (49)
	Nonfunctional requirements	-	Nonfunctional requirements	Informal notation	21% (26)	Nonfunctional requirements 13% (16)
	Formal semantics	-	No	No specific technique	48% (60)	N/A
Language features	Multiple viewpoints	Logical, Information, Physical, Deployment, Behavior, Concurrency, Development, Operational	Functional, Execution, Information, Implementation, Data state, Component, Deployment, Context, Resource	No specific technique	91% (113)	Functional +Component +Context 91% (113) +Information +Data state 77% (96) +Execution 47% (58) Deployment 15% (19)
	Extensibility and customization	Syntax extension, Semantic extension	Syntax extension Semantic extension	Language inheritance	16% (20)	Syntax extension 0.01% (1) Semantic extension 0% (0)
	Programming framework	Modeling editor, Software code generation	Modeling editor	No specific technique	56% (70)	Modeling editor 56% (70)
Tool support	Automated analysis	Consistency, Completeness, Correctness, Compatibility	Consistency	User defined property: Boolean Logic	47% (58)	Consistency (all techniques) 19.3% (24)
	Versioning	-	No	No specific technique	15% (18)	N/A
	Collaboration	Synchronous, Asynchronous	No	No specific technique	8% (10)	N/A
	Knowledge management	-	Knowledge management	No specific technique	23% (29)	Knowledge management 23% (29)
	Software architecture-centric design	-	Software architecture-centric design	Generating UML diagrams	60% (75)	No data
	Large-view management	-	Large-view management	Composite components	56% (70)	Large-view management 35% (43)

[a] The name of the requirement/sub-requirement is written before the percentage and number of architectural languages.
[b] In case a specific technique is not used by AFD, the depicted percentage and number of architectural languages do not consider technique.

Programming framework supports architectural languages in their utilization in the software development process. The programming framework consists of a modeling editor that enables the creation of views that architectural language supports, and can optionally generate software implementation code. Specifying, modifying, and maintaining the architectural description in AFD is enabled by the AFD Tool, which serves as an AFD modeling editor. The programming framework is provided by 56% of architectural languages, all of which provide modeling editor for creating architectural descriptions.

Tool support requirement group consists of automated analysis, versioning, collaboration, knowledge management, software architecture-centric design, and large-view management requirements. Automated analysis is considered as the automated checking for the following analysis goals:

TABLE 7. Decomposition levels covered by the annotated functional decomposition language rules – part 1.

No.	Rule	1st level - Decomposition	2nd level - Control flow	3rd level - Data flow	4th level - Reusage	5th level - Resource flow
1	Function ::= FunctionDefinition FunctionDecompositionEntry FunctionList FunctionDecompositionExit | FunctionDefinition;	X				
2	FunctionDecompositionEntry ::= INDENT;	X				
3	FunctionDecompositionExit ::= DEDENT;	X				
4	FunctionList ::= FunctionList Function | Function;	X				
5	FunctionDefinition ::= FunctionDefinitionWithoutNewLine NEWLINE;	X				
6	FunctionDefinitionWithoutNewLine ::= FunctionPrefix FunctionName DataFlows ImplementationFlows Resources Condition Roles Node;	X	X	X	X	X
7	FunctionPrefix ::= NUMBER SPACE | EXECUTION_TYPE SPACE | CONDITION_TYPE SPACE | NUMBER EXECUTION_TYPE SPACE | NUMBER CONDITION_TYPE SPACE | EXECUTION_TYPE CONDITION_TYPE SPACE | NUMBER EXECUTION_TYPE CONDITION_TYPE SPACE | ;		X			
8	FunctionName ::= NAME | NAME HASH | HASH NAME;				X	
9	Condition ::= SPACE SLASH BoolExpression | SPACE SLASH LOOP_CONDITION | ;		X			
10	BoolExpression ::= BoolOperand | BoolOperand Separation BOOL_OPERATOR Separation BoolExpression | LBRACE BoolExpression RBRACE | LBRACE BoolExpression RBRACE Separation BOOL_OPERATOR Separation BoolExpression;		X			
11	BoolOperand ::= NAME RelationalOperation NAME | NAME RelationalOperation Constant | Constant RelationalOperation NAME | StateDefinition | Constant;		X			
12	RelationalOperation ::= RELATIONAL_OPERATION | DIRECTION;		X			
13	Separation ::= SPACE Separation | SPACE;					
14	Roles ::= ROLES_START RoleList | ;					X
15	RoleList ::= RoleList COMMA Role | Role;					X

consistency, completeness, correctness, and compatibility. Consistency is concerned with architectural elements that overlap in different viewpoints and that do not have satisfactory joint description; therefore, it is concerned with

TABLE 8. Decomposition levels covered by the annotated functional decomposition language rules – part 2.

No.	Rule	1st level - Decomposition	2nd level - Control flow	3rd leve - Data flow	4th level - Reusage	5th level - Resource flow
16	Role ::= NAME;					X
17	Constant ::= NUMBER | STRING_CONSTANT | BOOLEAN_CONSTANT;					
18	DataFlows ::= LBRACE DataFlowList RBRACE | ;			X		
19	ImplementationFlows ::= LSBRACE ImplementationFlowList RSBRACE | ;					X
20	DataFlowList ::= DataFlowList COMMA DataFlow | DataFlow;			X		
21	DataFlowType ::= LCBRACE RCBRACE DataFlowTypeHierarchy Component | DataFlowTypeHierarchy Component;					X
22	DataFlowTypeHierarchy ::= DataFlowTypeElem INHERITANCE LBRACE DataFlowTypeHierarchies RBRACE | DataFlowTypeElem INHERITANCE DataFlowTypeHierarchy | DataFlowTypeElem;					X
23	DataFlowTypeHierarchies ::= DataFlowTypeHierarchies COMMA DataFlowTypeHierarchy | DataFlowTypeHierarchy;					X
24	DataFlowTypeElem ::= NAME;					X
25	DataFlow ::= DIRECTION NAME State | DIRECTION NAME State COLON DataFlowType | DIRECTION NAME State EQUAL Constant | DIRECTION NAME State COLON DataFlowType EQUAL Constant;			X		X
26	State ::= StateDefinition | ;					X
27	StateDefinition ::= LBRACE NAME RBRACE | LBRACE NAME Entry RBRACE | LBRACE NAME Exit RBRACE | LBRACE NAME Entry Exit RBRACE;					X
28	Entry ::= VERTICAL_BAR ENTRY SPACE FunctionDefinitionWithoutNewLine;					X
29	Exit ::= VERTICAL_BAR EXIT SPACE FunctionDefinitionWithoutNewLine;					X
30	ImplementationFlowList ::= ImplementationFlowList COMMA ImplementationFlow | ImplementationFlow;					X

any contradictions between architectural elements from different viewpoints. Completeness can determine whether the architectural description satisfies all the defined system requirements. Correctness can indicate whether an

TABLE 9. Decomposition levels covered by the annotated functional decomposition language rules – part 3.

No.	Rule	1st level - Decomposition	2nd level - Control flow	3rd leve - Data flow	4th level - Reusage	5th level - Resource flow
31	ImplementationFlow ::= NAME COLON NAME Component Implementation;					X
32	Implementation ::= COLON NAME | ;					X
33	Component ::= LBRACE ArtifactName COLON ComponentName RBRACE | ;					X
34	ArtifactName ::= NAME | ;					X
35	ComponentName ::= NAME | ;					X
36	Resources ::= LCBRACE ResourceList RCBRACE | ;					X
37	ResourceList ::= ResourceList COMMA Resource | Resource;					X
38	Resource ::= ResourceDefinition Node;					X
39	ResourceDefinition ::= NAME HYPHEN NAME ResourceType | NAME ResourceType;					X
40	ResourceType ::= COLON NAME | ;					X
41	Node ::= AT NAME | AT COLON NAME | AT NAME COLON NAME | ;					X

architectural description satisfies the desired system properties defined by stakeholders, such as well-definedness rules, deadlock, and race conditions. Compatibility can determine whether architectural descriptions match any architectural style or design guideline. The AFD Tool provides automated analysis with the goal of consistency in a way that the AFD tool automatically detects inconsistencies through its semantic rules, which rely on Boolean Logic, and delegates resolution of inconsistencies to a stakeholder. Automated analysis is provided by 47% of architectural languages, but none of the architectural languages supporting automated analysis considers all four goals of analysis. Of all the architectural languages, 19.3% provide an automated analysis with consistency as a goal.

Versioning is considered with keeping and accessing architectural elements of architectural desciption. Different versions of an architectural element can be stored in a repository and later accessed and reused as part of the architectural description. Versioning is not provided by the AFD Tool. Versioning is provided by 15% of the architectural languages.

Collaboration with stakeholders reduces the time needed for the creation of architectural descriptions and enhances their quality. Collaboration can be either synchronous or asynchronous. Synchronous collaboration requires stakeholders to work on architectural descriptions at the same time, whereas asynchronous collaboration allows stakeholders to work on architectural descriptions at different times that best suit their schedule. Collaboration is not provided by the AFD Tool. Collaboration is provided by 8% of all architectural languages, all of which provide asynchronous collaboration.

Knowledge management is considered providing and sharing knowledge on architectural language with practitioners.

```
Algorithm Build Class Diagram - Part 1
procedure BuildClassDiagram(functionality)
    BuildFunctionality(functionality)

procedure BuildFunctionality(functionality)
    if ExecutorExists(functionality) = true then
        executorClassName ← ConcludeClassName(Executor(functionality))
        executorClass ← AddClass(executorClassName)
    for each dataFlow in DataFlows(functionality) do
        dataFlowPartClass ← null
        for each dataFlowPart in DataFlowParts(dataFlow) do
            previousDataFlowPartClass ← dataFlowPartClass
            dataFlowPartClassName ← ConcludeClassName(dataFlowPart)
            if dataFlowPartClassName = null then
                continue
            dataFlowPartClass ← AddClass(dataFlowPartClassName)
            if executorClassName ≠ null AND dataFlowPartClassName ≠ null AND dataFlowPartClassName ≠ executorClassName then
                AddDependency(executorClassName, dataFlowPartClassName)
            if previousClass ≠ null then
                fieldName ← Name(dataFlowPart)
                AddField(previousClass, fieldName)
                if classNode ≠ null then
                    AddAssociation(previousDataFlowPartClass, dataFlowPartClass)
        dataFlowClassName ← ConcludeClassName(dataFlow)
        AddSuperClasses(dataFlowClassName)
    if ExecutorExists(functionality) = true then
        AddMethod(functionality)
    for each subFunctionality in GetSubfunctionalities(functionality) do
        BuildFunctionality(subFunctionality)
```

```
Algorithm Build Class Diagram - Part 2
procedure AddMethod(functionality)
    class ← GetClass(functionality)
    isStatic ← ExecutorIsClass(Executor(functionality))
    methodName ← MethodName(Executor(functionality))
    method ← AddMethod(class, methodName, isStatic, returnClass)
    for each dataFlow in DataFlows(functionality) do
        if IsInput(dataFlow) then
            AddArgument(method,dataFlow)
        if IsOutput(dataFlow) then
            AddReturnType(method,dataFlow)

procedure AddSuperClasses(dataFlowClassName)
    for each inheritedClassName in InheritedDataFlowClassNames(dataFlowClassName) do
        class ← AddClass(dataFlowClassName)
        superClass ← AddClass(inheritedClassName)
        AddSuperClass(class, superClass)
        AddSuperClasses(inheritedClassName)
```

FIGURE 15. A pseudo code of an algorithm for generation of UML class diagrams.

```
Algorithm Build Component Diagram
procedure BuildComponentDiagram(functionality)
    BuildFunctionality(functionality)

procedure BuildFunctionality(functionality)
    executor ← Executor(functionality)
    executorComponent ← null
    if executor ≠ null then
        if ComponentDefined(executor) then
            componentPartNames ← GetComponentPartNames(executor)
            artifactName ← GetArtifactName(executor)
            executorComponent ← AddComponentsAndArtifact(componentPartNames, artifactName)
        else
            componentName ← ConcludeComponentName(executor)
            executorComponent ← AddComponent(componentName)
    for each dataFlow in DataFlows(functionality) do
        dataFlowComponent ← null
        if ComponentDefined(dataFlow) then
            componentPartNames ← GetComponentPartNames(dataFlow)
            artifactName ← GetArtifactName(dataFlow)
            dataFlowComponent ← AddComponentsAndArtifact(componentPartNames, artifactName)
        else
            componentName ← ConcludeComponentName(dataFlow)
            dataFlowComponent ← AddComponent(componentName)
        if executorComponent ≠ null AND dataFlowComponent ≠ null AND executorComponent ≠ dataFlowComponent then
            AddDependency(executorComponent, dataFlowComponent)
    if executorComponent ≠ null then
        executorComponentInSuperFunc ← GetExecutorComponentInSuperFunc(functionality)
        if executorComponentInSuperFunc ≠ null AND executorComponentInSuperFunc ≠ executorComponent then
            AddDependency(executorComponentInSuperFunc, executorComponent)
    for each subFunctionality in GetSubfunctionalities(functionality) do
        BuildFunctionality(subFunctionality)
```

FIGURE 16. A pseudo code of an algorithm for generation of UML component diagram.

Knowledge about architectural languages is usually shared by a website that provides materials that can be used by practitioners such as tutorials, user manuals, publications, tools, and others, and directs them in any discussion platform such as forums and user groups. Knowledge of AFD and AFD Tool is managed on a website where an introduction to AFD and its tool, publications, and contact information are provided. Knowledge management is provided by 23% of the architectural languages.

Software architecture-centric design is considered by integrating software architecture processes in other stages of the software development process, such as the specification of requirements, low-level software design, and software implementation. Software architecture-centric design in some architectural languages is provided by their tool, which enables functionalities such as automated analysis, automatic generation of low-level software design and implementation code, reusing software architecture via repositories, and integrating software architecture specifications with requirements specification. Software architecture-centric design is provided in the AFD by integrating the AFD Tool as part of a software development process. After the system requirements are defined as the result of the first phase of the software development process, the AFD Tool can be used to create an architectural description of the system. After an architectural description is created, the AFD Tool can generate UML diagrams as a visual representation of the software architecture. UML is used to visually represent architectural description as it is popular among practitioners for modeling software architectures from different viewpoints [1]. After creating architectural description and optionally generating UML diagrams, software implementation is

```
Algorithm Build Deployment Diagram - Part 1
procedure BuildDeploymentDiagram(functionality)
    BuildFunctionality(functionality)

procedure BuildFunctionality(functionality)
    nodeName ← LogicalNodeName(functionality)
    nodeOccuranceName ← PhysicalNodeName(functionality)
    node ← null
    if nodeName ≠ null then
        node ← AddNode(nodeName)
    if nodeOccuranceName ≠ null then
        nodeOccurance ← AddNodeOccurance(nodeOccuranceName)
    nodeInSuperFunc ← GetNodeInSuperFunc(functionality)
    if nodeName ≠ null then
        if nodeOccuranceName ≠ null then
            AddOccurance(node, nodeOccurance)
        artifacts ← GetArtifactsUntilNextNode(functionality)
        for each artifact in artifacts do
            AddArtifactDeployment(node, artifact)
        if nodeInSuperFunc ≠ null AND nodeInSuperFunc ≠ node then
            AddCommunication(node, nodeInSuperFunc)
    nodesForResources ← GetNodesForResources(functionality)
    for each nodeForR in nodesForResources do
        nodeNameForResource ← LogicalNodeName(nodeForR)
        nodeOccuranceNameForResource ← PhysicalNodeName(nodeForR)
        nodeForResource ← null
        if nodeNameForResource ≠ null then
            nodeForResource ← AddNode(nodeNameForResource)
            if nodeOccuranceNameForResource ≠ null then
                nodeOccuranceForResource ← AddNodeOccurance(nodeOccuranceNameForResource)
                AddOccurance(nodeForResource, nodeOccuranceForResource)
        if nodeName ≠ null then
            AddCommunication(node, nodeForResource)
        else if nodeInSuperFunc ≠ null then
            AddCommunication(node, nodeInSuperFunc)
    for each subFunctionality in GetSubfunctionalities(functionality) do
        BuildFunctionality(subFunctionality)
```

```
Algorithm Build Deployment Diagram - Part 2
procedure GetArtifactsUntilNextNode(functionality)
    nodeName ← LogicalNodeName(functionality)
    artifacts ← []
    GetArtifactsUntilNextNode(functionality, artifacts, nodeName, false)
    return artifacts

procedure GetArtifactsUntilNextNode(functionality, artifacts, currentNodeName, stop)
    executor ← Executor(functionality)
    if executor ≠ null then
        artifactName ← ConcludeArtifactName(executor)
        if artifactName ≠ null then
            artifact ← AddArtifact(artifactName)
            AddNewArtifactToSet(artifacts, artifact)
    for each dataFlow in DataFlows(functionality) do
        artifactName ← ConcludeArtifactName(dataFlow)
        if artifactName ≠ null then
            artifact ← AddArtifact(artifactName)
            AddNewArtifactToSet(artifacts, artifact)
    if stop = false then
        for each subFunctionality in GetSubfunctionalities(functionality) do
            nodeName ← LogicalNodeName(functionality)
            if nodeName = null OR nodeName = currentNodeName then
                GetArtifactsUntilNextNode(subFunctionality, artifacts, currentNodeName, false)
            else if nodeName ≠ null AND nodeName ≠ currentNodeName then
                GetArtifactsUntilNextNode(subFunctionality, artifacts, currentNodeName, true)
```

FIGURE 17. A pseudo code of an algorithm for generation of UML deployment diagram.

```
Algorithm Build Activity Diagram - Part 1
procedure BuildActivityDiagram(functionality)
    AddBegin
    BuildFunctionality(functionality)
    AddEnd

procedure BuildFunctionality(functionality)
    BuildOnFunctionalityEnter(functionality)
    for each subFunctionality in SubFunctionalities(functionality) do
        BuildFunctionality(subFunctionality)
    BuildOnFunctionalityExit(functionality)
```

```
Algorithm Build Activity Diagram - Part 2
procedure BuildOnFunctionalityEnter(functionality)
    if Parallel(functionality) = true then
        OpenFork
    if Loop(functionality) = true then
        OpenLoop(functionality)
        AddActivityOrAction(functionality, "loop")
        if ExecutorExists(functionality) = true AND SubfunctionalityExists(functionality) = true then
            BuildActivityDiagram(functionality)
    else if Optional(functionality) = true then
        OpenIf(functionality)
        AddActivityOrAction(functionality, "optional")
        if ExecutorExists(functionality) = true AND SubfunctionalityExists(functionality) = true then
            BuildActivityDiagram(functionality)
    else if Standard(functionality) = true then
        if Reusable(functionality) = true then
            AddActivityOrAction(functionality)
            if Referenced(functionality) = true AND OneReusableFunctionalityWithSubfunctionalitiesExists(functionality) = true then
                BuildActivityDiagram(functionality)
        else
            if ExecutorExists(functionality) = false AND SubfunctionalityExists(functionality) = false then
                return
            AddActivityOrAction(functionality, "standard")
```

```
Algorithm Build Activity Diagram - Part 3
procedure BuildOnFunctionalityExit(functionality)
    if Loop(functionality) then
        CloseLoop
    else if Optional(functionality) then
        CloseIf
    else if Standard(functionality) then
        if ExecutorExists(functionality) = false AND SubfunctionalityExists(functionality) = false then
            return
    if Parallel(functionality) then
        CloseFork
```

FIGURE 18. A pseudo code of an algorithm for generation of UML activity diagrams.

performed as the next phase of the software development process.

Large-view management involves techniques for representing large and complex systems in an understandable way in a view that can be easily understood and analyzed. While multiple-viewpoint support is crucial for representing different aspects of a software system, architectural languages should also support the specification of large views.

Algorithm Build Sequence Diagrams - Part 1

```
procedure BuildSequenceDiagrams(topLevelFunctionality)
    BuildSequenceDiagram(topLevelFunctionality)
    for each functionality in GetPolymorphFunctionalities do
        EndBuildPolymorph(functionality)

procedure BuildSequenceDiagram(functionality)
    if MethodIsRedifined(functionality) = false then
        BuildRegularSequenceDiagram(functionality)
    else
        BuildStubSequenceDiagram(functionality)
        BuildPolymorph(functionality)

procedure BuildRegularSequenceDiagram(functionality)
    diagram ← BeginDiagram
    BuildFunctionality(diagram, functionality, functionality)
    EndDiagram(diagram)

procedure BuildFunctionality(diagram, functionality, currentTopFunctionality)
    BuildOnFunctionalityEnter(diagram, functionality, currentTopFunctionality)
    if Reusable(functionality) = false OR functionality = currentTopFunctionality then
        for each subFunctionality in SubFunctionalities(functionality) do
            BuildFunctionality(diagram, subFunctionality, currentTopFunctionality)
    BuildOnFunctionalityExit(diagram,functionality, currentTopFunctionality)
```

Algorithm Build Sequence Diagrams - Part 2

```
procedure BuildOnFunctionalityEnter(diagram, functionality, currentTopFunctionality)
    if Parallel(functionality) = true then
        previous ← PreviousFunctionality(functionality)
        if previous ≠ null AND Parallel(previous) = true then
            AddParallelOperand(diagram)
        else
            OpenParallelFragment(diagram)
    if (executor ≠ null OR SubfunctionalityExists(functionality) = true OR Reusable(functionality) = true) AND functionality ≠ currentTopFunctionality then
        if Loop(functionality) = true then
            OpenLoopFragment(diagram)
        else if Optional(functionality) = true then
            OpenOptFragment(diagram)
        BuildMessageOnFunctionalityEnter(diagram,functionality,currentTopFunctionality)

procedure BuildOnFunctionalityExit(diagram, functionality, currentTopFunctionality)
    BuildMessageOnFunctionalityExit(diagram, functionality, currentTopFunctionality)
    if (executor ≠ null OR SubfunctionalityExists(functionality) = true OR Reusable(functionality) = true) AND functionality ≠ currentTopFunctionality then
        if CallLoop functionality = true OR Optional(functionality) = true then
            CloseFragment(diagram)
        if Parallel(functionality) = true then
            succeeding ← SucceedingFunctionality(functionality)
            if succeeding = null OR Parallel(succeeding) = false then
                CloseFragment(diagram)
```

Algorithm Build Sequence Diagrams - Part 3

```
procedure BuildStubSequenceDiagram(functionality)
    diagram ← BeginDiagram
    className ← ConcludeClassName(Executor(functionality))
    class ← GetClass(className)
    classNameTop ← GetName( GetTopSuperClassWithMethod(functionality))
    if className = classNameTop then
        return
    sendMessage ← CreateSendMessage(functionality)
    replyMessage ← CreateReplyMessage(functionality)
    lifelineForClass ← GetLifeline(functionality)
    lifelineForTopClass ← GetLifelineForTopSuperClassWithMethod(functionality)
    AddFoundMessage(diagram, lifelineForClass, sendMessage)
    AddMessage(diagram, lifelineForClass, lifelineForTopClass, sendMessage)
    AddMessageReply(diagram, lifelineForTopClass, lifelineForClass, replyMessage)
    AddLostMessage(diagram, lifelineForClass, replyMessage)
    EndDiagram(diagram)
```

Algorithm Build Sequence Diagrams - Part 4

```
procedure BuildPolymorph(functionality)
    classes ← GetClassesThatContainAndDoNotRedifineMethod(functionality)
    conditionInstanceOf ← ""
    for each class in classes do
        AddToCondition(conditionInstanceOf,class)
    if PolymorphDiagramExists(functionality) = false then
        diagram ← BeginDiagram(functionality)
        lifelineForTopClass ← GetLifelineForTopSuperClassWithMethod(functionality)
        sendMessage ← CreateSendMessage(functionality)
        AddFoundMessage(diagram, lifelineForTopClass, sendMessage)
        OpenAltFragment(diagram,functionality)
        AddAlternative(diagram,functionality,conditionInstanceOf)
    else
        AddAlternative(diagram,functionality,conditionInstanceOf)
    BuildFunctionality(diagram,functionality,functionality)

procedure EndBuildPolymorph(functionality)
    diagram ← GetDiagram(functionality)
    CloseFragment(diagram)
    lifelineForTopClass ← GetLifelineForTopSuperClassWithMethod(functionality)
    replyMessage ← CreateReplyMessage(functionality)
    AddLostMessage(diagram,lifelineForTopClass,replyMessage)
    EndDiagram(diagram)
```

FIGURE 19. A pseudo code of an algorithm for generation of UML sequence diagrams.

Large view management can be performed in architectural languages using different techniques: composite components, composite connector structures, inheritance, composite behaviors, and aspect-oriented specifications. Composite components handle the scalability of a system by specifying component structures in terms of other subcomponents and their interactions. Composite components are the most preferred technique for large-view management used by architectural languages. Composite connector structures enable the specification of complex interaction protocols in terms of simpler interaction mechanisms. Inheritance, as a principle defined in the object- oriented software engineering paradigm as a technique in large-view management, enables the extension of component specification, its structure, and behavior, with another component specification. Composite behaviors specify component behaviors in terms of the behaviors of existing components. Aspect-oriented specifications handle complex cross-cutting concerns, such as security, access control, and nonfunctional properties, and specify them modularly as aspects. Large-view management is provided by the AFD Tool inside the component viewpoint by specifying the components and their structural composition, and therefore using a composite components technique. Large-view management is provided by 56% of architectural

```
Algorithm Build State Diagrams - Part 1
procedure BuildStateDiagrams(topLevelFunctionality)
    BuildFunctionality(topLevelFunctionality)
    for each diagram in GetStateMachineDiagrams do
        BuildTransitionsToFinalState(diagram)
```

```
Algorithm Build State Diagrams - Part 2
procedure BuildFunctionality(functionality)
    dataFlowWithState ← GetDataFlowWithState(functionality)
    superFunctionality ← SuperFunctionality(functionality)
    if HasStateInCondition(functionality) = false AND dataFlowWithState ≠ null AND IsOutput(dataFlowWithState) = true then
        stateMachineDiagram ← GetStateMachineDiagram(dataFlowWithState)
        if NOT Initialized(stateMachineDiagram) then
            AddInitialAndFinalStates(stateMachineDiagram)
        stateName ← GetStateName(dataFlowWithState)
        newState ← AddState(stateMachineDiagram, stateName)
        initialState ← GetInitialState(stateMachineDiagram)
        AddTransition(stateMachineDiagram, initialState, newState)
    else if HasStateInCondition(functionality) = true AND IsStateMachineFunctionality(superFunctionality) then
        firstDataFlow ← FirstDataFlow(functionality)
        stateMachineDiagram ← GetStateMachineDiagram(dataFlowWithState)
        if NOT Initialized(stateMachineDiagram) then
            AddInitialAndFinalStates(stateMachineDiagram)
        stateNameFrom ← GetStateNameFromCondition(functionality)
        stateNameTo ← GetStateNameFromLastSubfunctionality(functionality)
        event ← CreateTransitionEvent(superFunctionality)
        condition ← GetConditionWithoutStates(functionality)
        actions ← GetTransitionActions(functionality)
        stateFrom ← AddState(stateMachineDiagram, stateNameFrom)
        stateTo ← AddState(stateMachineDiagram, stateNameTo)
        AddTransition(stateMachineDiagram, stateNameFrom, stateNameTo, event, condition, actions)
    for each subFunctionality in SubFunctionalities(functionality) do
        BuildFunctionality(subFunctionality)
```

```
Algorithm Build State Diagrams - Part 3
function GetTransitionActions(functionality)
    actions ← ∅
    for each subFunctionality in SubFunctionalities(functionality) do
        executor ← Executor(subFunctionality)
        if executor ≠ null then
            if NOT HasStateInDataflows(subFunctionality) then
                action ← CreateAction(subFunctionality)
                AddAction(actions, action)
    return actions

procedure BuildTransitionsToFinalState(stateMachineDiagram)
    finalState ← GetFinalState(stateMachineDiagram)
    for each state in GetStatesWithoutOutputTransitions(stateMachineDiagram) do
        AddTransition(state, finalState)
```

FIGURE 20. A pseudo code of an algorithm for generation of UML state diagrams.

```
Algorithm Build Use case Diagrams
procedure BuildUseCaseDiagram(topLevelFunctionality)
    AddBegin
    BuildFunctionality(topLevelFunctionality)
    AddEnd

procedure BuildFunctionality(functionality)
    roles ← ConcludeRoles(functionality)
    hasNewRoles ← false
    for each role in roles do
        if RoleExistsInSuperFunctionalities(role) = false then
            hasNewRoles ← true
            useCaseName ← Name(functionality)
            AddAssociation(role, useCaseName)
    if hasNewRoles = true then
        superFunctionality ← GetSuperFunctionalityWithRoles
        if superFunctionality ≠ null then
            optionalExist ← OptionalExistsUpTo(functionality, superFunctionality)
            allAreExclusive ← AllAreExclusiveUpTo(functionality, superFunctionality)
            if allAreExclusive = true then
                AddGeneralization(functionality, superFunctionality)
            else if optionalExist = true then
                if IncludeExists(superFunctionality, functionality) = false then
                    AddExtends(functionality, superFunctionality)
            else
                if ExtendsExists(functionality, superFunctionality) = true then
                    RemoveExtends(functionality, superFunctionality)
                AddIncludes(superFunctionality, functionality)
    for each subFunctionality in SubFunctionalities(functionality) do
        BuildFunctionality(subFunctionality)
```

FIGURE 21. A pseudo code of an algorithm for generation of UML use case diagram.

languages, whereas composite components as a technique of large-view management is provided by 35% of architectural languages.

From the previous analysis, it can be summarized that AFD and its tool provide 9 out of 12 requirements for architectural languages, except for formal semantics, versioning, and collaboration, and provide 13 of 20 sub-requirements for architectural languages. However, in addition to supporting large-view management, which aims to increase the manageability of architectural descriptions, AFD provides integral modeling. Integral modeling, although not depicted as one of the requirements for architectural languages in Table 6, should enable easier resolution of inconsistencies, regardless of their source, which increases the manageability of architectural descriptions. According to a quantitative an qualitative evaluation presented in a previous work, AFD was perceived by two-thirds of students on an information systems course as easy to understand for use during problem-solving, more than one-third of students expressed optimism about the applicability of AFD in practice, and students who used AFD achieved higher average grades than those who used UML sequential diagrams for solving the same problems [19].

VII. CONCLUSION

Over the years, various architectural languages have been created to describe complex software systems. A large number of available languages describe architectures from multiple viewpoints, thus serving stakeholders' needs. Although using viewpoints helps in dealing with complex systems, it arises a problem of consistency among views created according to these viewpoints. The architectural language AFD described in this paper provides integral modeling as a possible solution to the problem. Integral modeling supports viewpoints and enables the creation of views side by side, thus facilitating both the detection and resolution of inconsistencies. Moreover, integral models created in the AFD can be translated into appropriate UML diagrams for further design and analysis.

Previous work and the initial experiments with AFD in an undergraduate information systems course showed positive results during exams and optimism regarding practical use. An additional evaluation conducted in this paper according to the requirements used in the literature for the evaluation of 124 architectural languages showed that AFD provides nine out of 12 requirements. Unsupported requirements encompassing formal semantics, versioning, and collaboration will be provided in upcoming versions of the AFD. Additional experiments and quantitative evaluations with examples of complex systems are planned in the future.

APPENDIX A
DECOMPOSITION LEVELS AND ANNOTATIONS

See Tables 7–9.

APPENDIX B
PSEUDO CODES OF ALGORITHMS FOR GENERATION OF UML DIAGRAMS

See Figures 15–21.

REFERENCES

[1] M. Ozkaya and F. Erata, "A survey on the practical use of UML for different software architecture viewpoints," *Inf. Softw. Technol.*, vol. 121, May 2020, Art. no. 106275, doi: 10.1016/j.infsof.2020.106275.

[2] M. Ozkaya, "The analysis of architectural languages for the needs of practitioners," *Softw., Pract. Exper.*, vol. 48, no. 5, pp. 985–1018, Jan. 2018, doi: 10.1002/spe.2561.

[3] A. Cicchetti, F. Ciccozzi, and A. Pierantonio, "Multi-view approaches for software and system modelling: A systematic literature review," *Softw. Syst. Model.*, vol. 18, no. 6, pp. 3207–3233, Dec. 2019, doi: 10.1007/s10270-018-00713-w.

[4] D. Soni, R. L. Nord, and C. Hofmeister, "Software architecture in industrial applications," in *Proc. 17th Int. Conf. Softw. Eng.*, Seattle, WA, United States, Apr. 1995, pp. 196–196.

[5] P. B. Kruchten, "The 4+1 view model of architecture," *IEEE Softw.*, vol. 12, no. 6, pp. 42–50, Nov. 1995, doi: 10.1109/52.469759.

[6] *IEEE Recommended Practice for Architectural Description for Software-Intensive Systems*, Standard 1471-2000, 2000.

[7] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting Software Architectures: Views and Beyond*, 1st ed., Boston, MA, USA: Addison-Wesley Professional, 2002.

[8] J. Garland and R. Anthony, *Large-Scale Software Architecture: A Practical Guide using UML*, 1st ed., Hoboken, NJ, USA: Wiley, 2002.

[9] N. Rozanski, and E. Woods, *Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives*, 1st ed., Upper Saddle River, NJ, USA: Addison-Wesley 2005.

[10] R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory, and Practice*, 1st ed., Hoboken, NJ, USA: Wiley, 2009.

[11] *Systems and Software Engineering—Architecture Description*, Standard ISO/IEC/IEEE 42010:2011(E), 2011.

[12] G. Spanoudakis, and A. Zisman, "Inconsistency management in software engineering: Survey and open research issues," in *Handbook of Software Engineering and Knowledge Engineering*, 1st ed. Singapore: World Scientific, 2001, ch. 15, pp. 329–380.

[13] J. M. Wing, "Computational thinking," *Commun. ACM*, vol. 49, no. 3, pp. 33–35, Mar. 2006, doi: 10.1145/1118178.1118215.

[14] L. G. Williams and C. U. Smith, "Performance evaluation of software architectures," in *Proc. 1st Int. Workshop Softw. Perform.*, Santa Fe, NM, USA, Oct. 1998, pp. 164–177.

[15] E. Jagroep, J. M. van der Werf, S. Brinkkemper, L. Blom, and R. van Vliet, "Extending software architecture views with an energy consumption perspective," *Computing*, vol. 99, no. 6, pp. 553–573, Jun. 2017, doi: 10.1007/s00607-016-0502-0.

[16] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," *IEEE Trans. Softw. Eng.*, vol. 26, no. 1, pp. 70–93, Jan. 2000, doi: 10.1109/32.825767.

[17] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, and A. Tang, "The road ahead for architectural languages," *IEEE Softw.*, vol. 32, no. 1, pp. 98–105, Jan. 2015, doi: 10.1109/MS.2014.28.

[18] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 869–891, Jun. 2013, doi: 10.1109/TSE.2012.74.

[19] S. Tubić, M. Cvetanović, Z. Radivojević, and S. Stojanović, "Annotated functional decomposition," *Comput. Appl. Eng. Educ.*, vol. 29, no. 5, pp. 1390–1402, Mar. 2021, doi: 10.1002/cae.22394.

STEFAN TUBIĆ received the B.Sc. and M.Sc. degrees in electrical and computer engineering from the School of Electrical Engineering, University of Belgrade, Serbia, in 2014 and 2016, respectively. He is currently working as a Teaching Assistant with the University of Belgrade. He teaches several courses on databases and database software tools, information systems, and computer networks. His research interests include database and information systems, data analysis, and artificial intelligence.

ZAHARIJE RADIVOJEVIĆ received the B.Sc., M.Sc., and Ph.D. degrees in electrical and computer engineering from the School of Electrical Engineering, University of Belgrade, Serbia, in 2002, 2006, and 2012, respectively. He is currently working as an Associate Professor with the University of Belgrade. He teaches several courses on computer architecture and organization, e-business infrastructure, and mobile device programming. His research interests include computer architecture and organization, concurrent and distributed programming, data analysis, simulations, and reverse engineering.

SAŠA STOJANOVIĆ received the B.Sc. and Ph.D. degrees in electrical and computer engineering from the School of Electrical Engineering, University of Belgrade, Serbia, in 2006 and 2016, respectively. He is currently working as an Assistant Professor with the University of Belgrade. He teaches several courses on embedded systems, system programming, and mobile devices programming. His research interests include artificial intelligence, software similarity, and reverse engineering.

MILOŠ CVETANOVIĆ received the B.Sc., M.Sc., and Ph.D. degrees in electrical and computer engineering from the School of Electrical Engineering, University of Belgrade, Serbia, in 2003, 2006, and 2012, respectively. He is currently working as an Associate Professor with the University of Belgrade. He teaches several courses on databases and database software tools, information systems, and e-business infrastructure. His research interests include database and information systems, artificial intelligence, big data, and reverse engineering.