Noname manuscript No.
(will be inserted by the editor)

The influence of cyclomatic complexity distribution on the understandability of xtUML models

Nenad Ukić · Josip Maras · Ljiljana Šerić

Received: date / Accepted: date

Abstract Executable software models formalize functional software requirements. This means that the understandability of software models is of paramount importance. In this paper, we investigate the influence of distribution of model cyclomatic complexity on the understandability of *Executable Translatable UML* (xtUML) models. We adapt traditional cyclomatic complexity metrics to different xtUML sub-models and present two different ways of measuring complexity distribution: *horizontal*, among elements of the same type, and *vertical*, among elements of different types. In order to test our hypothesis that cyclomatic complexity distribution influences the understandability of xtUML models, we have performed an experiment with student participants in which we have evaluated the understandability of three semantically equivalent xtUML models with different complexity distributions. Results indicate that a better distribution of cyclomatic complexity has a positive influence on model understandability.

Keywords xtUML · understandability · cyclomatic complexity · distribution

1 Introduction

Traditionally, software models are used in the initial stages of the software development process: for requirements analysis, documentation, or early design purposes. The first formal use of UML-based models in software development typically meant

N. Ukić
Ericsson Nikola Tesla d.d. Poljička cesta 39 Split 21000 Croatia
Tel.: +385-91-3655970
E-mail: nenad.ukic@ericsson.com

J. Maras
University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture in Split (FESB), R. Boskovica 32, 21 000 Split, Croatia
E-mail: Josip.Maras@fesb.hr

Lj. Šerić
University of Split, Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture in Split (FESB), R. Boskovica 32, 21 000 Split, Croatia
E-mail: Ljiljana.Seric@fesb.hr

using class models to generate the initial source code skeleton, which is then used as a base for traditional software development. The problem with such, *elaborative* approach is that the software skeleton generated from the initial model typically changes over time which makes the model deprecated.

Executable software models are changing this elaborative paradigm by making the model a central development artifact. A key feature of executable models is their ability to be executed, which implies the possibility of testing. In order for a model to become executable, it must become computationally complete, that is, in addition to the structure, the model must specify the details of application behavior [32]. The traditional iterative software development process of testing and correction can then be applied to models themselves. After a model is verified and meets all functional requirements, it is used to generate the application source code. In this *translational* process, the source code is considered as a temporary asset on the way from a model to binary form used in production. The intellectual property of such software development is not the source code, but a model of the application, as well as the translation mechanism used to generate the source code [47].

The paradigm of executable software models was introduced by *Model Driven Architecture* (MDA)[36] [22] [43] [35]. One of the oldest and most mature executable UML methodologies is *Executable Translatable UML* (xtUML) [5], a successor to the Shlaer-Mellor object-oriented methodology [55], adapted to the UML graphical notation. An open source tool – BridgePoint [42] [41] supports xtUML model development and is the main enabler of the methodology. There are, however, other methodologies and approaches to MDA. Except for xtUML, the Object Management Group (OMG) has published a *Foundational subset for executable UML models* (fUML) [38] which simplifies the UML standard and defines precise semantics for the execution of UML models. The main goal of this effort is to establish a standard language and execution semantics that will be common to all *higher level* languages, including xtUML. By applying the translational approach, other languages would only translate to fUML formalisms and reuse its execution and code generation tool chain.

There are several important implications of the translational approach used by executable software models. First, it is possible to separate functional and non-functional requirements. The model is used to specify functional requirements in the simplest possible way, while decisions about platform, language, robustness, and speed are handled in the code generator. In this way, the software model becomes independent of the platform on which the application will be executed and of the target language of the generated source code. Besides obvious reuse benefits, the separation of functional and non-functional requirements means that models can become more abstract and therefore simpler and more approachable to domain experts.

In general, cognitive simplicity and understandability of software is the result of clear abstractions, i.e. clear links between formal concepts used to formalize the software and actual domain concepts they represent [24]. Since functional and non-functional requirements can be separated, executable software models no longer have to be a compromise between simplicity and application implementation constraints. The focus of executable software models can be entirely moved to the clarity of application's functional specifications. This means that the understand-

ability of a model becomes essential. Because of this, objectively measuring model understandability is of paramount importance.

Approaches to measuring software understandability can be categorized into two groups: *i) linguistic*, which focus on the similarity between software element identifiers and domain concepts [24], and *ii) metric-based* which are concerned with the relationship between different software metrics and software understandability [18] [21].

The main contribution of this paper is a metric-based approach to measuring software understandability centered on cyclomatic complexity. Cyclomatic complexity is a metric based on the number of linearly independent execution paths through an application. Traditionally, the majority of software metrics are applied to source code, but in this case, since we are dealing with models, we adapted the existing cyclomatic complexity measures to xtUML models. We present a way to calculate the integrated cyclomatic complexity of a complete xtUML model and different ways to distribute it across a model.

Next, we propose the following hypothesis: ***(H) The understandability of xtUML models is influenced by the model's distribution of cyclomatic complexity.*** To validate this hypothesis we conducted an experiment with student participants in which we evaluated the understandability of three semantically equivalent models with different complexity distributions. Experiment results have shown that there exists a connection between the distribution of cyclomatic complexity and model understandability.

Our study is motivated with problems that often occur with executable software modeling. It is important to emphasize that merely using executable models and software visualization does not guarantee that software will be better in any aspect. In traditional software development abstractions are hidden in source code, they are often neglected and are rarely systematically assessed. In our opinion, the main benefit of executable software modeling is in the fact that it puts the quality of abstractions, a key challenge of software development, in the focus of the software development process. In other words, executable modeling has the power to expose our abstractions visually, but it does not guarantee that those abstractions are of high quality. It is very difficult to objectively measure the quality of abstractions and the quality of xtUML models. Having methods that enable developers to assign a quality metric to a certain model can help guide design decisions and, in turn increase the quality of model-driven software systems.

This paper is structured as follows: section 2 presents the background required for understanding the paper and related work relevant to our research. Section 3 describes our adaptations of cyclomatic complexity metric to different xtUML sub-models. In section 4 we present a way to calculate the integrated cyclomatic complexity of an xtUML model and the different ways of calculating complexity distribution. Section 5 describes the experiment setup, while section 6 presents the results. Finally, section 7 concludes the paper.

2 Background and related work

In this section, we describe the xtUML model and the four sub-models comprising the model. This is followed by an explanation of factors affecting software understandability, along with a description of cyclomatic complexity.

2.1 xtUML

xtUML (eXecutable and Translatable Unified Modelling Language) [30] [62] is an MDA methodology [31] and a successor to the Shlaer-Mellor object-oriented methodology [55], [5] adapted to UML graphical notation. It is a software development language that uses the graphical notation of the standard UML, but also defines precise execution and timing rules. xtUML models are semantically complete, and all information about software structure, behaviour, and processing is integrated in such a way that xtUML models can be executed. Although intended as a general-purpose language for executable software modelling, xtUML is probably best suited for embedded applications [33] [23]. In addition, as a graphical software modelling methodology, it is not an optimal choice for highly algorithmic applications. Despite its simplicity and matureness, xtUML is still not widely used in the software development community. One of the main reasons for this was the fact that the main enabler of the language, the BridgePoint tool[40], was not open source up until recently. This was also a reason why most of the executable software modeling community has gathered around OMG's fUML[38] and Alf[37] languages, which are more generic and closer to mainstream object-oriented languages, but are also less mature and more complex than xtUML.

A system designed with xtUML is composed out of four interconnected types of models: *i) component* models, which define the overall system architecture; *ii) class* models, which define concepts and relations within a component; *iii) state machine* models, which define class instance life-cycle; and the *iv) processing* model which specifies execution details. The component, class, and state machine models are graphical models, while the processing model is textual.

2.1.1 Components

The foundational building block of xtUML models are components. Each component is considered as a black-box that uses only interfaces to communicate with other components. An xtUML interface is a definition of a message set that can be used for inter-component communication. Interfaces are bidirectional and messages on a single interface can go in both directions.

Interfaces specify the types of component ports across which the actual communication is performed. Two components may only be connected across ports that are typed by the same interface, and where one component should provide the interface, while the other should require it. While interfaces are merely a specification of possible messages described with names, parameters, and directions relative to the "provider side", ports can actually contain action code that specifies actions to be taken when a certain message is received. It is important to emphasize that only incoming messages can be associated with action code.

The communication between two components can be either *synchronous* or *asynchronous*. Synchronous inter-component communication is enabled with interface operations, while asynchronous communication is achieved through interface signals. Figure 1 shows an example of an xtUML component model.

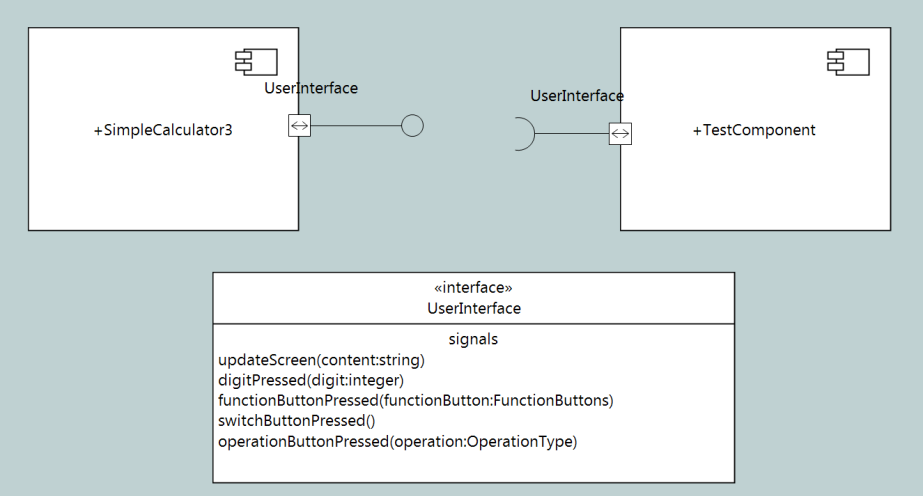

Fig. 1 An example of component model.

2.1.2 Classes

An xtUML class model describes domain concepts and the relationships between instances of those concepts. Similarly as in other object-oriented languages, a simple relation in xtUML is defined by two relation ends, each of them defined with multiplicity and conditionality flags, as well as with a phrase describing its semantics. Different combinations of multiplicity and conditionality flags specify different runtime limitations that apply to the number of instances on a relation end (see table 1)

Table 1 Different flag combinations and associated meaning.

Conditionality flag	**Multiplicity flag**	**Symbol**	**Instance number limitation**
TRUE	FALSE	0..1	at most one
FALSE	FALSE	1	exactly one
TRUE	TRUE	*	zero or more
FALSE	TRUE	1..*	at least one

Associative relations are used in cases where there is need to model details of a simple relation. Typically this includes cases when a relation has some attributes or a life-cycle of its own. In that case, a relation is represented as an associative class which can have all typical class features: a state machine, attributes, operations, or even relations towards other classes. Since an associative class is both a relation and a class, the mere existence of an associative class instance implies the existence of a pair of instances of *related classes* (non-associative classes involved in associative relation) and a link between them.

Unlike some other object-oriented (OO) languages such as C++ or Java, a generalization relation in xtUML assumes the existence of two separate instances which should be created independently and related explicitly by the user, the same

way as it is done with simple relations. In addition, an xtUML subclass does not inherit superclass operations and attributes. In the majority of OO languages, generalization is mostly used as an extension mechanism, but this is not the case with xtUML. The main purpose of such relation is to split an instance population into *complete* and *disjoint* sets of subclass instances. *Complete* sets means that there may be no other subclasses other than those stated initially, i.e. we cannot easily add new subclasses to an existing generalization relation; we can only add a new generalization relation and specify its complete and disjoint subclasses. *Disjoint* subclass instance sets imply that a single superclass instance is only related to a single subclass instance per generalization relation; a superclass is related to as many subclass instances as the number of generalization relations it defines. This semantics is very different from traditional OO semantics, but it does not assume anything about the target language. Generalization relations in xtUML do not rely on inheritance mechanism of traditional OO generalization and may be applied easily to non-OO languages as well.

2.1.3 State machines

State machine models in xtUML can only be defined within a context of a class. Typically, a state machine is used to visualize the life cycle of a class instance, and is usually composed out of one or more numbered states. Figure 2 shows an example state machine model.

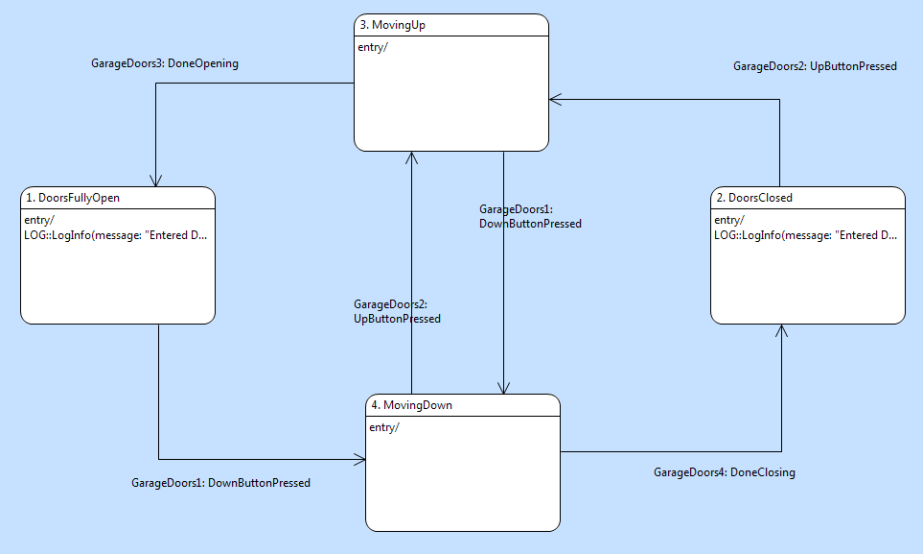

Fig. 2 Graphical view of the state machine

At a certain point in time, an instance can only be in one state, and it can move from one state to another only if there exists an explicit transition between those states. A transition is triggered by a certain event and cannot be addition-

ally guarded with extra conditions. In case where a transition does not have an associated event, the transition cannot occur.

With each transition we can associate some action code which will be executed when the instance moves from one state to another, following the transition. In addition, action code can also be associated with state entries, which means that we can execute additional action code, just after the execution of the transition code. On the other hand, action code cannot be associated with state exits nor with the fact that an instance resides in some state.

When a class instance, that has an associated state machine, is created, it immediately starts in the *initial* state, the lowest numbered state. Even though action code can be associated with state transition and state entries, in this case, even if the initial state has associated entry code, no code is executed since the instance is immediately placed in a state, without transitioning to it.

After being created, an instance waits for events that will move it to different states. If no events occur, the instance will remain in the initial state until deleted. If an event occurs, the corresponding transition action code (also called transition effect behaviour) is executed. This is followed by target state entry behaviour. Note that those two behaviours are always executed in a sequence, one after another, as a single *Request-To-Completion* (RTC) step, without any interruptions. After an event is processed, the instance resides in a new state and awaits for further events. This is repeated until the instance reaches the end of its life-cycle, which happens in the following situations:

- The instance reaches its final state. After executing the final-state entry behavior, the instance is deleted.
- The instance receives an event that it does not know how to handle; a runtime error happens. The current execution is stopped and all available information is logged for troubleshooting.
- The instance is explicitly deleted, if some other code request instance deletion.

An instance can process a certain event if its current state has exactly one outgoing transition that is triggered by that event. For other cases, the developer should specify a *State-Event matrix* (figure 3) which defines what action should be performed for any state, event combination. Unlike the graphical view of the state machine (figure 2), the state-event matrix represents a complete state machine with all possible combinations of states and events.

	GD1: DownButtonPressed	GD2: UpButtonPressed	GD3: DoneOpening	GD4: DoneClosing
DoorsFullyOpen	MovingDown	Event Ignored	Event Ignored	Can't Happen
DoorsClosed	Event Ignored	MovingUp	Can't Happen	Event Ignored
MovingUp	MovingDown	Event Ignored	DoorsFullyOpen	Can't Happen
MovingDown	Event Ignored	MovingUp	Can't Happen	DoorsClosed

Fig. 3 State-event matrix view of the state machine

Notice that for handling cases when a state is not ready to receive an event, the modeller has two options: to simply ignore the event (*Event Ignored*, figure 3), or to trigger an error handling procedure (*Can't Happen*, figure 3). When an event is ignored, nothing happens, an instance remains in its current state, ready to

receive new events. On the other hand, when an error case occurs, the execution is stopped and the error details are logged. The act of deciding which action should be taken in case an unexpected event occurs is not automated, and should be a part of the modelling decisions done by the modeler.

2.1.4 Processing code

The processing code, written in Object Action Language (OAL), describes the behavioural details of an xtUML model. It can be associated with port incoming messages, functions, class and instance-based operations, derived attributes, state machine transitions, and state entries.

```
a=4;
//If statement example
if(a%3==0)
    a=a+1;
elif(a%3==1)
    a=a-1;
else
    a=0;
end if;

//While loop example
while(a>0)
    //Do something
    a = a-1;
end while;
```

```
select many digit_set from instances of Digit;
//for loop example
for each digit in digit_set
    //instance operation invocation
    digit.doSomething();
end for;

//Port message invocation
UI::updateScreen(content:"0.001");

select any operation from instances of Operation;
//Event generation (asynchronous)
generate Operation1:EqualsPressed to operation;
```

Fig. 4 Statements affecting control flow execution in OAL language.

Code execution starts at the first statement of the action and proceeds sequentially through the succeeding lines as directed by program's control logic. The execution of an action terminates when the last statement is completed. The control-flow is affected by synchronous invocations, asynchronous event handling, conditional *if* branches and conditional loops (*while* and *for*), see figure 4.

Processing code is the only textual model in xtUML, because it is more efficient to specify processing instructions in a textual than in a visual way. Behind the scenes, processing models are represented with corresponding instances in the xtUML meta-model, similarly to component, class, or state machine models. From the perspective of execution and code generation tools, the processing code is handled similarly as other three models.

2.2 Other executable software methodologies

This section will briefly cover the other executable software model methodologies and standards.

2.2.1 Real-time Object-Oriented Modeling (ROOM) methodology

UML-RT is a UML profile defined by IBM for the design of distributed event-based applications. The profile is based on the Real-Time Object-Oriented Modeling (ROOM) methodology [51] and was implemented in the RSA-RTE [28] product, an extension of IBM's Rational Software Architect (RSA) product.

The basis of UML-RT is the notion of *capsules* that at the same time can have both the internal structure (via capsule parts) and behaviour (via a state machine). Capsules can be nested, and can communicate synchronously and asynchronously via messages that are sent and received through ports. The types of messages that a port can transmit are defined by protocols. Unlike xtUML components, capsules can be created both statically at design time and dynamically at run-time.

RSA-RTE tool allows several languages to be used to specify actions (e.g. C, C++ and Java). Model execution is only possible by translation to source code, but no model-level interpreter is available, at least not in the sense provided by the BridgePoint [42] tool.

2.2.2 Foundational Subset for Executable UML Models (FUML)

Realizing the importance of simplicity and clear semantics that are missing from UML standards, the OMG defined the semantics of a *foundational subset for executable UML models*(fUML)[38]. The main goal of this standard is to act as an intermediary language between surface subsets of UML used for modeling and computational platform languages used as the target for model execution. fUML is designed to be compact, in order to facilitate the definition of clear semantics and the implementation of execution tools. In addition, it is supposed to be easily translated from common surface subsets of UML to fUML and from fUML to common computational platform languages. However, if the feature to be translated is excluded from fUML, the *surface-to-fUML* translator has to generate a coordinated set of fUML elements that has the same effect as that feature. Then the *fUML-to-platform* translator would need to recognize the pattern generated by the surface-to-fUML generator, in order to map this back into the desired feature of the target language. Compactness can therefore conflict with ease of translation.

One of the future directions of UML is executable modelling which assumes simplicity and clear execution semantics. However, the problem with fUML is that tool support is weak and whether it will be accepted by the community.

2.2.3 Action Language for FUML (ALF)

The Action Language for Foundational UML (or *ALF*) [37] is a textual surface representation for UML modeling elements. The execution semantics for ALF are given by mapping the ALF concrete syntax to the abstract syntax of fUML. The primary goal of an action language is to act as the surface notation for specifying executable behaviors within a wider model. ALF also provides an extended notation that may be used to represent structural modelling elements. Therefore, it is possible to represent an UML model entirely using ALF. However, ALF syntax only directly covers the limited subset of UML structural modelling available in the fUML subset.

ALF is important because it is an attempt to standardize the action language for executable software models based on UML. In addition to that, a possibility to completely specify an fUML model using textual notation also helps alleviate the problem of poor tool support for resolving version conflicts in graphical models [57]. Since ALF is actually a textual surface notation for fUML models, higher level languages can be translated directly to ALF (instead to fUML). In such form, higher level models will be able to use ALF virtual machines and translate to target source code using ALF translators.

2.3 Software understandability

Software understandability or program understanding includes activities needed to obtain a general knowledge of what a software product does and how the parts work together [1]. In this section, we describe the factors influencing the understandability and related work.

2.3.1 Software understandability factors

There are several factors influencing software understandability:

- *Functional size of software* is the main factor influencing understandability. It refers to the number and complexity of use cases that the software system satisfies, and not the implementation size [3] [25] [39]. Unless we are interested in the relation between functional size and the understandability of software, comparing the understandability of software applications of various sizes is problematic.
- *Programming language* used for implementing the software system. Different implementation languages have different readability which influences source code understandability.
- *Consistency and quality of naming conventions.* As indicated by Laitinen [24] and Rajlich [48], the similarity between concepts used for software elements identifiers and domain concepts used in application specification is one of the key factors influencing the understandability of the application. The strength of mapping between those two sets of concepts reflects the consistency and intuitiveness of naming and coding conventions in the source code.
- *Modularisation approach, choice of design alternative and complexity distribution.* Different approaches to source code modularization [61] and the quality of the modularization [50] have an impact on the comprehension of the software source code. Notice that better modularization also implies better complexity distribution.

Most empirical evaluations of software understandability are based on subjective evaluation made by experts in which understandability of a number of software applications of different sizes, domains, and authors is ranked [49]. The main problem with such evaluations is precision, because the factors affecting software comprehension are confounded. In order to precisely relate source code or model understandability with some of the mentioned factors, it is necessary to eliminate all other factors, and variate the remaining factor. In our work we are focused on the relation between complexity distribution, introduced by different

modularization techniques, and model understandability. In order to evaluate this relationship, we need to minimize the effect of remaining three factors (functional size of software, programming language, and the consistency and quality of naming conventions).

2.3.2 Software understanding process and theories

The process of software comprehension usually starts by mapping domain concept names with source code identifiers relying on programmers intuition and experience. If this technique fails, a more formal, string pattern matching process is typically used. When these name-mapping techniques fail to locate the concepts, programmers typically instrument the code with various logs and execute different application features. This process of execution and trace-mapping is known as *dynamic search* or *software reconnaissance* [60] [13]. In addition to dynamic analysis, static analysis can also be performed [7]. Starting from the program or test case main function, top down control and/or data flow tracing can be used to find the relevant part of the source code.

Rajlich et. al. [48] introduced the process of *concept location* which implies a mapping between domain concepts and their code implementation. The first and the most significant phase in the concept location process is based on the similarity between names in software specification documents and identifiers used in the software implementation. In order to measure this similarity, Laitinen [24] considered that a *language* is a set of symbols with associated meanings. In this sense, every person, article, or source code has a language of its own. Languages are said to be related if they share the same symbol and its meaning. Smaller and more closely related languages are easier to understand than larger and more distantly related languages. With such definition of a language, only a relative measure of understandability makes sense.

Since these techniques are based on naming similarities, they are often not good enough at handling the problem of homonyms, synonyms, and polysemy. An additional problem in maintenance is that the vocabulary used to describe the software evolves and it may significantly differ from the initial vocabulary used for implementation. In addition to that, some domain concepts are not represented explicitly in the source code and cannot be found in this way.

There are also some more advanced approaches that are not based on naming similarities. *Latent Semantic Indexing*(LSI)[19] [12] is based on the fact that words with similar meaning appear close in the documents. The meaning of words in LSI is derived from their usage rather than from a dictionary or thesaurus. LSI has been shown to address problems of polysemy and synonymy [19] quite well which makes it a good fit for source feature/concept search problem because developers usually construct queries without precise knowledge about the target vocabulary [46].

2.3.3 A relation between software metrics and understandability

Understandability of a software artefact is a key factor in software maintainability. Problems with managing software projects and predicting maintenance efforts indicate a need for objective prediction of software maintainability. The common usage of software metrics is the creation of maintainability prediction models.

Welker [59] noticed a spiral of code degradation through maintenance and has proposed an integrated maintainability measure. The benefit of such integrated measure is the objective quantification of code degradation which could be used in software management decision support. Zhou [63] empirically investigated the relationship between 15 design metrics and the maintainability of 148 Java open source software. The results indicate that size and complexity are strong predictors of maintainability while cohesion and coupling metrics do not seem to have a significant impact on maintainability.

Nazir [34], proposed a regression model for estimating the understandability of OO software using the number of attributes, the number of associations, and the maximum depth of inheritance as metrics. He evaluated the model by correlating it with expert ratings on 28 programs, and reported the correlation of 0,948. Van Koten [58] proposed a Bayesian network maintainability prediction model for an object-oriented software system and has compared it with the prediction of traditional regression models. The model used Li and Henry's set of object-oriented metrics [26] collected from two different object-oriented systems. His results suggest that the Bayesian network model can predict maintainability more accurately than the regression-based models. Shibata *et. al.* [54] proposed a stochastic, queueing model for predicting software reliability and maintainability. The model used real software fault detection/correction data obtained from practice. Aggarwal *et. al.* [2] proposed an integrated fuzzy-logic model of understandability which takes into account the number of comments in the source code, the quality of documentation calculated through the Gunnings Fog index [16], and the similarity between the language of the specification and the one used in source code [24].

Riaz *et. al.* [49] made a systematic overview of software maintainability prediction and metrics and concluded that there is no obvious model for prediction of software maintainability. They have indicated that most of the metrics are based on size, complexity, and coupling gathered on source code level. Maintainability in most of the studies is based on experts judgment and is expressed on an ordinal scale.

2.4 Cyclomatic complexity

Cyclomatic complexity, introduced by McCabe [29], is a software metric most frequently used to estimate testing effort required to achieve complete branch coverage. It is a quantitative measure of the number of linearly independent paths through the program's source code. The main approach when calculating McCabe's cyclomatic complexity is to represent software as a *single-entry, single-exit* (SESE) control flow graph (CFG). Edges in such a graph represent the possible control flows between blocks of code that contain no control flow branches. Regardless of the number of lines of code they actually contain, such blocks are represented with a single node in the graph. The initial approach taken by McCabe actually represented each software module (subroutine) as a separate, fully connected single-entry and single-exit CFG [29]. This approach is useful to estimate unit testing effort, because each subroutine is observed independently. An alternative approach, proposed by Henderson-Sellers *et. al.* [17] proposed that software should be observed as a single CFG with a single entry and a single exit node. This approach is useful for estimating software integration testing and has

shown that a single, big SESE graph can be constructed from multiple smaller ones.

In Henderson-Sellers [17] approach multiple-entry, multiple-exit (MEME) graphs are handled in the following way: multiple entries in multiple-entry, single-exit (MESE) modules represent a module reuse mechanism and do not affect module's cyclomatic complexity. Generally, single-entry, multiple-exit (SEME) nodes typically occur in branches of *if* structures where an immediate return ("early exit") to the calling module occurs. One way of handling such cases is to add an additional, virtual node for each early exit and connect it with virtual edges to early and normal exit nodes. This means we need to add one point to cyclomatic complexity for each exit (+1 = 2 new edges - 1 new node). The formula is given with:

$$CC_{seme} = e - n + p + 1 + \sum_{j=1}^{p-1}(r_j - 1) \tag{1}$$

where r_j is the number of exit points in the graph representation of the *j-th* module and there are *p-1* such modules (subroutines). This means that modules with multiple exit points increase the overall module complexity. In addition, Henderson-Sellers noted that multiple entries can be considered as a reuse mechanism and that multiple-entry, multiple-exit (MEME) modules can be treated as modules with multiple exit points. This implies that previous equation applies to such modules as well.

Strongly connected graphs are those in which each node can reach any other node. In our case, a CFG of a single method or complete software is not strongly connected but it can be turned into one by adding a single, virtual edge from the exit to the entry node.

3 Measuring cylcomatic complexity of xtUML models

xtUML builds applications from four different types of models: component models, class models, state-machine models, and processing code. Component and class models describe the structure, while state machines and action code describe the runtime behavior of an application.

When calculating cyclomatic complexity of an application, we are interested in application runtime behaviour, so the focus should be on the behavioural aspects of the model: state machines and processing code. However, structural parts of xtUML models only partially visualize model's cyclomatic complexity. Because of this, when discussing cyclomatic complexity of structural models, we discuss the complexity of processing code visualized by those models.

In this section, we present our approach for calculating cyclomatic complexity visualized by structural parts of the application (component and class models), as well as the total cyclomatic complexity from the behavioural parts of the model (state machines and processing code). Later, we will utilize these metrics to determine the distribution of cyclomatic complexity across different model layers.

3.1 Cyclomatic complexity of components

Components are foundational building blocks of xtUML models. They are considered as black boxes that communicate though their interfaces. For this reason, the cyclomatic complexity of xtUML components mostly depends on the interfaces that the component uses. Since the component model falls into the category of structural models, we cannot calculate its cyclomatic complexity, but we can calculate the complexity of visualized entry and exit points to the behaviours wrapped within the component.

The basic approach for calculating cyclomatic complexity is constructing a strongly connected single-entry, single-exit (SESE) control flow graph (CFG). Unfortunately, an xtUML component typically has many entry and exit points, and it is not trivial to construct an SESE CFG from it.

The original Henderson-Seller's approach states that multiple entries to a module can be ignored because multiple entries are used as a reuse mechanism for parts of the CFG already taken into account. If we apply this idea to xtUML components, an estimation of cyclomatic complexity of an xtUML component should be calculated by taking into account only exits on all component ports, while entries to the component can be ignored. However, the issue when applying the Henderson-Seller's approach to xtUML components is dealing with multiple entries. Henderson-Sellers considered only strongly connected CFGs with multiple entries, because this is the limitation introduced in the original McCabe's calculus [29]. This means that additional entry nodes in a MESE CFG are reachable even when they are not used as entry nodes (the left graph in figure 5). In other words, there is at least one incoming edge leading to each entry node, implying that there exists at least one control flow path where those nodes are not entry nodes. This justifies ignoring multiple entry nodes when calculating cyclomatic complexity using the Henderson-Sellers approach.

However, an entry point into an xtUML component is an entry point to an implementation of a port incoming message which cannot be invoked from behaviours within a component. This means that entry nodes are not part of any existing path in a CFG and that such graphs are not strongly connected (even when a *single* virtual edge is added; the right graph in figure 5). Consequently, neither the original, nor the adapted Henderson-Sellers calculation can be applied to such graphs. To handle this, we will adapt an approach similar to what Henderson-Sellers used for single-entry, multiple-exit modules. For n entry nodes, we will add one new (virtual) node and n new edges to connect the new virtual node with all entry nodes. The new virtual node will then act as a single entry for the CFG which will make the graph an SESE. This effectively increases cyclomatic complexity by $n - 1$. When we incorporate this with the existing Henderson-Sellers [17] formula applied to MEME CFGs we get:

$$CC_{meme} = e - n + p + 1 + \sum_{j=1}^{p-1}(r_j - 1) + \sum_{k=1}^{p-1}(i_k - 1) \tag{2}$$

where e is the number of edges, n is the number of nodes, p is the number of (graph) components, r_j is the number of exits from j-th connected component and i_k is the number of entries to the k-th component in the CFG. In case there exists only a single connected graph component, the equation becomes:

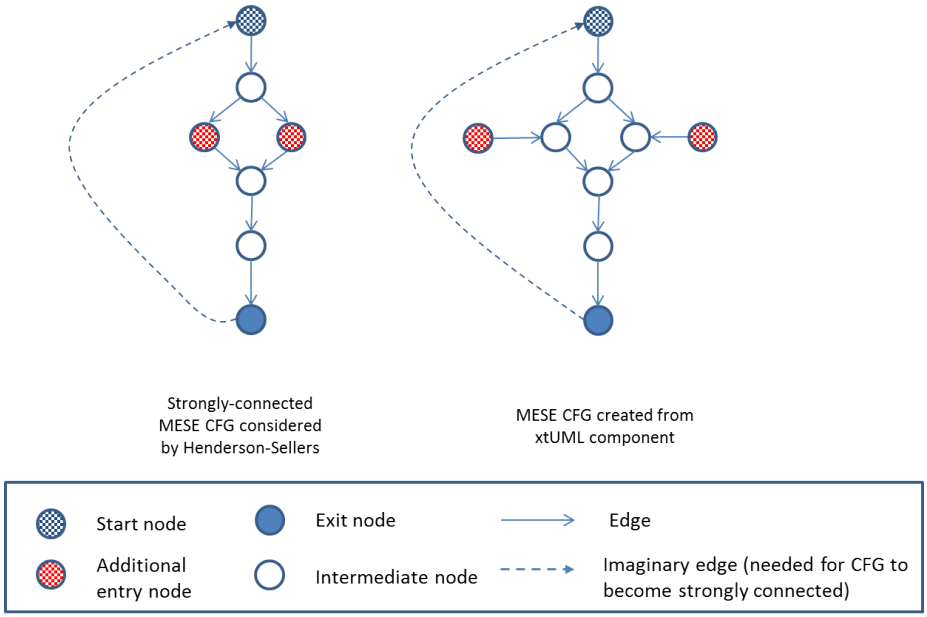

Fig. 5 MESE CFG considered by Henderson-Sellers [17] and the one created from xtUML component.

$$CC_{meme} = e - n + 2 + (r - 1) + (i - 1) = e - n + r + i \tag{3}$$

where r is the number of exits and i the number of entries to the CFG. Notice that eq. 2 and eq. 3 assume the existence of at least one exit and at least one entry, in which case we get the original Henderson-Seller equation. Applying this to eq. 3 results with:

$$CC_{compFull} = e - n + r + i = e - n + N_{op} + N_{sig} \tag{4}$$

where N_{sig} denotes the number of signals and N_{op} the number of operations on all ports of a given xtUML component. Since each operation and each signal represents either an entry or an exit from an xtUML component, the sum of all operations and all signals on all xtUML component ports $(N_{op} + N_{sig})$ is actually equal to the sum of all entries and all exits from that component $(r + i)$. Notice that we did not differentiate between synchronous and asynchronous entries and exits assuming that they equally contribute to cyclomatic complexity. For details about this please refer to section 3.4.3 and figures 11 and 13.

Eq. 4 refers to the complete CFG of an xtUML component, including all of its content. The part of overall xtUML model cyclomatic complexity visualized by the component model can be calculated with the following equation:

$$CC_{comp} = N_{op} + N_{sig} \tag{5}$$

The Henderson-Sellers approach (and our alteration for multiple entries) assumes that a complete component CFG has at least one entry and at least one exit node. However, in xtUML a component does not always need to have ports or

messages on ports. Such components do not communicate with other components, their complete CFG is closed inside the component which is used only as a package. In that case, entry and exit nodes still exist but they are not explicitly exposed in the component model which, in that case, does not give us any information about cyclomatic complexity.

Notice that semantically equivalent interfaces may expose different cyclomatic complexities, depending on the level of abstraction used to create messages on the interface. A more general interface will have fewer messages, but it will also have a larger total number of parameters. For example, consider a simple calculator component: instead of four messages *add*, *subtract*, *divide*, and *multiply* with two parameters, we can have a single message (*performCalculatorOperation*) with three parameters where the additional parameter encodes the calculator operation to perform. The total cyclomatic complexity of the component remains the same because the port message code in the case with single message will need to branch according to the value of the third parameter. This implies that, although eq. 4 seems to depend on the number of operations and signals on component interfaces, it actually does not, because its effect is compensated with the remainder of the equation (the $e - n$ part). At the same time, the cylomatic complexity visualized by the component model is lower in case of a more general interface because only one interface operation is used. Despite the functional and cyclomatic equivalence of models, a model with a specific interface is obviously simpler to comprehend because it states the interface functionality more clearly.

3.2 Cyclomatic complexity of classes

Similarly as the component model, the class model describes the structure, and not the behaviour of an xtUML model. As with a component model, this implies that a class model only partially visualizes the cyclomatic complexity of processing.

A class model visualizes the cyclomatic complexity of processing code with its operations and with the relations it exposes. Similarly to port operations, each class operation adds two edges and a single node to the CFG effectively increasing the cyclomatic complexity visualized by the class model by one (see figure 11):

$$CC_{oper} = \sum_{n=1}^{N_{cl}} N_{oper}(n) \tag{6}$$

where CC_{oper} stands for cyclomatic complexity of class operations, N_{cl} is the number of classes, and $N_{oper}(n)$ is the number of operations in the $n - th$ class.

Class relations also have an influence on the processing model. The processing code operates on class instances; it creates and relates them, selects them across relations, reads or changes their values, unrelates them, and eventually deletes them. Single- or multi-hop instance selections across relations are a very important part of the processing model, because each relation (chain) exposes specific meaning within an application. Depending on the multiplicity and the conditionality of a relation traversed in a selection statement, it will be followed by a *i)* conditional loop (iterating across selected instances), *ii)* a conditional *if* branch (checking for emptiness of a selection variable), *iii)* both, a loop and an *if* branch (if the

relation was conditional and multiple), or *iv)* neither, if the relation was unconditional and single. A loop for iteration across selected instances and/or a check for result emptiness are needed in order to evade runtime errors. This is how selections influence the cyclomatic complexity of processing code.

The general approach for calculating cyclomatic complexity visualized by a relation in a class model relies on the assumption that all directions of all relations in a class model are used in at least one selection statement in the processing code. For simple relations there are only two possible selection directions: from one class to another, and in reverse. The result of each of those selection directions may be conditional and/or multiple, depending on the flags on the remote end of the relation direction used in the selection. If the result of a selection can be empty, a single *if* branch (required to check the emptiness of the result) will increase the cyclomatic complexity by 1. If a selection can result in a set of instances, a *for* loop that iterates through the instances will also increase the cyclomatic complexity by 1. In the worst case scenario, where both relation ends of a simple relation are conditional and multiple, there are four additional control-flow paths added, or:

$$CC_{sr} = \sum_{i=1}^{N_{sr}} [N_{ce}(i) + N_{me}(i)] \tag{7}$$

where CC_{sr} stands for cyclomatic complexity of simple relations, N_{sr} the number of simple relations, $N_{ce}(i)$ the number of conditional relation ends in the i-th simple relation, and $N_{me}(i)$ the number of multiple relation ends in the i-th simple relation.

Before we can calculate complete cyclomatic complexity of a class model, we need to define the cyclomatic complexity of two remaining relations in xtUML: *generalization* and *associative relation.*

3.2.1 Generalization relation and cyclomatic complexity

Each generalization relation defines $2n$ selection directions: n directions towards n subclasses and n in reverse direction. However, the n directions from subclasses towards a general class are always unconditional and single and do not require a conditional check or an iteration. The remaining n directions towards subclasses are always conditional (they require an emptiness check with a conditional *if* branch) and single (they do not require a conditional loop iteration). This means that each generalization relation increases cyclomatic complexity by n additional control-flow paths or:

$$CC_{gr} = \sum_{j=1}^{N_{gr}} N_{sub}(j) \tag{8}$$

where CC_{gr} stands for the cyclomatic complexity of generalization relations, N_{gr} is the number of generalization relations and $N_{sub}(j)$ is the number of subclasses in the j-th generalization relation.

3.2.2 The effect of associative relations on cyclomatic complexity

With associative relations there are three classes involved: two *related* classes and a single *associative* class. Each of these classes may be involved in a selection in two different ways, which means that there are six different selection directions, each with a different meaning. However, the multiplicity and conditionality of some of those directions is not trivial to detect. For this reason, we will use a *replacement* model consisting of two simple relations, from each *related* class towards the associative class. We will refer to these two models as the *original* and *replacement* class models.

Since the existence of an associative class instance assumes the existence of a pair of related classes, all selections starting from an associative instance always result with a single instance. This implies that relation ends towards related classes are unconditional and single. A selection starting from one of the related classes to another related class, and a selection from the same related class towards the associative class share the same conditionality. This simply means that if there is no associative class instance found, there should be no remote related class instance found as well.

The same rule applies to the multiplicity flag, but only in case when the associative link is single. In this case, the multiplicity flag at an associative class end in the replacement class model is the same as the multiplicity of the direction towards the remote related class in the original model. However, when the associative link is multiple, the relation ends at the associative class on the replacement model are always multiple, regardless of the multiplicity in the original class model.

Now that we know how to determine the multiplicity and conditionality of all selection directions across an associative relation, we can determine its effect on cyclomatic complexity. It is important to emphasized that, from the cyclomatic complexity perspective, not all branches are independent. This is a consequence of the fact that the conditionality flag of the selection direction towards an associative class is always the same as the conditionality of the selection direction towards a remote related class. This actually lowers the cyclomatic complexity because, without any semantic change, one of the dependent branches can be eliminated and their processing can be merged into a single branch. Table 2 and eq. 9 summarize the associative relation effect to processing code cyclomatic complexity.

$$CC_{ar} = \sum_{k=1}^{N_{ar}} [N_{ce}(k) + N_{me}(k) + 2N_{ma}] \tag{9}$$

where CC_{ar} stands for the cyclomatic complexity of associative relations, N_{ar} the number of associative relations, $N_{ce}(k)$ the number of conditional relation ends, $N_{me}(k)$ the number of multiple relation ends, and $N_{ma}(k)$ the number of multiple associative link ends in the k-th associative relation.

Finally, class model cyclomatic complexity is calculated as a sum of cyclomatic complexities of all its operations and relations:

$$CC_{cls} = CC_{oper} + CC_{sr} + CC_{gr} + CC_{ar} \tag{10}$$

where CC_{cls} stands for the cyclomatic complexity of a class model, CC_{oper} is the cyclomatic complexity of class operations, CC_{sr} is the cyclomatic complexity

Table 2 Summary of associative relation effect to cyclomatic complexity

Flag	Adds to cyclomatic complexity of CFG	Count	Maximal total effect
Conditionality on related class end	+1	2	+2
Multiplicity on related class end	+1	2	+2
Multiplicity of associative link	+2	1	+2
Total			+6

of simple relations, CC_{gr} is the cyclomatic complexity of generalization relations, and CC_{ar} is the cyclomatic complexity of associative relations.

3.3 Cyclomatic complexity of state machines

An xtUML state machine describes runtime behaviour of a class instance. This implies that, unlike component or class models, cyclomatic complexity of state machines directly influences the overall xtUML model cyclomatic complexity. In this section we describe two approaches for calculating cyclomatic complexity of xtUML state machines.

Calculating cyclomatic complexity of UML-based state machines is relatively straightforward. The basic formula is given with the following equation [10]:

$$CC_{sm1} = N_t - N_s + 2 \tag{11}$$

where N_t is the number of transitions and N_s the number of states within a state machine. Eq. 11 is obtained from the original McCabe's formula by considering the state machine graph as a CFG: each state is represented by a node and each state transition by an edge.

This is the most common way of constructing a CFG from an xtUML state machine. The problem with this approach is that it does not consider the complete xtUML state machine described by the state-event matrix.

When discussing cyclomatic complexity, a CFG describes possible paths of a single, uninterrupted thread of execution. However, this is not how a state-machine actually works. Each instance of a state machine runs in its own logical thread. The execution of each instance state machine is independent of other state machines and is only affected by the events sent to this particular instance. From the modeler's perspective, each instance state machine gains execution control when it receives an event, resulting in the execution of a corresponding transition effect and state entry behaviour; an *Request-To-Completion* (RTC) step (see section 2.1.3 for details). After an RTC step execution finishes, the state machine execution is paused until the next event is received. While waiting for the next event, the state machine temporarily loses execution control.

Since state-machines communicate asynchronously, a new event may be received while the instance is processing an RTC step. For this reason, there exists an *event pool* that stores events that are waiting to be processed, along with a dispatching mechanism that continuously checks for incoming events and triggers the corresponding state machine RTC executions.

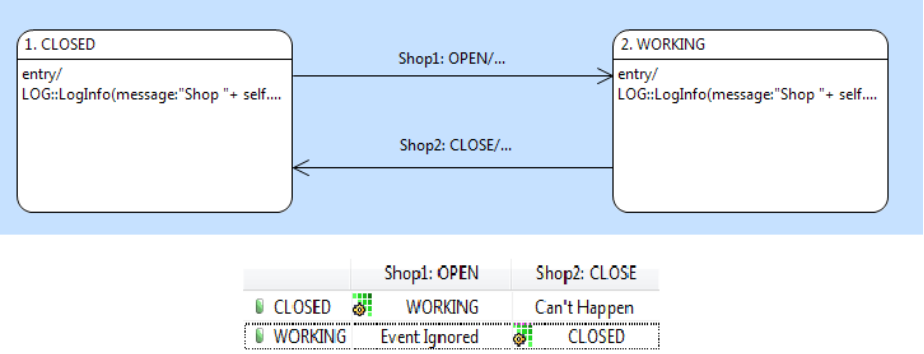

Fig. 6 State machine of the *Shop* class. Notice the *Event Ignored* case when event *OPEN* is received in *WORKING* state and *Can't Happen* case when event *CLOSE* is received in state *CLOSED*.

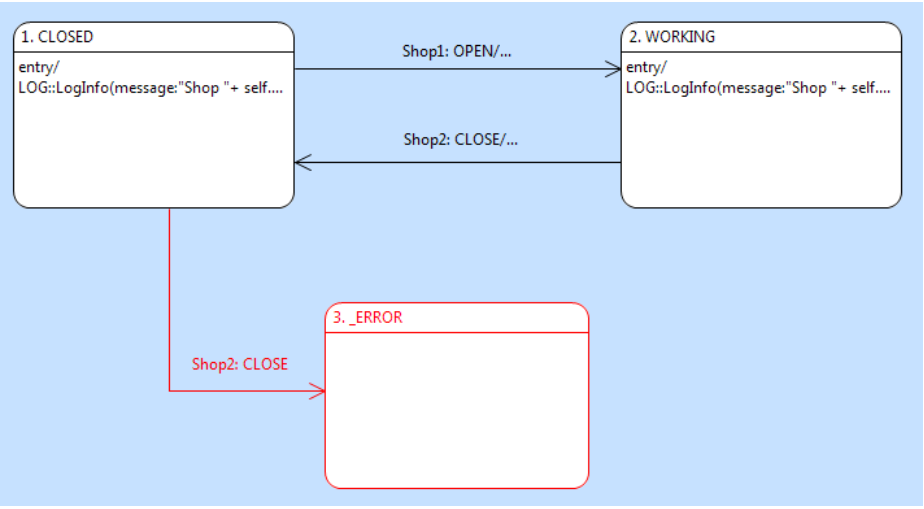

Fig. 7 Simplified replacement state machine of *Shop* class.

Figure 8 shows a snippet of pseudo-code that describes the execution semantics of *Shop* instance state machines. In every while loop iteration, first the event pool is checked for any waiting events (line 5). In case an event exists, its event type is checked (lines 8, 15 and 19) and the corresponding RTC step (transition effect and the state entry behaviour) is executed. In case of an *Event Ignored*, the loop simply skips an iteration without doing any work (line 13), while in case of a *Can't Happen* (lines 19 – 22), the loop is stopped, and the state machine is terminated.

By using this execution semantics, a complete CFG with a single entry and a single exit node can be constructed. A concrete example that shows a state machine CFG obtained through this execution semantics is shown in figure 9. The part of the graph outside the rectangle does not depend on the state machine and will be the same for any state machine because it represents the state machine infrastructure code. This implies that, for smaller state machines such as this

```
BEGIN
    SET running = true
    SET currentState = CLOSED
    WHILE running
        SET event = dequeue event from the event pool
        IF event is not empty
            IF currentState is WORKING
                IF event is CLOSE
                    execute WORKING->CLOSED transition effect behaviour
                    SET currentState = CLOSED
                    execute CLOSED state entry behavior
                ENDIF
                (Do nothing if event is OPEN)
            ELSE IF currentState is CLOSED
                IF event is OPEN
                    execute CLOSED->WORKING transition effect behaviour
                    SET currentState = WORKING
                    execute CLOSED state entry behavior
                ELSE IF event is CLOSE
                    SET running = false
                    SET currentState = ERROR
                ENDIF
            ENDIF
            IF currentState is final
                SET running = false
            ENDIF
        ENDIF
        wait for some time before iterating again
    ENDWHILE
END
```

Fig. 8 A pseudo-code description of the execution semantics of Shop instance state machines.

one, cyclomatic complexity introduced by state machine execution infrastructure is comparable to the cyclomatic complexity of the state machine itself. However, for larger state machines this will not be the case.

In figure 9, the part within the dashed rectangle actually depends on the structure of the state machine. For each state in the state machine we have two *state* nodes in the CFG ($2N_{st}$ in eq. 14), which are connected to infrastructure nodes with two edges ($2N_{st}$ in eq. 13). For each non-ignored event in the state, we have an *event* node connected to the state nodes with two edges ($\sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)]$ part in both equations, in eq. 13 multiplied with 2). An *event* node actually contains RTC execution as a synchronous invocation of *transition effect* and *state entry* actions ($2N_t$ in eq. 13). In case a state ignores any number of events it will also have a single direct edge between the two state nodes, regardless of the number of ignored events (N_{stei} in eq. 13). We will use this rule and figure 9 to induce the general rule for calculating strict cyclomatic complexity of an xtUML state machine:

$$CC_{sm2} = N_{edge} - N_{node} + 2 \tag{12}$$

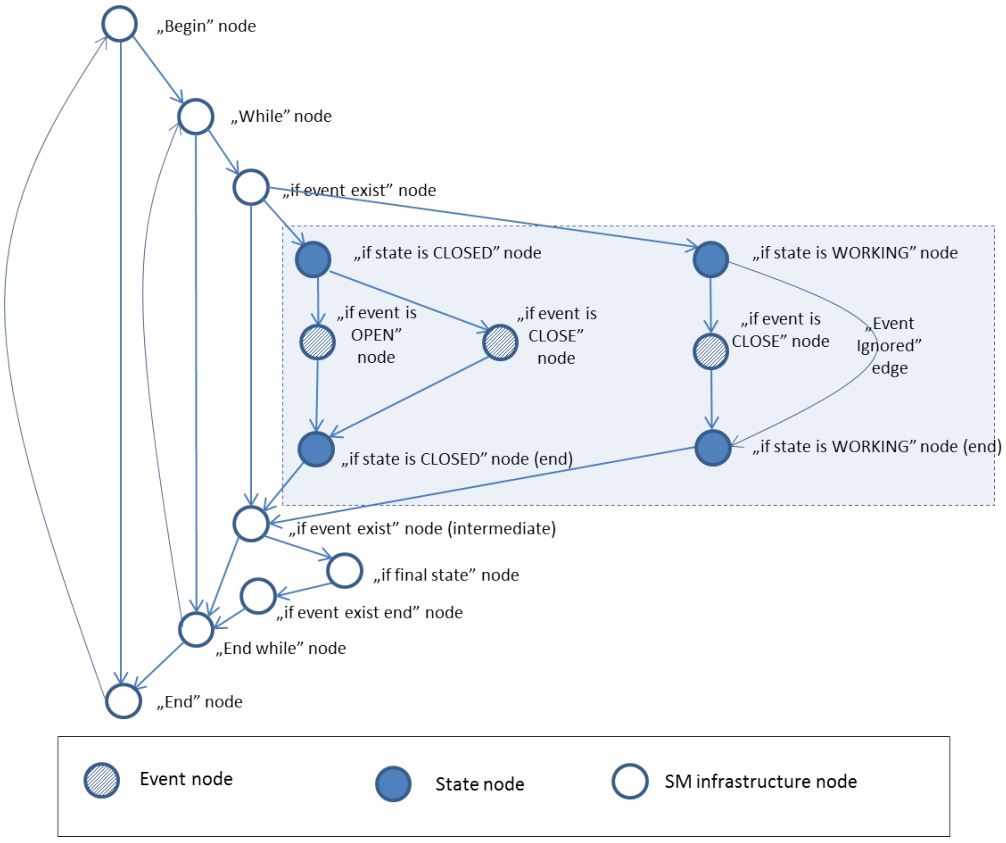

Fig. 9 A CFG created from the execution semantics of the Shop class instance state machine.

where the number of edges N_{edge} and the number of nodes N_{node} is defined with the following equations:

$$N_{edge} = 12 + 2N_{st} + 2\sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t \tag{13}$$

$$N_{node} = 8 + 2N_{st} + \sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)] \tag{14}$$

where N_{st} is the number of states, N_{ev} the number of events, $N_{ei}(s)$ the number of *Event Ignored* records the state s has in the matching row of the state machine matrix, N_{stei} the number of states that have at least one *Event Ignored* record in the matching row of the state machine matrix, and N_t the number of transitions. Since the number of transitions is actually the number of RTC steps and each RTC step contains (invocations of) exactly two bodies, each RTC step adds 2 to the overall cyclomatic complexity of a state machine (therefore the $2N_t$ in the equation). Notice that, if a state has more than one *Event Ignored* record in the matching row, it will still have only one "event ignored" edge in a CFG (see figure 9). This explains the N_{stei} part of the equation.

When we substitute the expressions for N_{edge} and N_{node} into eq. 12, we end up with the following equation:

$$CC_{sm2} = N_{edge} - N_{node} + 2$$

$$CC_{sm2} = 12 + 2N_{st} + 2\sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t - (8 + 2N_{st} + \sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)]) + 2$$

$$CC_{sm2} = 6 + \sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t \tag{15}$$

Since the number of events is the same for each state (row) we have the following equation:

$$\sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)] = N_{st} * N_{ev} - \sum_{s=1}^{N_{st}} N_{ei}(s) \tag{16}$$

Furthermore, if each state has at most one *Event Ignored* record, N_{stei} is equal to the total number of *Event Ignored* cases in the state machine, i.e.:

$$N_{st} * N_{ei}(s) - \sum_{s=1}^{N_{st}} N_{ev} = N_{st} * N_{ev} - N_{stei} \tag{17}$$

When applied to equation 15 we get:

$$\begin{aligned} CC_{sm2} &= 6 + \sum_{s=1}^{N_{st}} N_{ev} - \sum_{s=1}^{N_{st}} N_{ei}(s) + N_{stei} + 2N_t \\ &= 6 + \sum_{s=1}^{N_{st}} N_{ev} - N_{stei} + N_{stei} + 2N_t = 6 + N_{st} * N_{ev} + 2N_t \end{aligned} \tag{18}$$

In eq. 18, we can observe that the state machine execution infrastructure adds a constant amount (6) to cyclomatic complexity. This constant becomes less significant as the state machine grows. In addition, such calculus for cyclomatic complexity is inline with the intuition because the product of the number of states and the number of events actually represents the number of possible linearly independent paths through the state machine, while the $2N_t$ actually denotes the number of actions we actually invoke in those paths.

Note also that the constant amount of 6 is introduced by the state machine infrastructure and does not describe complexity directly visible in the state machine model. Therefore, if we are trying to calculate the *visible* state machine complexity we can ignore the constant in our calculus:

$$CC_{sm2vis} = \sum_{s=1}^{N_{st}}[N_{ev} - N_{ei}(s)] + N_{stei} + 2N_t \approx N_{st} * N_{ev} + 2N_t \tag{19}$$

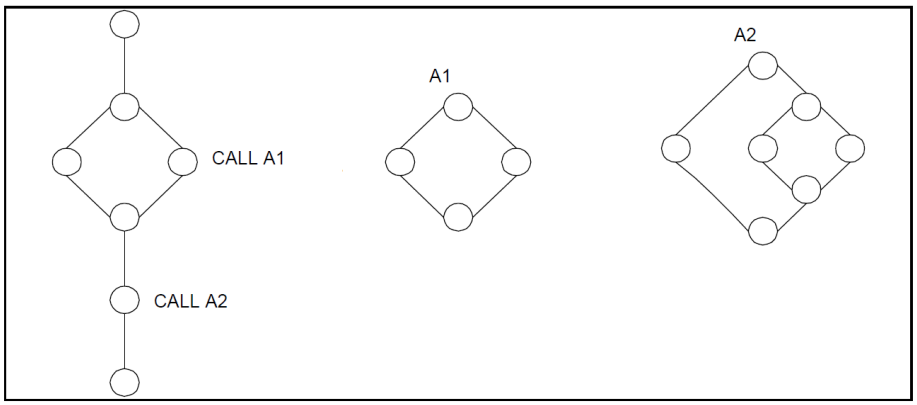

Fig. 10 McCabe's approach to cyclomatic complexity.

3.4 Cyclomatic complexity of processing code

An xtUML processing model represents textual processing instructions and is similar to more traditional programming languages. For this reason, from the cyclomatic complexity point of view, the processing model follows similar control- and data-flow rules as other traditional programming languages. This implies that standard, source-code level complexity metrics can easily be applied to the processing model.

In xtUML, code is organized into OAL actions (often called *bodies*), and there are several ways to calculate their cyclomatic complexity. But first, several considerations have to be made.

3.4.1 Choosing the basic approach

When choosing the basic approach for measuring cyclomatic complexity of processing code there are two options: the original McCabe's approach [29] which considers each subroutine separately, and the Henderson-Sellers approach [17] which creates a single connected component from all subroutines. In order to make this decision, we have to consider their effects on cyclomatic complexity.

In the original McCabe's approach, the CFG of each subroutine has to be strongly connected. This means that each subroutine should have its single entry and single exit node connected with an additional virtual edge. The resulting equation for McCabe's cyclomatic complexity of a CFG with p connected components is:

$$CC_{McCabe} = \sum_{i=1}^{p} [N_{edge}(i) - N_{node}(i) + 1 + 1] = N_{edge} - N_{node} + 2p \qquad (20)$$

In Henderson-Sellers approach, a single CFG is created from all subroutines. To achieve this, a node containing a call to the subroutine is split into two nodes: one used to connect to the subroutine entry node and one used to connect back from the subroutine exit node. This means that, for each subroutine except the one

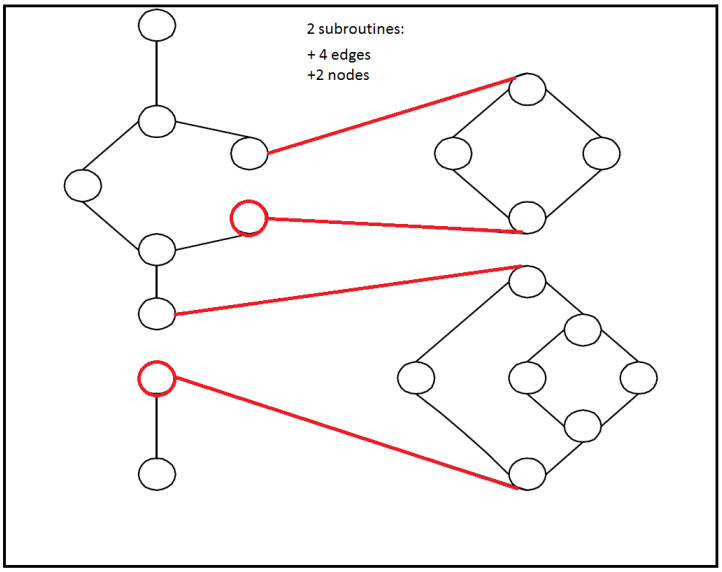

Fig. 11 Henderson-Sellers approach to cyclomatic complexity. Compare this graph with the one on figure 10

we are merging into, we have to add a single additional node and two additional edges, and each subroutine increases cyclomatic complexity by 1. For this reason eq. 21 contains $(p-1)$. In order to make this complete CFG strongly connected, a single edge from the exit node to the entry node has to be added (thus the $+1$ in eq. 21).

$$CC_{HS} = N_{edge} - N_{node} + 1 + (p-1) + 1 = N_{edge} - N_{node} + p + 1 \tag{21}$$

$$CC_{HS} = \sum_{i=1}^{p}(d_i + 1) = D + 1 \tag{22}$$

where d_i represents the number of decisions (branching places) in i-th subroutine, while D represents the total number of decisions in all subroutines. Expressing the Henderson-Sellers equation [17] using edges and nodes requires an integrated CFG of an application to be created. Using the number of decisions instead, simplifies practical usage of the equation.

It is important to emphasize that modularization has no effect on the Henderson-Sellers cyclomatic complexity (see figure 11). As the number of components (p) is reduced by merging them back into the caller's graph (eq. 21), the number of nodes N_{node} is also reduced by the same amount (figure 11). This is not the case with the original McCabe's approach. Also notice that, since there is only a single control flow graph, Henderson-Sellers cyclomatic complexity can be easily calculated by counting the number of decisions in all subroutines (see eq. 22). For these reasons, in our approach, we use Henderson-Sellers's approach as a base for calculating cyclomatic complexity.

3.4.2 Handling multiple synchronous calls of the same subroutine

The standard Henderson-Seller's approach (as well as the original McCabe's approach) ignores multiple synchronous calls of the same subroutine. The reason for this is that its standard usage is to estimate application testing effort which is not affected by the number of times a subroutine is called. Since our goal is to use cyclomatic complexity to estimate cognitive complexity or understandability, this is not good enough, because multiple calls to the same subroutine influence the program's cognitive complexity and understandability. For this reason, we modify the Henderson-Seller's approach to also take into account the number of subroutine calls.

We modify eq. 21 in the following way: the expression $p-1$ should be replaced with the total number of subroutine calls N_{call}, because cyclomatic complexity will be increased by 1 for each subroutine call, and not for each subroutine definition (see figure 12). This leads to the following equation:

$$CC_{allBodies} = N_{edge} - N_{node} + 1 + N_{call} + 1 = N_{edge} - N_{node} + N_{call} + 2 \quad (23)$$

Eq. 23 assumes the existence of a CFG and is not suitable for practical uses. For this reason, we have to adapt eq. 22 as well. Since eq. 22 assumes that each subroutine (except the main one that calls all others) is called exactly once, we need to increase the cyclomatic complexity by $N_{call} - (p-1)$ (because $p-1$ of them are already included). This results with the following equation:

$$CC_{allBodies} = D + 1 + (N_{call} - (p-1)) = D + N_{call} - p + 2 \quad (24)$$

where D represents the number of decisions in all components (subroutines), N_{call} the number of calls in all subroutines and p the number of components (subroutines).

With this, we have replaced the original assumption that each subroutine is called *exactly once* with the assumption that a subroutine is called for each occurrence of a call expression (implying $N_{call} \geq p$ in the equation).

This approach takes into account multiple subroutine calls similarly to Shepperd's approach [53]. The difference is that Shepperd extends the original McCabe's approach, which makes the calculus somewhat more complex. In this approach we are using Henderson-Sellers approach as a base, but instead for each subroutine definition, we are incrementing the overall complexity by 1 for each subroutine *call*. This difference can be clearly seen on figure 12.

In case a subroutine is called only once, modularization is done only for abstraction sake and our approach is equivalent to Henderson-Sellers approach. However, if there are multiple calls to the same subroutine (in case modularization is done because of reuse), cyclomatic complexity increases, but not as much as it would increase if modularization is not done at all (in which case the calling CFG would contain as many subroutine CFGs as there are calls to the subroutine). This is different from the Henderson-Sellers approach in which cyclomatic complexity remains the same regardless of the reasons for modularization. We consider that our approach is inline with the intuition that cognitive complexity does not increase with modularization itself, but does reduce overall complexity if there are multiple calls to the same subroutine.

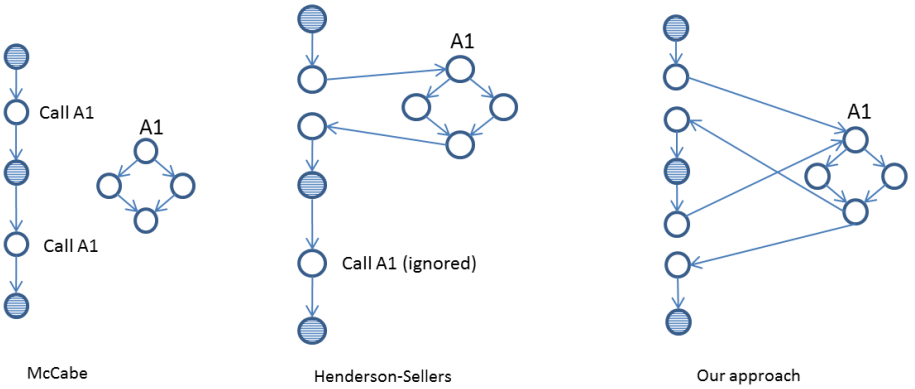

Fig. 12 Different approaches for handling multiple calls when calculating cyclomatic complexity.

3.4.3 Handling asynchronous calls

When dealing with components ports, we have already seen that communication can be both synchronous and asynchronous. In that case, we decided to treat asynchronous communication as synchronous communication where it is not necessary to wait for the execution control to return. This means that the effects on the CFG for those two types of communications is somewhat different (figure 13). Notice however that, despite the different effect on the CFG, they equally contribute to the cyclomatic complexity: synchronous calls introduce two edges but also add one additional node, unlike asynchronous calls. Practically, this means that we can reuse eq. 23 (and 24) for both synchronous and asynchronous calls.

Asynchronous calls (invocations) in OAL language are represented by the event and signal sending statements. In addition to that, upon the creation of a class instance that defines the state machine, the state machine will be started. This will be treated as the third type of asynchronous invocation. Notice that the creation of class instance that does not have a state machine will not count as an asynchronous invocation.

3.4.4 Handling compound branching conditions

A single branch with a compound condition can be split into multiple branches with simple conditions. This effectively increases the cyclomatic complexity. However, the initial complex condition can also be abstracted away into a single boolean flag which will have the same cyclomatic complexity as a single conditional branch. Since we are observing cyclomatic complexity from the modelling perspective, we assume the modeller will use the most abstract alternative for presenting compound conditions. Practically, this means we will use the number of decision points as D in the equation for calculating cyclomatic complexity and not the number of conditions.

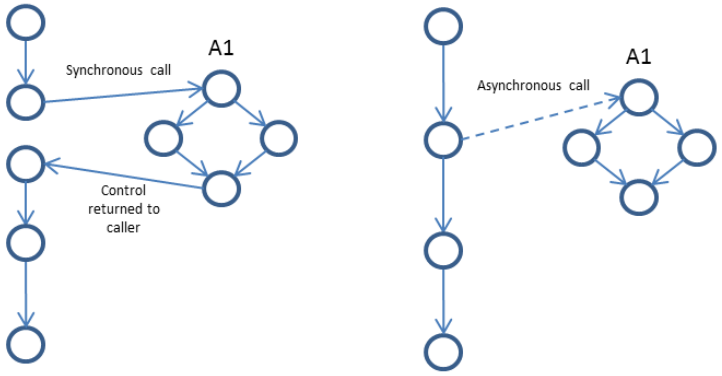

Fig. 13 Handling of asynchronous communication and its effect on cyclomatic complexity. The overall cyclomatic complexity remains the same as if the call was synchronous, because we have reduced both the number of edges and the number of nodes equally, by 1 for each call.

4 Measuring complexity distribution

A complete xtUML model is obtained by semantically integrating four different models: component, class, state machine, and processing models. So far, we have been analysing the cyclomatic complexity of each of those models separately, with minimal considerations towards other models or the xtUML model as a whole. In this section, we will describe our approach to *i)* calculating cyclomatic complexity of a complete xtUML model, and *ii)* calculating the distribution of the model complexity across different model layers.

4.1 Calculating the overall cyclomatic complexity

The first step in calculating cyclomatic complexity is constructing a CFG. In case of complete xtUML models, the CFG construction starts with components, since they are the basic building blocks of xtUML systems. Components contain classes, which contain operations and act as wrappers around state machines. For this reason, constructing a CFG graph of the whole model starts with constructing a CFG of a single xtUML component.

An incoming port message, whether it is synchronous or asynchronous, has its implementation on a port. Component ports are static constructs that do not have to be created, while most of the component functionality is distributed across class instances. This means that the main task of a port message implementation is to find the correct class instance (or create a new one) and forward the message to it. Typically, but not mandatory, if the incoming port message is asynchronous, an asynchronous message (an event) is passed to the instance state machine. Similarly, if the incoming port message is synchronous, usually a synchronous operation on the selected instance is invoked.

In any case, a class instance represents a context (i.e. provides contextual data) used by behaviors in operations and state machines. This implies that, in order to create a CFG of an xtUML component, we can observe operations and state

machines as standalone entities, without their container classes. From cyclomatic complexity perspective, which is not concerned with data complexity, classes, their instances, and the data context they provide, are not relevant. They are only used to logically organize those behaviours and they have no effect on cyclomatic complexity of an xtUML component. Practically, this means that class model cyclomatic complexity can be ignored when calculating complexity of a complete xtUML component. This is inline with the fact that cyclomatic complexity of a class model is included in the cyclomatic complexity of processing code and does not need to be explicitly taken into account.

As a base for calculating cyclomatic complexity wrapped within a component, we use eq. 4, where an xtUML component is considered as an MEME CFG constructed from a set of synchronously communicating bodies and asynchronously communicating state machines. Except with asynchronous invocation (event and signal sending) statements, the synchronous and asynchronous part of a component's CFG are connected with edges introduced by instance creation statements. In case a class defines an instance state machine, the instance creation statements start the execution of a state machine. The total number of edges and nodes within a component in eq. 4 is therefore determined as a sum of all edges and nodes in all bodies and state machines wrapped in a component. While eq. 24 provides this number for *all* bodies within a component, eq. 15 gives cyclomatic complexity for a *single* state machine. This means that we need to sum the number of edges and nodes for all state machines within a component. However, when observing state machines in the context of a component, they become integrated in a single component CFG. For this reason, a virtual edge connecting each exit and each entry node of a CFG is not needed. Therefore, we need to decrement the overall component cyclomatic complexity by 1 for each state machine in a component. Applying this to eq. 4 results with the following equation:

$$\begin{aligned} CC_{comp} &= CC_{allBodies} + CC_{allSm} + N_{op} + N_{sig} \\ &= CC_{allBodies} + \sum_{i=1}^{N_{sm}} CC_{sm}(i) - N_{sm} + N_{op} + N_{sig} \end{aligned} \tag{25}$$

where $CC_{allBodies}$ represents cyclomatic complexity of all bodies within an xtUML component (given with eq. 24), CC_{allSm} represents the cyclomatic complexity of all state machines within an xtUML component, $CC_{sm}(i)$ is the cyclomatic complexity of the i-th state machine (given with eq. 15), N_{sm} is the number of state machines within a component, and N_{op}, N_{sig} are the number of operations and signals on all component ports, respectively. In the rest of the paper we use eq. 25 for calculating the total cyclomatic complexity of an xtUML component.

4.2 Calculating the distribution of cyclomatic complexity

Complexity in an xtUML model can be distributed: *i) horizontally*, between the elements of the same type, and *ii) vertically*, between different types of models. An example of *horizontal* complexity distribution is determining how complexity is distributed across different components of a system, or among different classes within a single component. *Vertical* complexity, on the other hand, compares complexities exposed on component, class, state machine, and processing model levels. This

separation between *vertical* and *horizontal* complexity is specific for model-driven technologies, and does not have an equivalent in traditional software development metrics.

In our approach, for calculating *vertical* distribution of cyclomatic complexity across different xtUML sub-models (i.e. component, class, state machine and processing models), we will use cyclomatic complexity of a sub-model *relative to the complexity of the complete xtUML model*. In order to calculate the cyclomatic complexity of a component, class, state machine and processing model, we will use equations 5, 10, 19 and 24 respectively. To calculate the complete xtUML model cyclomatic complexity, we will use eq. 25 from the previous section. Although eq. 25 contains the number of operations and signals on all component ports, the cyclomatic complexity of the component layer does not influence the overall cyclomatic complexity (as is already explained in the last paragraph of section 3.1). Similar is true for class model complexity because a class and component model complexity only visualize a subset of procedural model complexity and that they *do not* influence the total xtUML model cyclomatic complexity. There is, however, a value in analysing the degree of visually exposed complexity, so we include those complexities in the process. Since cyclomatic complexity depends on the number of execution (control) paths, we expect that processing model cyclomatic complexity will dominate over cyclomatic complexities of other layers. In addition, we can also compare the cyclomatic complexity expressed through graphical models (components, classes, and state machines) with the complexity of the textual model expressed in the processing model.

Horizontal complexity can be calculated on several layers. On system level, horizontal complexity distribution can be calculated between different components of a system, including all complexities of all state machines and bodies within a component. This provides information about the distribution of cyclomatic complexity among different components. Inside a single component, complexity distribution can be analyzed across different classes, by taking into account the complexity of their state machines and bodies. Although component complexity may also be located outside its classes (ports, functions, bridges), the majority of cyclomatic complexity will be wrapped within classes. The distribution of complexity among classes will provide useful information about key classes within a component and their relative complexity. The contribution of a single class can be calculated by using the following equation:

$$CC(c) = CC_{body}(c) + CC_{sm}(c) = D(c) + N_{call}(c) - p(c) + CC_{sm}(c) \qquad (26)$$

where $D(c)$ is the total number of decisions in all bodies of class c, $N_{call}(c)$ the total number of synchronous and asynchronous calls in all bodies contained in the class, and $p(c)$ the number of bodies defined in the class.

On the lowest level, we will analyse the distribution of processing complexity across different bodies, taking into account the number of decisions (D) and calls (N_{call}) contained within each body (b).

$$CC(b) = D(b) + N_{call}(b) \qquad (27)$$

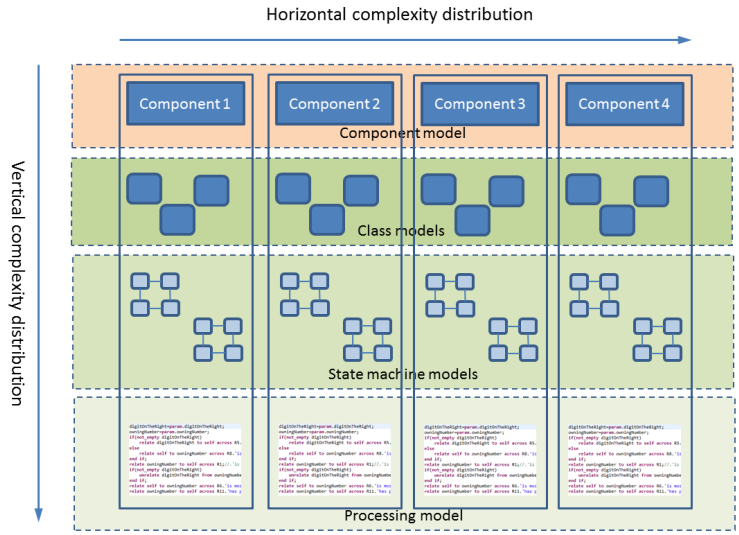

Fig. 14 Horizontal and vertical complexity distribution.

5 Hypothesis and experiment setup

The goal of our experiment is to test the influence of xtUML model complexity distribution on the understandability of software models. For this reason, we specify our hypothesis in the following way:

- **Null hypothesis (H_0):** The distribution of cyclomatic complexity does not affect the understandability of xtUML models.
- **Alternative hypothesis (H_a):** The distribution of cyclomatic complexity significantly affects the understandability of xtUML models.

In order to test this hypothesis, we have used three different versions of the same application. All test applications implement the same set of requirements; they have the same interface and functionality, and they pass a common test suite. The only difference is in their internal implementation, which results with the fact that every application has a significantly different distribution of complexity. Our goal was to study the effect of complexity distribution on the understandability of xtUML models. For this we have used students, by measuring the quality of their understanding for each test application.

5.1 Study objects

The experiment objects were three functionally equivalent calculator applications developed with xtUML by an xtUML expert. Because of the practical limitations of the study, three versions of the application are developed specially for this experiment. Each application is composed out of a single xtUML component, the *SimpleCalculator* component. Each application has its own version of the *SimpleCalculator* component, but all component versions have the same interface and

pass the same set of tests (meaning they are functionally equivalent). The three components are intentionally modelled differently, in order to demonstrate different ways to distribute application complexity:

A. *Model 1* uses the structured programming paradigm which relies on functional decomposition and data structures. It makes no use of object-oriented concepts such as classes, nor does it use state machines.
B. *Model 2* heavily relies on classes, but it does not use state machines.
C. *Model 3* uses both classes and state-machines.

All three applications, each with a model and a test suite, can be found at the *xtUMLProjects* directory at the public git repository available at [56].

In the following sections, we will compare the three models according to various dimensions.

5.1.1 Comparing the models in terms of LOC

The difference in internal structure resulted in the fact that three models also differ in their total LOC (shown in Table 3 and figure 15). Model 1 implements the desired functionality in only 333 LOC, while Model 3 uses 518 LOC (a difference of 36%). In addition, the total number of non-empty bodies in Model 3 is more than twice the number in Model 1 (78 compared to 32). In comparison, Model 2 has a similar (but somewhat lower) number of non-empty bodies. Model 3 has the lowest average LOC per body, while all three have comparable standard deviation.

Table 3 Horizontal distribution of lines-of-code (LOC) across bodies

	Model 1	**Model 2**	**Model 3**
Total non-empty bodies	32	67	78
Total LOC	333	479	518
Average	10,41	7,15	6,64
Standard deviation	6,72	6,02	6,37
Min value	1	1	1
First quartile	6	2	2
Median	9	5	4
Third quartile	14	9	9
Max value	29	28	29

If we observe LOC distribution across classes in table 4 and figure 16, we can see that Model 3 has lower average and standard deviation than Model 2. This is a consequence of the fact that Model 2 has almost 50% of the total LOC (237 out of total 479) within a single class. Note that Model 1 contains only a single class, which means that it makes no sense to compare it with the other two models, since it obviously has the worst LOC distribution across classes. It is obvious that Model 3 has the best distribution across classes as well.

5.1.2 Comparing the naming conventions used by models

One of the most important factors influencing software understandability is consistency and quality of naming conventions. Since in this work we are not interested

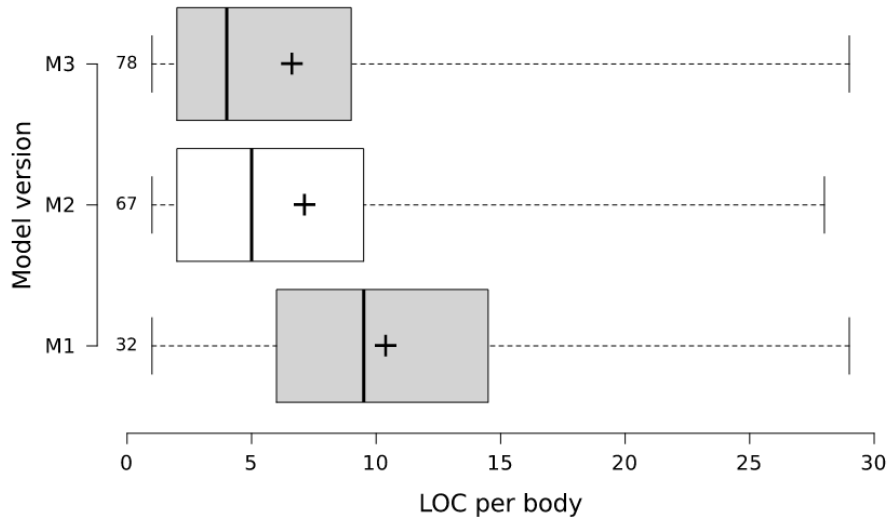

Fig. 15 Horizontal distribution of LOC across bodies.

Table 4 Horizontal distribution of lines-of-code (LOC) across classes

	Model 1	**Model 2**	**Model 3**
LOC within classes	279	418	456
Total classes	1	5	6
Average	279	83,60	76,00
Standard Deviation	0	90,88	49,25
Min value	279	5	27
Max Value	279	237	153
Total model LOC	333	479	518
Class-Total LOC ratio	84%	87%	88%

in the relation between naming conventions and software understandability, we need to minimize its effect as a factor. To verify that our models do not differ considerably in that sense, we used Laitinen's language theory idea [24], in which the notion of a *language* refers to the set of symbols and identifiers used in some (software) document. He identified two main rules for comparing such languages:

- Smaller languages, in terms of number of elements, are easier to understand than larger ones. The main idea here is that each symbol has associated semantics in the context of the language, which needs to be understood.
- It is easier to understand closely related languages than more distantly related languages. The closeness of two languages is determined by the number of common symbols (and semantics) they share. This implies that no absolute measure of language understandability exists, only relative measures are possible.

Our idea was to extract the languages of the three models and compare it with the language of the specification, with regard to these two rules. In order to be able to do this, we used individual words from the names of the model elements as the language symbols of the models. We first extracted all string attributes from all

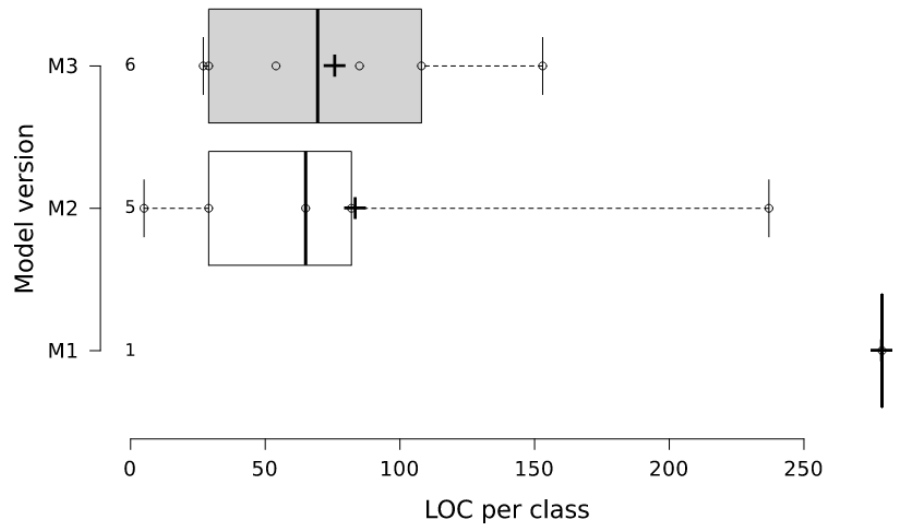

Fig. 16 Horizontal distribution of LOC across classes.

model elements (meta-class instances) for all three models. Since the populated meta-model contains all semantic information of the model, we actually extracted all string information from the models. Some of those string attributes were single character strings, while some of them contained a complete OAL implementation of a body. String attributes from each model were written in a dedicated file, one string attribute per line. This file is then used as input for further processing. Each non-alphabetic character in those files was then replaced with a space and multiple spaces are replaced with a single one. Each line is then split to tokens using space as a delimiter. Complex identifiers in all three models are obtained by concatenating capitalized words (Java naming convention). This implies that we also had to split complex tokens according to capital letters in order to get words from it. Before adding a word to a dictionary, each word is converted to lower case and then the Porter stemming algorithm [44] [45] is used to reduce it to the root English form (also known as stem). Note that we did not eliminate OAL keywords from the language because they contribute to the language of the models equally and there is only a limited number of them.

Table 5 shows the number of common words between the languages. Notice that the numbers on the diagonal of the table represent the size of the corresponding language. If we observe the size of those languages, we can see that they are almost the same in size: models 1, 2 and 3 have in total 274, 283 and 286 words, respectively. If we now compare the languages of the models and the specification language (the last row or column), we can see that there is almost no difference between the models: models 1, 2 and 3 have 121, 123 and 124 words in common with the specification language, respectively. By applying Laitinen's language theory rules[24] regarding size and similarity between the languages, we can conclude that there is no considerable difference in consistency and quality of naming conventions in three models we used in our experiment.

Table 5 Number of common words in the languages of three models and the language of the specification

	Model 1 language	**Model 2 language**	**Model 3 language**	**Specification language**
Model 1 language	**274**	230	232	**121**
Model 2 language	230	**283**	264	**123**
Model 3 language	232	264	**286**	**124**
Specification language	**121**	**123**	**124**	**235**

By minimizing the effect of other factors, we are able to examine the effect complexity distribution (modularization) has on the understandability of xtUML models. We will determine this by the experiment described in the following sections.

5.1.3 Comparing the models in terms of cyclomatic complexity

In addition to LOC, we can also compare the cyclomatic complexity distribution across bodies (table 6 and figure 17). To be able to judge cyclomatic complexity distribution across bodies we used the sum of decisions and calls within a single body (see section 4.2 for details).

Model 3 has the largest total number of decisions and calls, but it also has the best distribution (the lowest average value and standard deviation) of those decision and calls across bodies. It is interesting to note that Model 2 and Model 3 have a relatively large number of bodies without any decisions or calls. Partially, this can be explained by a large number of *getter* methods which are used to abstract away selections across relations in a class model.

Table 6 Horizontal distribution of cyclomatic complexity across bodies

	Model 1	**Model 2**	**Model 3**
Total non-empty bodies	32	67	78
Total decisions and calls	161	221	239
Average	5,03	3,30	3,06
Standard deviation	4,79	4,56	4,18
First quartile	2	0	0
Median	4	2	2
Third quartile	6,5	5	4
Max value	20	25	23
Bodies with zero decisions and calls	4	21	25

We can also analyze cyclomatic complexity distribution across classes, as shown in table 7 and figure 18. In this case, Model 2 has the lowest average cyclomatic complexity, but its distribution is worse than in Model 3 which has lower standard

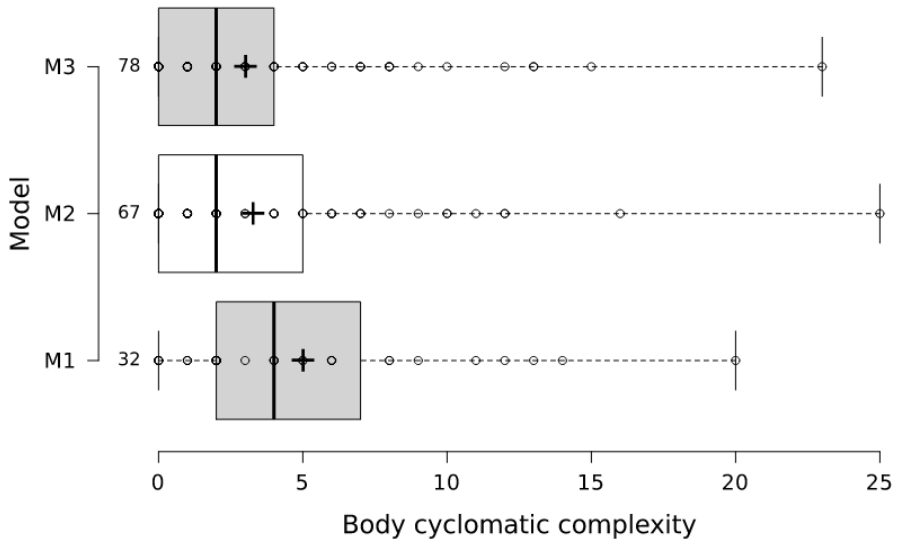

Fig. 17 Horizontal distribution of cyclomatic complexity across bodies.

Table 7 Horizontal distribution of cyclomatic complexities across classes

	Model 1	**Model 2**	**Model 3**
Complexity within classes (body + SM)	93	110	159
Total classes	1	5	6
Average	93	22	26,5
Standard deviation	0	26,34	17,47
Max value	93	65	51
Min value	93	0	1
Total complexity	136	161	208
Class-total ratio	68%	68%	76%

deviation. Model 1 has only one class and has all its complexity in one class so it obviously has the worst per-class distribution.

In addition to determining how certain model metrics are distributed among the elements of the same type – *horizontal* distribution, we can also analyze the distribution of complexity metrics across different modelling layers – *vertical* distribution of complexity. Table 8 shows how cyclomatic complexity is distributed among the four modeling layers (components, classes, state machines, and processing code) for all three model versions.

Note that all three models have the same absolute component model complexity because all three versions of the component use the same interface. Model 1 and Model 2 have low or no complexities at certain layers because they are intentionally modelled without them: Model 1 uses a single class as a wrapper for its functionality, while both Model 1 and Model 2 do not use the state machines.

The metric that best reflects the difference between vertical complexity distribution among the models is the relative cyclomatic complexity of *Graphical models* that expresses how much of the total cyclomatic complexity is visualized. It in-

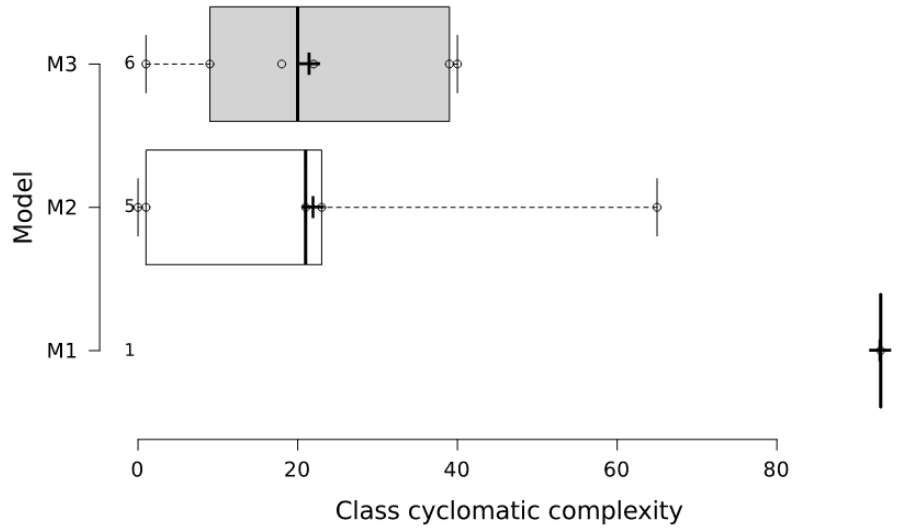

Fig. 18 Horizontal distribution of cyclomatic complexity across classes.

Table 8 Metrics for vertical complexity distribution

	Model 1		Model 2		Model 3	
	absolute	relative	absolute	relative	absolute	relative
Component model	5	3,68%	5	3,11%	5	2,40%
Class model	27	19,85%	77	47,83%	74	35,58%
State machine model (complete)	0	0%	0	0,00%	40	19,23%
State machine model (graphical)	0	0%	0	0%	30	14,42%
Processing model	131	96,32%	156	96,89%	163	78,37%
Graphical models	32	23,53%	82	50,93%	109	52,40%
TOTAL	136	100%	161	100%	208	100%

cludes cyclomatic complexity visually expressed by component, class, and state machine models. As expected, Model 3 has the largest value for this metric which indicates the best cyclomatic complexity vertical distribution.

To summarize, according to the horizontal and vertical complexity metric distributions, Model 3 has the best distribution, while Model 1 has the worst. This is done intentionally in order to observe the effect that complexity distribution has on model understandability in our experiment. This means that we intentionally limited the usage of certain types of models in a certain version. With this, we have directly affected vertical complexity distribution.

As for horizontal distribution, we strictly followed the *rule of 30* [27] for the number of LOC in action bodies for all model versions. Still, we were mildly surprised with the fact that, despite the difference in LOC, more abstract versions

of model had better *per-body* horizontal complexity distribution. It seems that new levels of abstraction affect, not only vertical complexity distribution (as is expected), but also horizontal distribution, to the extent that it compensates for the greater number of LOC. For example, in xtUML models that use class relations, it is trivial to abstract relation navigations to getter operations, at least in cases when we navigate over single-hop chains. The same getter abstraction can actually be implemented without classes, but not as easy. In addition, an xtUML state machine introduces two additional types of bodies (states and transition bodies) which also improves the horizontal, per-body, distribution.

5.2 Study subjects

The subjects of the experiment were 66 third-year, undergraduate computer science students enrolled in the Systems Analysis and Design course at the University of Split, Croatia, during the spring semester of 2015. This was their first exposure to MDE, but it is important to note that they already finished courses Software Engineering and Object-Oriented programming, in which they gained experience with the general ideas and techniques behind software modeling. In addition, before participating in the experiments, the students were exposed to the following treatment:

A. *Theoretical lecture*: the students were given a two-hour theoretical course explaining the basic ideas, principles, and assumptions behind MDE and xtUML in particular. In the end, student knowledge was tested with a short test comprised out of 20 multiple-choice, multiple-answer questions.
B. *Practical lab assignments*: in the following three weeks, the students have participated in three two-hour lab assignments which were designed to reinforce the theoretical concepts, as well as to provide the students with practical experience of creating and understanding models. They also gained practical experience working with an open-source xtUML tool – Bridgepoint[1]

In addition, during the lab assignments the students were introduced to the main requirements and the common test suite (domain knowledge), which were used for all three applications.

Since we used three structurally different applications, we have divided the students into three groups, where each student group was assigned to one application. When creating these groups, we had to assess student abilities, in order to create groups with equal student abilities. For this reason, each student was tested with a:

A. *Theoretical test*: After the theoretical lecture, each student was given a short test with 20 multiple-choice questions.
B. *Domain test*: After completing all the labs, each student was given a short domain-knowledge test with 11 multiple-choice questions.

After ordering students according to their success on the tests and their average grade, the students were randomly distributed into three groups in a way that the total ability of each group is approximately the same. The first, the second, and

[1] https://xtuml.org/download/

Table 9 Average ability and standard deviation of students in each group as measured by the tests and average grade. None of the groups show significant difference at 0.05 confidence level.

	Theory correct answers (TCA)		**Theory points (TP)**		**Domain correct answers (DCA)**		**Domain points (DP)**		**Average mark (AM)**	
	avg	std	avg	std	avg	std	avg	std	avg	std
G1	21,71	4,341	14,24	5,562	10,53	2,611	8,00	4,749	3,24	1,254
G2	21,10	4,833	14,65	7,147	12,11	1,487	10,79	2,275	3,46	0,826
G3	21,05	3,845	14,40	7,344	11,50	2,066	9,36	3,522	2,85	0,708
df bg	2		2		2		2		2	
df wg	54		54		49		49		45	
F	0,126		0,018		2,700		2,743		1,729	
p	0,882		0,983		0,077		0,074		0,189	
SGN (0,05)	NO		NO		NO		NO		NO	

the third group had 21, 20, and 25 students, respectively. The groups were not equal in size because group membership was influenced by student availability at the time slot of the experiment. To verify that there is no significant difference between the mean ability within the groups, we performed the ANOVA variance analysis for each of the ability indicators:

- Theory Correct Answers (TCA) – The number of total correct answers achieved in the theoretical xtUML exam.
- Theory Points (TP) – The number of total correct answers minus the number of incorrect answers achieved in the theoretical xtUML exam.
- Domain Correct Answers (DCA) – The number of total correct answers achieved in the domain exam.
- Domain Points (TP) – The number of total correct answers minus the number of incorrect answers achieved in the domain xtUML exam.

Table 9 shows the average ability and standard deviation of students in each group as measured by the tests and average grade. None of the ability indicators showed significant difference between the groups at $p = 0.05$ significance level.

5.2.1 Data collection

Data collection was done through online questionnaires, one per group. Questions in all three questionnaires were the same and in the same order, while the answers differed depending on the model explored by the group. Most of the questions were multiple-choice questions with multiple correct answers. Each question indicated whether there is a single or multiple correct answers. Although having different answers, questions in all three questionnaires had exactly the same number of correct answers, in exactly the same order (for example a, d and e answers were correct answers for the 6th question in all 3 questionnaires). This was done in order to minimize the differences between the questionnaires. Each questionnaire had exactly 20 questions separated in three groups:

- Model browsing: First five question are intended to get information on the ease of finding certain information in a model. For example, one of the questions

in this group required students to locate a body that needs to be altered if rounding has to be changed from 3 to 4 decimal places.
- Model understanding: The following five questions tested general understanding of the application. For example, one of the questions asked students to specify screen content after the sequence of buttons "12+=" is entered into the calculator.
- Visualized difference: The final ten questions targeted areas of application logic that are intentionally modelled differently in different models. For example, one of the questions required of students to find a body where a printout command can be added when a negative power digit is entered. Such question is relevant because the third model visualizes the process of adding digits using a state machine while the other two versions do not use state machines at all.

Table 10 shows a couple of example questions. The complete set of questions for all three groups is available at course_materials/experiment directory at the publicly available git repository [56].

Table 10 Example questions

(I) How does your implementation of calculator represent numbers (operands and results)? (Single correct answer)

a Each number is represented by one instance of class `Number` and one instance of any of the two sub-classes: `CalculatedNumber` or `EnteredNumber`.
b Each number is represented by an attribute `value` and `power` in class `Digit`.
c Each number is represented by a set of all instances of the class `Digit`.
d Each number is represented by an attribute `absoluteDigitalValue` in class `Number`.

(II) We wish to change the screen printout so that the decimal point is printed as a comma, not as a dot. Which OAL code (body) do we need to change? (Single correct answer)

a The `absoluteDigitalValue` derived attribute in class `Number`.
b The `decimalDotPressed` derived attribute in class `Screen`.
c The `decimalDotPressed()` operation in class `Screen`.
d The `content` derived attribute in class `Screen`.

(III) Which statements about numbers are correct? (Multiple correct answer)

A. If the number (an instance of class `Number`) is related to an instance of class `EnteredNumber` and is related to an operation across R4, then that number is the first operand of an operation.
B. If the number (an instance of the class `Number`) is related to an instance of class `CalculatedNumber` than it was created as a result of an operation.
C. The result of an operation cannot become the first operand of the next operation.
D. The same number (instance of the class `Number`) can be both the first and the second operand of the same operation.

(IV) We wish to print out the message each time we enter the digit of negative power. Where do we need to add a printout command? (Single correct answer)

A. In the state machine of class `EnteredNumber`, in the OAL code of the reflexive transition in state 1 (Entering non-negative power digits) with the `EnteredNumber6:addDigit(value)` event.
B. In the state machine of class `EnteredNumber` in OAL code of state 1 (Entering non-negative power digits)
C. In the state machine of class `EnteredNumber` in OAL code of state 2 (Entering negative power digits).
D. In the state machine of class `EnteredNumber` in OAL code of reflexive transition in state 2 (Entering negative power digits) which has assigned `EnteredNumber6:addDigit(value)` event.

Students were presented one question at a time, and were instructed to keep the questionnaire on the question whose answer they were trying to answer, while exploring the code base. We have collected the following data:

- Correct answers (CA): The total number of correct answers. If a question has multiple correct answers, each correct answer is taken into account.
- Incorrect answers (IA): The total number of incorrect answers. If a question is a multiple-choice question, each incorrect answer is taken into account.
- Total number of points (P): The total number of correct answers reduced by the total number of incorrect answers: $P = CA - IA$.
- Total Time (TT): The total time for completing the whole questionnaire.
- Correct answers per minute ($CAPM$): The ratio between the total number of correct answers and the total time, expressed in minutes.
- Points per minute (PPM): The ratio between the total number of points and the total time, expressed in minutes.

5.3 Experiment design summary

We have performed the experiment guided by the following research question:

- **RQ:** Is the understandability of xtUML models influenced by the distribution of cyclomatic complexity in the model?

The understandability of the three models is measured through time-relative success of the three student groups using online questionnaires. In particular, we used *the number of correct answers per minute* (CAPM) and *the number of points per minute* (PPM) to evaluate success of each student group (and understandability of corresponding version of a model). Those data indicators are selected as they consider both dimensions of model understandability: absolute success as well as the time required to finish the questionnaire.

To check for any significant differences between the groups, we performed the ANOVA variance analysis [14] [20] [9]. In order to verify the normality of the results and the applicability of ANOVA variance analysis, we used the Wilk-Shapiro normality test [52]. Since ANOVA analysis does not determine exactly between which two sets of data the difference exists, we have also performed a series of t-tests between each pair of result samples. In addition, we also calculated Cohen's d value [8] between each group pair in order to evaluate the effect size (the order of magnitude) of observed differences between the groups.

6 Experiment results

The goal of our experiment was to measure the understandability of xtUML models and to check whether there exists a relationship between the distribution of complexity of an xtUML model and its understandability. For this reason, we have set up an experiment that measures the understandability of three different models by testing students divided into three groups.

Table 11 summarizes the results of the experiment. For each group and for each data indicator the average value with the standard deviation is given. The table

Table 11 Summary of experiment results with ANOVA single factor analysis

	CA		**P**		**IA**		**TT**		**CAPM**		**PPM**	
	avg	std	avg	std	avg	std	avg	std	avg	std	avg	std
G1	20,10	4,15	13,10	7,08	7,00	3,15	36,77	6,32	0,57	0,18	0,38	0,22
G2	21,80	4,99	17,10	6,34	4,70	2,18	36,07	3,64	0,62	0,17	0,49	0,21
G3	19,36	4,22	11,92	7,45	7,44	3,57	27,20	8,56	0,78	0,28	0,46	0,35
F (2,63)	1,718		3,215		4,88		14,869		5,687		0,947	
p	0,188		0,046		0,011		0,00001		0,005		0,393	
SGN (0.05)	NO		YES		YES		YES		YES		NO	

Table 12 Cohen's d factor indicating the effect size of observed differences (S = small, M = medium and L = large effect size)

	CA		**P**		**IA**		**TT**		**CAPM**		**PPM**	
G1G2	-0,38	S	-0,61	M	**0,87**	**L**	0,14	S	-0,29	S	-0,52	M
G2G3	0,55	M	0,76	M	**-0,92**	**L**	**1,33**	**L**	-0,69	M	0,10	S
G1G3	0,18	S	0,17	S	-0,13	S	**1,28**	**L**	**-0,90**	**L**	-0,27	S

also provides the results of ANOVA statistical test for each indicator. In addition, table 12 shows the Cohen's d value calculated on each group pair for all indicators.

The results show that there is no statistically significant difference in *Correct Answers (CA)* between any of the groups. Although statistically insignificant, the average value is the highest in the second group, while the first and the third group have almost the same average and standard deviation value.

Regarding the number of *Incorrect Answers (IA)*, the second group has a significantly lower average number of incorrect answers t other two groups. The difference between the first and the third group is of small effect.

As expected from CA and IA results, the second group had a significantly higher average value for *Points* (P) and large effect size when compared to the first and the third group. The difference is significant at $p = 0.05$ level when compared to the third group, and only at $p = 0.1$ level when compared to the first group. The difference between results of the first and the third group are not significant and are of small effect size.

When we consider the average total time, we can see that students in the third group had a surprisingly shorter (25%) average total time (27 minutes when compared to 36 minutes in the other two groups). These results are significant even on $p \leq 0.01$ level and are of large effect. The difference between the first and the second group is not significant and of is small effect size.

The last two results are calculated and take into account both dimensions of the experiment results, time and absolute success. Similarly as for the total time, the third group had the highest average value of correct answers per minute (CAPM). This result is significantly different ($p \leq 0.05$) from the results of the first two groups. When compared to the first group, the effect size of the observed difference is large, and only moderate when compared to the second group. The CAPM result for the second group is not significantly greater than from the first group.

The points per minute (PPM) results do not significantly differ. However, the second and the third group have a somewhat greater average value than the first group.

To summarize the experiment results, the second group had the best absolute results of the experiment (P, IA), while the third group had the best time (TT) and time-relative results ($CAPM$). The first group was not the best (not even insignificantly) in any of the categories that we observed. Since $CAPM$ results take into account both dimensions of the model understandability (the absolute success and the time), we can conclude that students from the third group had the best overall success in the experiment.

6.1 The relation between experiment results and complexity distribution

If we compare the experiment results with the complexity distribution across models, we can conclude that the complexity metric distribution indeed has a significant effect on model understandability.

Group 1, which worked with Model 1 (the model with the worst complexity distribution) also had the worst results in the experiment. The difference in complexity distribution between Models 2 and 3 was not that emphasized, even though Model 3 had better complexity distribution. This was reflected in the results of the second and the third group. Because of these results we reject the null hypothesis and accept the alternative one: "Distribution of cyclomatic complexity significantly affects the understandability of xtUML models".

While the experiment results indicate that there exists a dependency between model understandability and model's complexity distribution, the experiment itself was not designed to reveal more details about that dependency. In order to do this, a much larger set of xtUML models is required. Traditionally, experimenters use expert opinion to order software applications of different requirements and sizes on an ordinal scale. Such experiment setup is problematic for testing our hypothesis because the functional size and complexity distribution effects are confounded. In our experiment we used only three models but, since semantically equivalent, we were able to minimize the effect of all other factors and focus on the effect of complexity distribution. Because of the low number of observed models, such experiment design prevents us to come up with any prediction model, but it enables us to reliably detect the existence of the dependency.

6.2 Threats to validity

In this section we discuss different threats to validity: *internal validity*, *external validity*, and *construct validity*.

6.2.1 Internal validity

The main threat to internal validity of the experiment is related to the *maturation* effect [6] [15]. The three groups have participated in their experiments in dedicated time slots: the first group from 8:00 to 9:30 in the morning, the second one from 9:30 and 11:00, and the third one from 11:00 till 12:30 (just before lunch). Unfortunately, such experiment setup is generally not ideal, because of potential *hunger/fatigue* effect [11] that the students of the last group may have experienced. This probably resulted in lower accuracy and shorter total time of

participants in the group. This systematic error could have been eliminated by equally distributing students from different groups in different time slots. This error is, to some extent, compensated with the fact that time is taken into account by the experiment results.

The second internal validity threat is related to the *selection* effect [4]. Although we tried to equally distribute student ability across different groups (see table 9), it was not possible to perfectly align student's availability with experiment time slots.

The third internal validity threat is related to *experimental mortality* [6]. Table 9 shows a satisfactory distribution of ability across groups, but does not take into account students that were absent from theory or domain classes (and tests) but still participated in the final experiment. It was necessary that these students participate in the experiment because the experiment was conducted in the scope of a university class, so it would be unfair to deny the learning experience to students that were not able to attend all lectures. The absence distribution is shown in table 6.2.1. When compared to other groups, we can see that group 3 had a significantly larger share of students that did not participate in the domain exercise. To check for any negative effects, we performed an additional ANOVA and t-Test analysis, this time excluding the students that did not attend the domain exercise. The results of the analysis show that this did not have any effect on the experiment results.

It is important to emphasize that all mentioned internal validity threats had a negative effect on the third group. Despite this, the third group achieved the best overall results in the experiment.

Table 13 Distribution of absence between the groups

Absent from\Group	**G1**	**G2**	**G3**	**TOTAL**
Theory	4	0	5	9
Domain	2	1	11	14
Both	1	0	3	4
Total students	21	20	25	66

It is also important to mention the co-founding effect of other understandability factors. In this research, we have measured model understandability through the number of correct answers per minute (CAPM), a metric which depends on several factors: *i*) student ability, *ii*) model's functional size, *iii*) quality of naming conventions, and according to our hypothesis, *iv*) vertical and horizontal cyclomatic complexity distribution (which depend on the used modelling approach). Since our hypothesis is dealing with cyclomatic complexity distribution, we had to make sure that the first three factors do not differ across our three case-study models. We have minimized the student ability factor by equally distributing students across groups, as described in Section 5.2. The effect of the second factor, functional size, has been minimized by taking into account the actual functional requirements that are implemented in each of the models. A test suite consisting of 30 modeled tests is used to confirm that all three versions of the model satisfy the same set of functional requirements. Except the required functionality, neither model includes any other functionality that might increase its functional size. This can be verified by downloading the models from a publicly available

repository [56]. With this we have also eliminated functional size as a factor that influences the understandability of our models. The *size* of the models can also be measured through LOC. However, in our opinion, in model-driven systems, this is not a good metric for measuring understandability. Our experiment shows that, out of three functionally equivalent models, the smallest one in terms of LOC has the smallest average CAPM (the worst understandability). This directly contradicts the premise that it could be used as a factor for model understandability. The third factor, the quality of naming conventions, used Laitinen's language theory work to compare the *languages* of all three models. As explained in Section 5.1.2, the results indicate no considerable difference between the models. We can conclude that, although the experiment is designed to minimize co-founding effects, we cannot be sure that we completely eliminated them between the models. Even with all this we do not claim that we have completely eliminated the co-founding effects, but we have made an effort to minimise them.

6.2.2 External validity

There are several concerns regarding external validity. First, the experiment subjects were third year computer science students, so the results may not be the same if the subjects were trained professionals with experience in xtUML modelling. During their previous education, students were trained in the structured programming approach (mostly used in Model 1) and class modeling approach (used in Model 2 and Model 3). However, students had little previous exposure to state machines, which were used only in the third model.

The second concern is the representativeness of models used as experiment objects. All three models are relatively simple and do not represent typical real world examples. The main reason for such model selection was the limited time required to understand the application requirements and the limited time of the experiment. The simple calculator application used in the experiment was suitable as the experiment object because it was relatively easy to add as much complexity as necessary. Since this was a relatively small application, we consider that the full abstraction potential of the xtUML methodology was not taken advantage of.

6.2.3 Construct validity

The results of the experiment show that complexity distribution affects the xtUML model understandability. However, we should be careful with the possible implications of that conclusion. In terms of vertical and horizontal complexity distribution, the three models can be ordered: Model 1, Model 2, Model 3, where Model 1 has the worst and Model 3 the best horizontal and vertical complexity distribution. However, since those models differ in horizontal as well as in vertical complexity distribution, we cannot attribute the increased understandability only to better vertical complexity distribution. This means that in our case, the horizontal and vertical complexity distribution are confounded and no definite conclusion can be made if we observe them separately. In other words, we do not know if Model 3 has the best understandability because of the new abstraction layers or simply because the largest number of bodies (subroutines) we used to modularize the processing code into. Both of those approaches are forms of abstractions, but with new layers we are introducing new types of abstractions (such as classes and state machines)

while with code modularization we are using subroutines as the only abstraction. We should also keep in mind the possibility that horizontal and vertical complexity distributions are not fully independent. As stated in the conclusion of section 5.1.3, it seems that new levels of abstraction make it easier to make abstractions on subroutine level, i.e. to detect new subroutines. A separate study and an experiment with different set of models is required to investigate isolated effects of vertical and horizontal complexity distribution to xtUMl model understandability.

7 Conclusion

In this paper we have investigated the influence of cyclomatic complexity distribution on the understandability of xtUML models. We first adapted traditional cyclomatic complexity metrics to component, class, state machine, and processing models that constitute an xtUML model. We followed this by specifying equations that calculate the integrated cyclomatic complexity of an xtUML model. We have also presented two different ways of measuring the complexity distribution: *horizontally*, among the elements of the same type, and *vertically*, among the elements of different types. In order to verify our hypothesis that cyclomatic complexity distribution has an influence on the understandability of xtUML models, we performed an experiment with student participants in which we evaluated the understandability of three semantically equivalent xtUML models with different complexity distributions. The results indicate that better distribution of cyclomatic complexity positively influences model understandability. However, since models with better understandability have better horizontal and vertical complexity distributions, the effect cannot be attributed to either of them. We speculate that horizontal and vertical distributions are not independent and that new layers of abstraction introduce and ease detection of new subroutines which reflects on the horizontal distribution of complexity.

In our opinion, there are three interesting directions for future work. The first direction is related to the problem of experiment external validity. In the research presented in this paper, we have investigated the relationship between cyclomatic complexity distribution and understandability through experiments with student participants who have interacted with relatively simple model applications. For this reason, one possible direction of future work is to perform additional experiments, but this time focused on professional developers interacting with real-world applications. The second direction for future work is related to the effects of horizontal and vertical complexity distributions on model understandability. As we already mentioned, in our current experiment models that students were able to understand better also have better horizontal and vertical distribution. For this reason, a possible direction for future work is to isolate the effects of horizontal and the effects of vertical complexity distribution on model understandability. The third direction for future work is related to deriving a regression model between cyclomatic complexity distribution and model understandability. Currently, our experiments were only able to confirm the existence of a relationship between cyclomatic complexity distribution and model understandability. By performing more experiments and deriving a regression model between these two variables, we could develop tools that could help developers develop models with higher understandability.

References

1. Abran, A., Bourque, P., Dupuis, R., Moore, J.W.: Guide to the software engineering body of knowledge-SWEBOK. IEEE Press (2001)
2. Aggarwal, K.K., Singh, Y., Chhabra, J.K.: An integrated measure of software maintainability. In: Reliability and maintainability symposium, 2002. Proceedings. Annual, pp. 235–241. IEEE (2002)
3. Albrecht, A.J.: Measuring application development productivity. In: Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, pp. 83–92 (1979)
4. Berk, R.A.: An introduction to sample selection bias in sociological data. American Sociological Review pp. 386–398 (1983)
5. Burden, H., Heldal, R., Siljamaki, T.: Executable and translatable uml–how difficult can it be? In: 2011 18th Asia-Pacific Software Engineering Conference, pp. 114–121. IEEE (2011)
6. Campbell, D.T.: Factors relevant to the validity of experiments in social settings. Psychological bulletin **54**(4), 297 (1957)
7. Chen, K., Rajlich, V.: Case study of feature location using dependence graph. In: IWPC, p. 241. Citeseer (2000)
8. Cohen, J.: Statistical power analysis for the behavioral sciences (revised ed.) (1977)
9. Cohen, Y., Cohen, J.Y.: Analysis of variance. Statistics and Data with R: An applied approach through examples pp. 463–509 (1988)
10. Cruz-Lemus, J.A., Maes, A., Genero, M., Poels, G., Piattini, M.: The impact of structural complexity on the understandability of uml statechart diagrams. Information Sciences (2010)
11. Danziger, S., Levav, J., Avnaim-Pesso, L.: Extraneous factors in judicial decisions. Proceedings of the National Academy of Sciences **108**(17), 6889–6892 (2011)
12. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. Journal of the American society for information science **41**(6), 391 (1990)
13. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process **25**(1), 53–95 (2013)
14. Fisher, R.A.: Statistical methods for research workers. Genesis Publishing Pvt Ltd (1925)
15. Fraenkel, J.R., Wallen, N.E., Hyun, H.H.: How to design and evaluate research in education, vol. 7. McGraw-Hill New York (1993)
16. Gunning, R.: The fog index after twenty years. Journal of Business Communication **6**(2), 3–13 (1969)
17. Henderson-Sellers, B., Tegarden, D.: The theoretical extension of two versions of cyclomatic complexity to multiple entrylexit modules. Software Quality Journal **3**(4), 253–269 (1994)
18. Henry, S., Kafura, D., Harris, K.: On the relationships among three software metrics. ACM SIGMETRICS Performance Evaluation Review **10**(1), 81–88 (1981)
19. Hofmann, T.: Probabilistic latent semantic indexing. In: Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval, pp. 50–57. ACM (1999)
20. Iversen, G.R., Norpoth, H.: Analysis of variance. 1. Sage (1987)
21. Khoshgoftaar, T.M., Munson, J.C.: Predicting software development errors using software complexity metrics. Selected Areas in Communications, IEEE Journal on (1990)
22. Kleppe, A.G., Warmer, J., Bast, W., Explained, M.: The model driven architecture: practice and promise (2003)
23. Labrosse, J.: Embedded software. Elsevier/Newnes, Amsterdam Boston (2008)
24. Laitinen, K.: Estimating understandability of software documents. ACM SIGSOFT Software Engineering Notes **21**(4), 81–92 (1996)
25. Lavazza, L., Robiolo, G.: Introducing the evaluation of complexity in functional size measurement: a uml-based approach. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (2010)
26. Li, W., Henry, S.: Object-oriented metrics that predict maintainability. Journal of systems and software (1993)
27. Lippert, M., Roock, S.: Refactoring in large software projects: performing complex restructurings successfully. John Wiley & Sons (2006)
28. Mattias Mohlin, I.: Modeling real-time applications in rsarte (2013). URL https://www.ibm.com/developerworks/community/wikis/home?lang=en#!/wiki/W0c4a14ff363e_436c_9962_2254bb5cbc60/page/RSARTEConcepts

29. McCabe, T.J.: A complexity measure. Software Engineering, IEEE Transactions on (1976)
30. Mellor, S.: Introduction to executable and translatable uml. Application Development Toolkit, Whitepapers CNET Networks (2005)
31. Mellor, S.J.: MDA distilled: principles of model-driven architecture. Addison-Wesley Professional (2004)
32. Mellor, S.J., Balcer, M., Foreword By-Jacoboson, I.: Executable UML: A foundation for model-driven architectures. Addison-Wesley Longman Publishing Co., Inc. (2002)
33. Mellor, S.J., Wolfe, J.R., McCausland, C.: Why systems-on-chip needs more uml like a hole in the head. In: UML for SOC Design, pp. 17–36. Springer (2005)
34. Nazir, M., Khan, R.A., Mustafa, K.: A metrics based model for understandability quantification. arXiv preprint arXiv:1004.4463 (2010)
35. Object Management Group, O.: Mda guide version 1.0 (2003). URL http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf
36. Object Management Group, O.: Model driven architecture (2016). URL http://www.omg.org/mda/
37. OMG: Omg alf standard (2013). URL http://www.omg.org/spec/ALF/
38. OMG: Semantics of a foundational subset for executable uml models (fuml) (2016). URL http://www.omg.org/spec/FUML/
39. (OMG), O.M.G.: Automated function points. http://www.omg.org/spec/AFP/1.0/ (2014)
40. OneFact: Bridgepoint tool. https://xtuml.org/download/ (2015)
41. OneFact: Bridgepoint faq (2016). URL https://github.com/xtuml/bridgepoint/blob/master/doc-bridgepoint/process/FAQ.md
42. OneFact: Bridgepoint xtuml tool (2016). URL https://www.xtuml.org/download/
43. Perisic, B.: Model driven software development-state of the art and perspectives. Invited Paper, INFOTEH pp. 1237–1248 (2014)
44. Porter, M.F.: An algorithm for suffix stripping. Program **14**(3), 130–137 (1980)
45. Porter, M.F.: The porter stemming algorithm (2006). URL https://tartarus.org/martin/PorterStemmer/
46. Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V.C.: Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. Software Engineering, IEEE Transactions on **33**(6), 420–432 (2007)
47. Raistrick, C.: Model driven architecture with executable UML, Chapter 2.7, Mapping of models,. Cambridge University Press (2004)
48. Rajlich, V., Wilde, N.: The role of concepts in program comprehension. In: Program Comprehension, 2002. Proceedings. 10th International Workshop on, pp. 271–278. IEEE (2002)
49. Riaz, M., Mendes, E., Tempero, E.: A systematic review of software maintainability prediction and metrics. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 367–377. IEEE Computer Society (2009)
50. Sarkar, S., Rama, G., Siddaramappa, N., Kak, A., Ramachandran, S.: Measuring quality of software modularization (2012). URL https://www.google.com/patents/US8146058. US Patent 8,146,058
51. Selic, B., Gullekson, G., Ward, P.T.: Real-time object-oriented modeling, vol. 2. John Wiley & Sons New York (1994)
52. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). Biometrika **52**(3/4), 591–611 (1965)
53. Shepperd, M.: A critique of cyclomatic complexity as a software metric. Software Engineering Journal **3**(2), 30–36 (1988)
54. Shibata, K., Rinsaka, K., Dohi, T., Okamura, H.: Quantifying software maintainability based on a fault-detection/correction model. In: Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on, pp. 35–42. IEEE (2007)
55. Shlaer, S.: The shlaer-mellor method. Project Technology white paper (1996)
56. Ukić, N.: Bitbucket public repository. https://bitbucket.org/nukic/phd (2016)
57. Ukić, N., Pályi, P.L., Zemljić, M., Asztalos, D., Markota, I.: Evaluation of bridgepoint model-driven development tool in distributed environment. In: Workshop on Information and Communication Technologies conjoint with 19th International Conference on Software, Telecommunications and Computer Networks, SoftCOM 2011 (2011)
58. Van Koten, C., Gray, A.: An application of bayesian network for predicting object-oriented software maintainability. Information and Software Technology **48**(1), 59–67 (2006)
59. Welker, K.D., Oman, P.W., Atkinson, G.G.: Development and application of an automated source code maintainability index. Journal of Software Maintenance: Research and Practice **9**(3), 127–159 (1997)

60. Wilde, N., Scully, M.C.: Software reconnaissance: mapping program features to code. Journal of Software Maintenance: Research and Practice **7**(1), 49–62 (1995)
61. Woodfield, S.N., Dunsmore, H.E., Shen, V.Y.: The effect of modularization and comments on program comprehension. In: Proceedings of the 5th international conference on Software engineering, pp. 215–223. IEEE Press (1981)
62. XTUML: Executable and translatable uml (2016). URL http://xtuml.org/
63. Zhou, Y., Xu, B.: Predicting the maintainability of open source software using design metrics. Wuhan University Journal of Natural Sciences **13**(1), 14–20 (2008)