Journal of Applied Technical and Educational Sciences jATES

ISSN 2560-5429

http://jates.org

jATES Journal of Applied Technical and Educational Sciences

Testing of xtUML Models across Auto-Reflexive Software Architecture

Zdeněk Havlice [1], Veronika Szabóová [2], Branislav Madoš [1]

[1] *Department of Computer Science and Informatics, Faculty of Electrical Engineering and Informatics, Košice 04011, Slovak Republic,* zdenek.havlice@tuke.sk, *branislav.mados@tuke.sk*

[2] *R-SYS s.r.o., Rybárska 7389, 911 01 Trenčín, Slovak Republic, veronika.szaboova@r-sys.sk*

Abstract: *Application of MDA in the software development enables a synchronization of the system models and corresponding source files used for the building of the executable version of a software system. Because of often use of manual modifications of some parts of code without equivalent changes in connected models, there is no guarantee that the output of the process of building of the target application will be consistent with the relevant design and implementation models. Possibility of generating of the source files from the models is a necessity, but not a sufficient condition in the process of development and modification of software systems synchronously with the changes in all related models. More safe approach is building the target application with the use of an automated building process with nested steps for consistency verifications of all critical models and related source files and the usage of model compilers. This article describes the use of xtUML and OAL for extending the software process of building the target system using special files with specification of dependencies between models and source files. Such dependencies represent the core of the critical knowledge, and it is possible to make this knowledge an integral part of the proposed new software architecture.*

Keywords: *Auto-Reflexive Software Architecture; Model Driven Architecture; Model-Driven Maintenance; xtUML; OAL; software maintenance metrics*

1. Introduction

Among the serious problems of the currently used software systems (SwS) belong~~s~~ small efficiency, reliable and conversely great expense of the processes dealing with maintenance, protection, and in particular of modifications, extensions, repairs of detected errors, innovations based on new technologies and integration with other SwS (Wuerthinger, T. et al., 2011). Methods and tools for development of effectively maintainable software systems are subject of many current research activities. The concept of sustainable software systems was introduced (Robillard, M. P., 2016). Most of the problems in these post-project processes relate to the weak or insufficient usability of knowledge about design of SwS and knowledge related to the application domain (AD), in which the SwS is used, especially after longer period of the system usage. Successful SwS is evolving and the changes in the system are making it difficult for the developers to find a suitable knowledge necessary for the future changes (Fritz, T. et al., 2014). There is current research and experimental use of many different approaches for these problems solving (Kovari 2020) e.g. based on timed automata models (Weins, D., Iftikhar, U. M., 2022), use of rule-based languages for model transformations after changes (Rodriguez-Echevaria et al., 2022).

This paper presents the way how to solve the problem of understanding large and/or complex SwS for maintenance or by other words how to develop effectively sustainable systems with use of xtUML- eXecutable Translatable UML (xtUML, 2015), OAL – Object Action Language (OAL, 2015) and ARSA - Auto-Reflexive Software Architecture (Havlice, Z., 2009, 2013, 2014), (Rajnak, B., 2015).

This problem can be solved with:

1. Preparing special file/files containing critical knowledge about SwS for their maintenance. We can consider these files as special knowledge layer (KL) - an auto-reflexive knowledge about SwS.
2. Integrating KL into software package so that each building of new version of the system will use this KL. Special architecture of SwS with a knowledge layer – ARSA can be used for this purpose.
3. Updating this layer continuously during the preparation of the new version of SwS and consistently with new version of SwS.

This paper contributes to the methods for development of sustainable SwS in the following way:

- by the proposal of extension of the MDA concept with processes of KL creation, usage and modification (KL is related to SwS design/implementation domain and SwS application domain, KL is integrated in architecture of the SwS with use of architecture ARSA),
- by the proposal of the software maintenance metrics for measuring reliability and efficiency of the software modification,
- by the case of study for suitability the above concept for maintenance of example SwS while applying ARSA with knowledge suitable for generating of the consistency tests with the use of the proposed metrics.

The goals are:

1. The usage of MDSE (Model Driven Software Engineering) including MDA (Model Driven Architecture), MDG (Model Driven Generation) and related paradigms, methods, and tools for creating models representing critical knowledge about SwS (critical models).

2. The integration of critical models into the knowledge layer in new ARSA software architecture. The usage OAL for implementation of KL, which can be used for generating consistency tests. Measurement of the rate of MDSE utilization for the integration of critical models into KL is based on measurement of the share of utilization of model transformations based on MDA, reverse engineering, forward engineering, and round-trip engineering the in the process of creating of KL and integration KL into ARSA.

3. Experimental implementing simple mobile application example with KL in ARSA for safe and effective modifications and corrections of this application. Effective modifications and corrections are based on testing and ensuring consistency of the architecture across different layers of abstraction such as models, generated source code and the resulting executable code.

4. Proposal of the metrics to measure of reliability and efficiency of software modification described in section 2.6 and use of these metrics for the assessment of our experiment with ARSA is in Section 4.5.

Besides main goals mentioned above, we also focus on the autonomic computing system development (Bocciarelli, P. et al., 2015) using ARSA. We dedicate our example application to this point of interest, and we would like to answer the following three research questions while implementing ARSA for our autonomic system instrumentation:

RQ1. Does the auto-reflection implementation with ARSA work usefully?

RQ2. Does the application of the methodology of ARSA define any limitations on the developed autonomic system?

RQ3. Do we need to change ARSA in any way to allow the autonomic system development?

Although we have used a small example of autonomic software for the experiment with ARSA, we do not focus on the methodology and the tools for development of the autonomic software systems, but we focus on the methodology and the tools for SwS development with constantly available, consistent on-line documentation containing models of critical properties of SwS that are suitable for system maintenance.

The autonomy (independence) of systems from the point of view of our design solution is limited to the autonomous use and maintenance of critical knowledge of the current version of the system by software itself (self - reflexion) in a form that allows effective and reliable maintenance.

The self - reflexion of the software system can be defined as the ability of SwS to have/ recognize the information and to be able to use the information about its own structure, behaviour, internal and external relations to improve software processes. Auto - reflexion is a necessity but not a sufficient prerequisite for the autonomy of the systems.

We do not focus on improving the testing methods, but the use of documentation testing (current models) and the system version to ensure the consistency of the system documentation and the system version in use. The consistency of system documentation and the current executable version of the system is a necessity, but not sufficient prerequisite for self-reflexing of systems (it needs implemented methods for maintaining this consistency). Our aim is to improve the quality of target SwS from viewpoint of its maintainability. We want to achieve the improving of the quality by integrating the self-reflective KL into the SwS installation package and by using implemented methods for maintaining the consistency between KL and currently used SwS. The paper is structured as follows:

Section 2 of the paper is discussing the conditions under which there is a possibility for implementation of the ARSA with suitable standards, methods, and tools. Reliability and efficiency of software processes of SwS modifications are defined and extension of the UML language is introduced via executable and translatable UML (xtUML, 2015) and Object Action Language (OAL, 2015). Weak places of the classical MDA approach are named and the knowledge - based software architecture (KBSA) that can be used to address these issues with introduction of the Knowledge Layer (KL) that can be the basis for the ARSA is proposed. The last part of this section of the article is discussing the knowledge - based approach to the software architecture (SA) and summarizes the frequency of studies which are dealing with different processes in which the knowledge - based approach is used.

Section 3 of the paper includes the description of the implementation of the knowledge layer into the ARSA with use of the xtUML, which is proposed within this research. Special attention is paid to the description of the data flows and the description of the processes which are involved in transformations of UML models into the target architectures with the use of defined sets of templates and predefined translation rules.

Section 4 of the paper describes experimental implementation of the knowledge layer within the ARSA architecture via the use of xtUML). BridgePoint as the Integrated Development Environment (xtUML, 2015) and its use in the knowledge layer generation and consistency tests generation is also described. To point out the efficiency and the reliability of ARSA process application, we present our measurement results on these properties.

Section 5 of the paper represents conclusions and describes achievements of the research and outlines its future direction.

2. Conditions for ARSA implementation

This section discusses conditions under which there is a possibility for implementation of the ARSA with suitable standards, methods, and tools.

2.1. *xtUML in MDA - Benefits*

There is possible to identify the following benefits of using xtUML in MDA (xtUML, 2015):

- separation of concerns,
- executable and testable models,
- clear and unambiguous models,
- early integration,
- component based,
- reuse,
- accelerated development life cycle,
- manageable lightweight development process,
- scalable, proven, industrial-strength process,
- resilience to change,
- 100% code generation from models,
- no redundancy,
- fully configurable code generation,
- higher quality generated software,

- reduced life cycle cost.

2.2. *Suitable standards, methods and tools*

There are three important assumptions for the effective implementation of ARSA for the use in the processes of the software systems modifications (Rajnak, B., 2015):

- Software design approach based on Model-Driven Architecture (MDA) (OMG, 2015) - based on the use of Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM). These models can be used as the resource of important knowledge about structure and behavior of software system in the process of effective and reliable re-building of new version of the system after making any modifications and/or extensions of this system.
- Executable and translatable UML suitable for simulation of modeled systems and forward engineering (Mellor, S. J. and Balcer, M. J., 2002).
- Suitable CASE tools with implemented executable and translatable UML (e. g. BridgePoint based on Eclipse with xtUML) (xtUML, 2015).

CIM allows modeling of the target system with focusing on the user requirements from viewpoint of the application domain without stating details about how to achieve expected services with algorithms, data and processes and without focusing to the required IT tools, systems and implementation details. These models can represent important knowledge about the services: information structure of the data input/output, stored (internal) data, services, and external structure of the system from the viewpoint of the users of these services (Dukan 2013). This knowledge can be used for:

- consistency testing between modeled information structure of the input, output and internal data of the services and implementation of this structure in the source files of the target system,
- generating suitable PIMs (transformation of CIM into PIM).

PIM allows for modeling the software architecture focused on algorithms, data structures, processes, restrictions, rules, and so on, which have no dependency on the target platform. These models can represent important knowledge about dynamic and static properties of the target system, expected inputs, outputs, timing, control and so on, i. e. all properties that are independent on the implementation environment. PIM therefore can be used for generating (Porkolab and Sinkovics, 2011) of the tests for verification of future implementations and for next automated transformation to suitable PSM.

PSM represents use of the platform specific models that are modeling software or business systems in connection with specific hardware or software platforms, e. g. servers with specific architectures, instruction set architectures (ISA) of microprocessors, operating systems (OS), database-management systems (DBMS), programming languages (Sinkovics and Porkolab, 2013), document file formats etc. The platform specific viewpoint provides a view of a system in which platform specific details are integrated with the elements in a PIM (France and Rumpe, 2007). PSM therefore can be seen as the PIM adapted to the specific hardware and/or software platform.

The concept of the PIM adaptation to the PSM is supplemented with the possibility of the usage of Model Transformation Languages (MTL) (Mens and Gorp, 2006; Edwards and Gruner, 2013), which allows to transform Platform Independent Models into the Platform Specific Models. The example of MTL implementation can be seen in AndroMDA framework, which is open source code generation platform (application of the AndroMDA in the development of multi-agent systems is described in (Maalal and Addou, 2011), VIATRA framework that supplies transformation language and also Event-Driven Virtual Machine (EVM) (Bergman et al., 2015), or in ATL Transformation Language (ATL) (Jouault et al., 2008; Jouault and Kurtev, 2006) that is developed by OBEO and INRIA and provides the transformation of source model sets into the target model sets. ATL is also supported by the ATL Integrated Development Environment (IDE) built on top of the Eclipse platform. In specific cases, several advantages could be achieved by using trace-based Just-In-Time (JIT) (Haeubl et al., 2014) compilation in machine code generation.

2.3. *Executable UML*

Modeling language UML itself has no tools for the simulation of the behavior of the systems described using UML models (Jouault et al., 2014). Extension based on the action semantics, that would make behavioral models (Sarjoughian et al., 2015) directly translatable and executable, can be used not only for the purpose of the simulation but also for the generating source code. The eXecutable Translatable Unified Modeling Language (xtUML) is such suitable extension of UML for more effective software processes. Action semantics is in the xtUML defined with the action language – Object Action Language (OAL) (OAL 2015). It is possible to use the OAL to specify the action semantics of such behavioral elements of UML comprising states, transitions, events etc. The xtUML models can be, according to the MDA paradigm (Mellor and Balcer, 2002; OMG, 2015), translated from PIM level into PSM and then to the source code of the target platform. This translation is based on a set of transformation rules and templates (Mellor and Balcer, 2002).

On the market, there are several tools supporting usage of xtUML in the above-described processes of software development. One of them is the BridgePoint tool (xtUML, 2015), which one is an integrated development environment including xtUML editor, verifier of models, model compiler, and the possibility to define semantics of models and to extend the process of compiling with use of OAL.

2.4. MDA and the modification of the software systems

Classical approach applying MDA (see Fig. 1) has two weak places from the viewpoint of reliability and efficiency of the software process oriented to modification, extension and/or correction of the existing version of the software system in the maintenance phase of the software life cycle (Mezghani et al., 2013; Lahiani and Bennouar, 2015):

1. Some modifications of the models on the higher level of abstraction, which are affecting other model(s) and/or parts and are extending these influences, cannot be realized automatically. Well-done consistency verification and finding of all problematic places in the models and the source files is required in this situation. For example: inserting of new methods and/or attributes into the class without their use (some artifact is defined, but not used), extending of use-case diagram with new use-cases without implementing them with use of some communications and/or processes (some artifact is defined, but not implemented) and so on.
2. Some modifications of the models on the lower level of abstraction are realized sometimes or source files are changed directly in some programming language, without doing any consequent changes in the dependent models. These models will become useless in next modifications of the target system after such inconsistent modifications of the models.

These weak places have influence on growing the gap between the models (documentation of the system) and target application (executable system) after hand-made modification of source files and/or after changes in some models without synchronizing these changes with other dependent models (see Fig. 1).

Software architecture with layer of knowledge (knowledge layer, KL) can be named knowledge - based software architecture (KBSA) (Havlice, 2013) and can be used to address these issues. The knowledge layer in the KBSA can consist of auto-reflexive knowledge about structure, behavior, design decisions, and external context of the system itself and other critical knowledge about the system and its surroundings (Havlice, 2013; 2014; 2009; Rajnak, 2015] – this architecture will be auto - reflexive software architecture (ARSA).

KL can be integrated into the software architecture on the source code level and/or even on the executable level in the suitable coded form.

KL on the source code level can be used for testing (verifying) the consistency (Khan et al., 2013) of the architecture, properties, and behavior from higher level to lower level of abstraction, between different levels of models and the source files after extensions, changes and repairs of SwS (Polák and Holubová, 2015). Critical models stored in KL are mandatory part of the install packages of the system and will be used in the make process for the building new version of the target system consistent with all critical models.

KL on the target code level (executable level) can be used for visualization of architecture of the system in run-time, localization of critical parts (modules, components, functions, objects etc.) of the currently used software system from different viewpoints (for example security risks, response time, availability of services, efficiency etc.) (Havlice, 2013).

The use of all accessible knowledge about the entire architecture and behavior of the system could be a rather difficult and not effective for the integration into ARSA. Only critical knowledge represented with the critical models therefore we can select, which have the most significant impact on the functionality of the system. This will simplify this integration process in the case of more complex architectures and can increase the efficiency and the reliability of the software maintenance processes.

2.5. *Knowledge based approaches to software architectures*

A systematic mapping study of the application of the knowledge based (KB) approaches in software architectures (SA) was conducted by Li, Liang and Avgeriou in (Li et al., 2013]. This work was based on the analysis of fifty-five studies and concluded that knowledge - based approach has a rising trend between years 2000 and 2011. According to the two previous studies (Liang and Avgeriou, 2009; Tang et al., 2010) the study classified KB approaches to the five categories comprising the Knowledge Capture and Representation (KCR) – which means the extraction of knowledge from the source codes or acquisition from stakeholders and its representation in the form that is usable by human or it is possible to process it automatically, Knowledge Reuse (KR) – which represents application of knowledge in various contexts, Knowledge Sharing (KS) – which represents sharing of knowledge in participating community, Knowledge Recovery (KRv) – which represents recodification of knowledge from tacit knowledge, and Knowledge Reasoning (KRs) – which represents production of new knowledge by the process of conclusions making and deriving new knowledge via inference. They considered all these studies according to the particular knowledge - based approach to the software architecture as it is shown in Fig. 2.

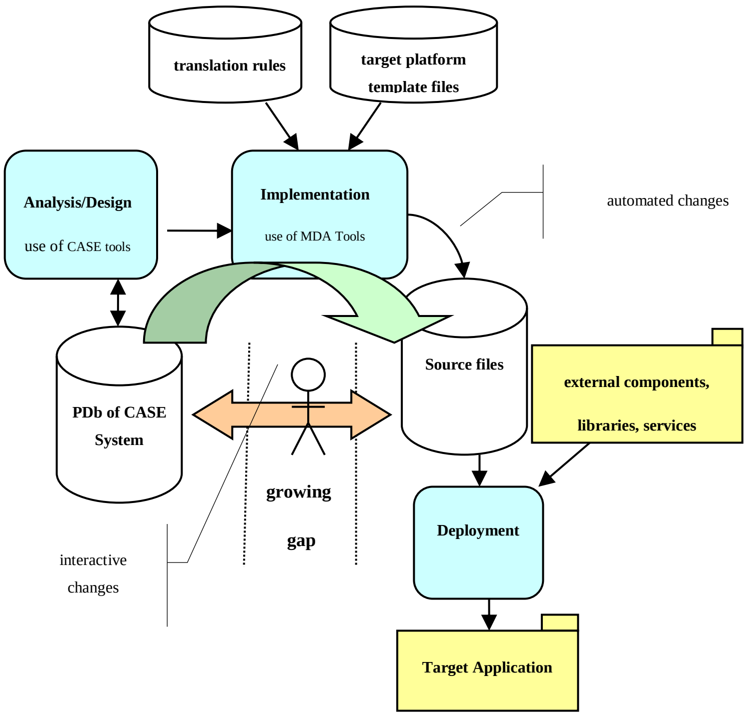

Fig. 1. Classical MDA approach in software life cycle with use of CASE system (Havlice, 2013)

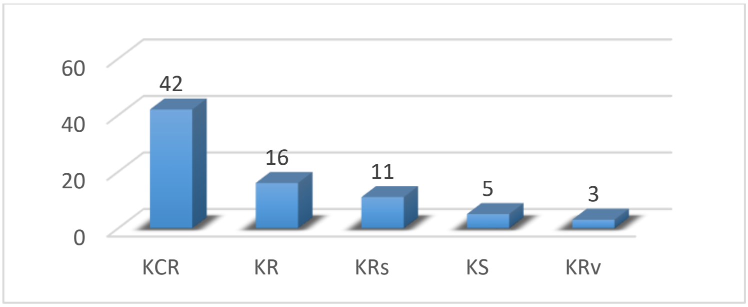

Fig. 2 Number of studies out of fifty-five considered according to the particular knowledge - based approach to the software architecture (Li et al., 2013)

Software life cycle based on ARSA includes application of these knowledge-based approaches:

KCR - critical models are coded and stored in KL,

KR - knowledge stored in KL is reused in model driven maintenance (MDM), in processes of model driven monitoring (MDMon) and model driven modifying (MDMod) - see Section 3 for evidence,

KRs - production of new knowledge based on actual knowledge stored in KL and results of MDMod of system,

KS - sharing of knowledge stored in KL in participating community.

2.6. Proposal of metrics for reliability and efficiency of software modification

The possibility of using automated transformations between the 3 levels of models CIM-PIM-PSM on code generating is very important for reliability and efficiency in software engineering processes of modification of SwS based on MDA.

It is possible to define the reliability of software process of modification of SwS (R_m) with the use of the following formula:

$$R_m = N_m / (N_m + e * N_{ni}) * 100 \text{ [\%]} \qquad (1)$$

where:

N_m	is the number of modified artifacts of SwS,
N_{ni}	is the number of artifacts of SwS negatively influenced with realized modification,
e	is coefficient of elaborateness of corrections for all negatively influenced artifacts.

Artifact of SwS in the above-described formula is any structural or behavioral part of SwS on the same level of abstraction used in N_m and N_{ni} (it can be for example service from viewpoint of the user, class from viewpoint of design, or line of code in programming language from the viewpoint of implementation).

The coefficient of elaborateness of corrections for all negatively influenced artifacts is an empirical value, which can be discovered by analyzing the software development processes, namely comparing the number of the identified negative influences and the real ones.

The reliability of modification is 100%, if the modification of specified artifact of target system has no negative influence for properties and behavior of any other artifacts of the system. Reliability of modification process declines with the increasing number of induced new errors.

Efficiency of software process of modification of SwS (E_m) we can simply define like this:

$$E_m = N_{am} / (N_{mm} + N_{am}) * 100 \text{ [\%]} \qquad (2)$$

where:

N_{am} is number of automated modifications of artifacts of SwS with use of generating code from models,

N_{mm} is number of artifacts of SwS, which need manual modifications realized by programmer.

Artifact of SwS in this formula can be the same as in previous definition of R_m. The efficiency of modification is 100%, if the modification of the specified artifact of target system needs no additional manual modification in source files in the same and/or other artifacts. Efficiency of modification process goes down with increasing number of needed manual corrections.

3. Implementing ARSA with Use of Executable and Translatable UML

MDA approach can be used for KL integrating into ARSA on the source code level and also can be used for assembling of the target software system by using model compilers (generator engines) with automated transformations of the PIM from a higher level of abstraction to a lower level and finally to source files in suitable programming language (PSM).

This MDA approach is based on transformations of UML models across different levels of abstraction into source files and these transformations are defined by the sets of templates and predefined translation rules as shown in Fig. 3 (Havlice, 2013; Rajnak, 2015). The models are created in CASE system and stored in project database (PDb).

Such transformations processes altered by modifying rules and/or templates and can be also applied automatically with consistency checking for all critical models stored in KL on the level of source files.

Description of the dataflows in Fig.3:

1. Feedback from the monitoring of the system in maintenance processes for system & software engineers (diagnostics and error messages, warnings, indications of error states and critical conditions).

2a. New requirements (user and system requirements for corrections, changes, modifications, extensions, integrating).

2b. Information about the inconsistency found between critical models (CIM, PIM) (dataflow 3a) and implementations (PSM) (dataflow 4a).

3a. Critical CIM and PIM models from KL (source level).

3b. New CIM and PIM models modified and synchronized between each other and with the implementation level.

4a. Implementation of the software system (critical PSMs, source files, components, services, libraries).

4b. New implementation of the software system (critical PSMs, source files, components, services, libraries).

5. Feedback from the monitoring of the system in maintenance process - input data for the automated modification of models and implementations in maintenance process (coded data from diagnostic and error messages, warnings, indications of error states and critical conditions).

6a. Additional information about critical CIM and PIM from PDb.

6b. Actualized additional information about critical CIM and PIM from PDb.

7a. New CIM and PIM models - input for generating PSM (transformations CIM-PIM-PSM).

7b. New generated PSM.

8. Critical CIM and PIM models from KL (executable level) for visualization in runtime.

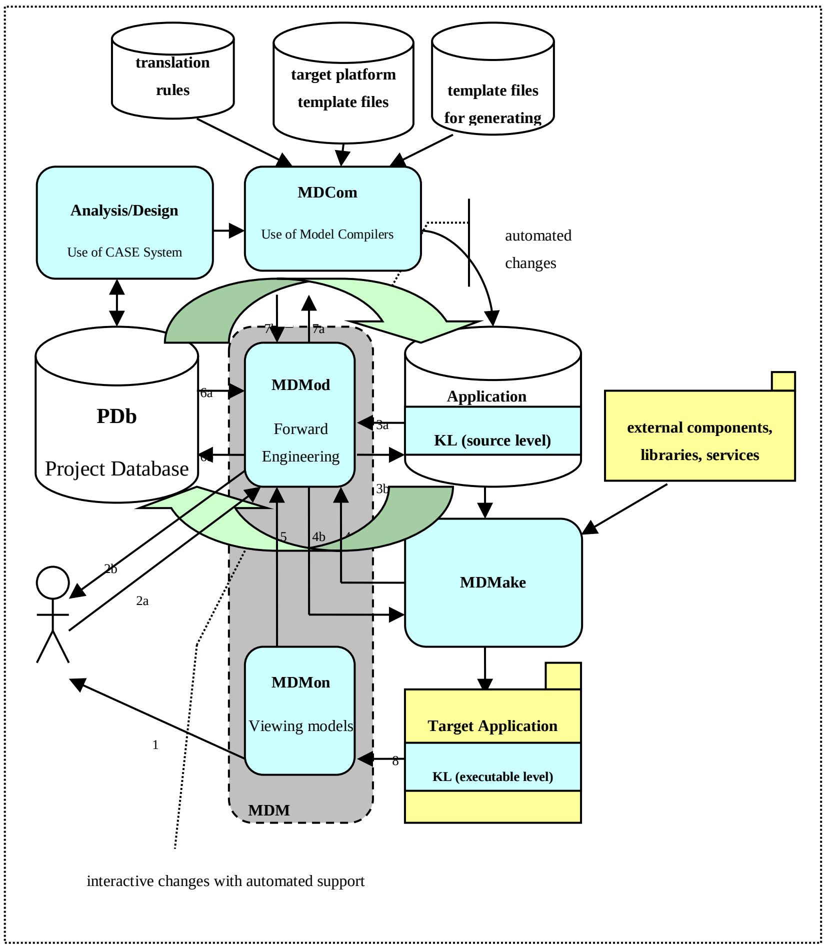

Fig. 3 Generating and use of ARSA based on MDA and executable UML (Havlice, 2013)

Description of the processes in Fig.3:

Model Driven Maintenance (MDM) - software process which use~~s~~ suitable analysis /design/ implementation models, which are synchronized with the maintained system. These models could be a representation of the concentrated knowledge about the system, and they can be used for improving of the maintenance activities (Kunstar et al., 2009a).

Consistency Testing I (CTestI) - testing existing dependences between the models (CIM, PIM, PSM) in the same level of abstraction and between the models in the different level of abstraction (if transformation between them was realized not exclusively with model compilers).

Consistency Testing II (CTestII) - testing existing dependences between critical models (PSM) stored in KL and source files (if transformation between them was realized not only solely and exclusively with model compilers).

Model Driven Modifying (MDMod) - modification of artifacts of target system with use of models and model compilers. Modifications are realized by top-down approach from models on the highest level of abstraction to models on the lower level of abstraction.

Model Driven Monitoring (MDMon) - use of models from KL and their visual representation for monitoring critical aspects of monitored system.

Model Driven Compiling (MDCom) - use of model compilers for transformation between CIM-PIM-PSM, for Consistency Testing I, for generating source files from PSM and for generating of KL on the source level.

Model Driven Make (MDMake) - assembling target application with use of model compilers for generating source files from PSM, with use of tests for Consistency Testing II and consistency testing between dependent source files, object files, libraries, services, use of compilers of programming languages and linkers for linking and with generating of KL on the executable level.

4. Experimental Implementation of ARSA

BridgePoint is an integrated development environment including xtUML editor, verifier of models, model compilers (xtUML, 2015) and possibility to define semantics of models and to extend process of compiling with use of OAL. This environment was used as suitable CASE system for experimenting with KL for ARSA.

For the different kinds of existing SwS properties, we found the following options:

- Structural properties of SwS can be modeled in xtUML with use of

  - component diagram (COMPD) - for modeling components and their interfaces
  - class diagram (CLAD) - for modeling classes of objects existing in the components of the system.
- Behavioral properties of SwS can be modeled with use of state machines
  - state transition diagrams (STAD) for classes of objects with transitions between states based on signals or events.

These three types of diagrams were used for modeling the critical knowledge about the structure and the behavior of the demo SwS. These diagrams were coded into KL with use ofXMI. This KL was integrated with all source files of the demo into one package.

4.1. Preparation

We specified the following details during the experiment preparation:

1. Storing knowledge in text form has helped automate the processing of all the facts, as structured by the native output of the compiler used. The XML format used by the generator of the source code of the model was used to compare two successive models, such that differences in the case of a simple mechanism for self-reflection notify a warning that changes have been made in the system since the last compilation. For completeness, notices are accompanied by all the details of the changes found.
2. The process of generating the system overwrites the old files in the directory specified for the new system, therefore, as by (Rajnak, 2015) we added custom directory and modified scripts designed by the author of (Rajnak, 2015) to operate and serve their functionality according to our expectations. The change was especially important because the original scripts were not recognized by the current version of the development system.
3. Although we had several choices such as txtUML (Dévai et al., 2014; Gregorics et al., 2015), it was chosen the development system, which consists of the development environment BridgePoint xtUML, whose core (Eclipse) was supplemented with additional plug-ins for requirements representation on the developed system.
4. We had to add to the directory structure of files OAL model compiler scripts that represent logic and share knowledge creating layers and consistency tests.
5. Consistency check is based on the definition of critical models as UML component diagrams, class diagrams and state transition diagrams.

4.2. Knowledge Layer Generation

Template for generating KL in XMI is part of templates of the KL Generator. One part of all algorithms of the KL Generator implemented in OAL and representing generating XMI only for all components from COMPD is described in Fig. 4.

The short source code in Fig. 4 is aimed to create the XMI files of the KL, which ones are the basis of consistency test generation. As these exist in parallel to the consistency tests, we consider them as one copy of total two instances of the KL. This redundancy is used to increase the safety of the KL.

```
.// GENERATING XMI KL for ALL COMPONENTS
.select many o_components from instances of C_C
   .// class of componets in xtUML metamodel has identifier C_C
.for each o_component in o_components
    <UML:Component name="${o_component.Name}"
xmi:id="${o_component.Id}">
    .select many ports related by o_component->C_PO[R4010]
    .if (not_empty ports)
      .for each port in ports
        .select many interfaces related by port->C_IR[R4016]->C_I[R4012]
        .for each interface in interfaces
          <UML:Interface name="${interface.Name}">
          .select    many    signals    related    by    interface->C_EP[R4003]-
>C_AS[R4004]
          .for each signal in signals
            <UML:Signal name="${signal.Name}" xmi:id="${signal.Id}" >
            </UML:Signal>
          .end for
          .select many operations related by interface->C_EP[R4003]-
>C_IO[R4004]
          .for each operation in operations
            .select any returnType related by operation->S_DT[R4008]
            <UML:Operation name="${operation.Name}"
                    returnType="${returnType.Name}" xmi:id="${operation.Id}">
            </UML:Operation>
          .end for
          </UML:Interface>
        .end for
      .end for
    .end if
    </UML:Component>
.end for
.// END OF GENERATING XMI KL for ALL COMPONENTS
```

Fig. 4 Part of OAL template for generating KL in XMI for components from COMPD (Rajnak, 2015)

To include all implementation specific assets stored within the COMPD, the above script compiles to the resulting KL OAL statements used to describe the behavior of states in state machines and to describe the actions in activity diagrams as well. An example of XMI representation of such generated KL for the demo SwS is shown in Fig. 5.

The XMI representation is re-generated at every run of the model compiler, which fact introduces a complication that the generated file cannot be used in consistency checking as the old version is already deleted and a new version is not yet present. It reduces the scope of usage of this representation on internal state visualization during software execution.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<XMI version="1.4" xmlns:uml="http://schema.omg.org/spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<XMI.header>
   <XMI.metamodel xmi.name="UML" xmi.version="2.1"/>
</XMI.header>

<XMI.content>
   <XMI:model xmi:type="uml:Model" name="Tablet" xmi:id="1">
         <UML:Component name="Battery" xmi:id="47">
            <UML:Interface name="BatteryInterface">
               <UML:Signal name="batteryUpdated" xmi:id="15" ></UML:Signal>
               <UML:Signal name="tabletOn" xmi:id="17" ></UML:Signal>
               <UML:Signal name="rechargedBattery" xmi:id="18" ></UML:Signal>
            </UML:Interface>
         </UML:Component>
         <UML:Component name="Car" xmi:id="127">
            <UML:Interface name="CarInterface">
               <UML:Signal name="connectToCar" xmi:id="20" ></UML:Signal>
               <UML:Signal name="disconnectFromCar" xmi:id="21" ></UML:Signal>
               <UML:Signal name="speedUpdated" xmi:id="22" ></UML:Signal>
               <UML:Signal name="fuelUpdated" xmi:id="24" ></UML:Signal>
            </UML:Interface>
         </UML:Component>
         <UML:Component name="GPS" xmi:id="235">
            <UML:Interface name="GPSInterface">
               <UML:Signal name="registerGPS" xmi:id="27" ></UML:Signal>
               <UML:Signal name="unregisterGPS" xmi:id="28" ></UML:Signal>
               <UML:Signal name="locationUpdated" xmi:id="29" ></UML:Signal>
            </UML:Interface>
         </UML:Component>
         <UML:Component name="MapData" xmi:id="350">
            <UML:Interface name="MapInterface">
               <UML:Signal name="registerMap" xmi:id="32" ></UML:Signal>
               <UML:Signal name="unregisterMap" xmi:id="33" ></UML:Signal>
               <UML:Signal name="mapPositionUpdated" xmi:id="34" ></UML:Signal>
               <UML:Signal name="getNewPosition" xmi:id="39" ></UML:Signal>
            </UML:Interface>
         </UML:Component>
         <UML:Component name="TabletCore" xmi:id="650">
            <UML:Interface name="CarInterface">
               <UML:Signal name="connectToCar" xmi:id="20" ></UML:Signal>
               <UML:Signal name="disconnectFromCar" xmi:id="21" ></UML:Signal>
               <UML:Signal name="speedUpdated" xmi:id="22" ></UML:Signal>
               <UML:Signal name="fuelUpdated" xmi:id="24" ></UML:Signal>
            </UML:Interface>
            <UML:Interface name="GPSInterface">
               <UML:Signal name="registerGPS" xmi:id="27" ></UML:Signal>
               <UML:Signal name="unregisterGPS" xmi:id="28" ></UML:Signal>
               <UML:Signal name="locationUpdated" xmi:id="29" ></UML:Signal>
            </UML:Interface>
            <UML:Interface name="BatteryInterface">
               <UML:Signal name="batteryUpdated" xmi:id="15" ></UML:Signal>
               <UML:Signal name="tabletOn" xmi:id="17" ></UML:Signal>
               <UML:Signal name="rechargedBattery" xmi:id="18" ></UML:Signal>
            </UML:Interface>
            <UML:Interface name="MapInterface">
               <UML:Signal name="registerMap" xmi:id="32" ></UML:Signal>
               <UML:Signal name="unregisterMap" xmi:id="33" ></UML:Signal>
               <UML:Signal name="mapPositionUpdated" xmi:id="34" ></UML:Signal>
               <UML:Signal name="getNewPosition" xmi:id="39" ></UML:Signal>
            </UML:Interface>
            <UML:Interface name="UserInterface">
               <UML:Signal name="turnOn" xmi:id="44" ></UML:Signal>
               <UML:Signal name="turnOff" xmi:id="45" ></UML:Signal>
            </UML:Interface>
         </UML:Component>
         <UML:Component name="User" xmi
```

Fig. 5 Part of generated KL for the demo SwS in XMI format

4.3. Consistency Test Generation

Template for consistency test generation CTestII was implemented in OAL and one part of all algorithms representing generating consistency test only for all components from COMPD is described in Fig. 6.

Inclusion of the template for consistency tests generation into the Model Compiler according to Fig. 8 is realized with use of file consisting of these OAL statements:

```
.include "${arc_path}/sk.tuke.dci.consistency.test-gen.main.arc"
```

```
.// GENERATING CONSISTENCY TEST for COMPONENTS
.select many old_components from instances of C_C
.for each old_component in old_components
  ..select any new_component from instances of C_C where (selected.Name == "${old_component.Name}")
..if (empty new_component)
    ..print "\nWARNING: Component '${old_component.Name}' has been changed or removed!"
  ..end if
.end for
..assign oldComponentFound = 0;
.assign i=0;
.select many old_components from instances of C_C
..select many new_components from instances of C_C
..for each new_component in new_components
  .for each old_component in old_components
   .if(i==0)
    ..if(new_component.Name == "${old_component.Name}")
      ..assign oldComponentFound = 1;
   .else
    ..elif (new_component.Name == "${old_component.Name}")
      ..assign oldComponentFound = 1;
   .end if
   .assign i=i+1;
  .end for
    ..end if
  ..if (oldComponentFound == 0)
    ..print "\nWARNING: New Component '$${new_component.Name}' Added!\n"
  ..end if
  ..assign oldComponentFound = 0;
..end for

.// END OF GENERATING CONSISTENCY TEST for COMPONENTS
```

Fig. 6 Part of OAL template for generating consistency tests CTestII for components from COMPD (Rajnak, 2015)

Consistency tests (CTestII) are also generated by an OAL script. As we mentioned it above, one can consider these tests as second and redundant version to XMI version of the KL. The significant difference is that while the XMI format is useful during execution, the CTestII file plays a significant role in software evolution. The test file is written in OAL as set of tests. Execution of the test cases is implemented within the same frame as the generation of them, using the same extension of the model compiler, the same language (OAL), the same logic of automation. Fig. 6 included an example of the XMI format of the KL, next we present the CTestII file belonging to the same project in Fig. 7. Note that both files share knowledge on object attributes.

```
.print "Testing consistency with last compiled version"
Testing consistency with last compiled version...
.print "CLASS DIAGRAM CONSISTENCY TEST START"
.//
    .select any new_component from instances of C_C where (selected.Name == "Battery")
    .if (empty new_component)
        .print "\nWARNING: Component 'Battery' has been changed or removed!"
        WARNING: Component 'Battery' has been changed or removed!
    .end if
    .select any new_component from instances of C_C where (selected.Name == "Car")
    .if (empty new_component)
        .print "\nWARNING: Component 'Car' has been changed or removed!"
        WARNING: Component 'Car' has been changed or removed!
    .end if
    .select any new_component from instances of C_C where (selected.Name == "GPS")
    .if (empty new_component)
        .print "\nWARNING: Component 'GPS' has been changed or removed!"
        WARNING: Component 'GPS' has been changed or removed!
    .end if
    .select any new_component from instances of C_C where (selected.Name == "MapData")
    .if (empty new_component)
        .print "\nWARNING: Component 'MapData' has been changed or removed!"
        WARNING: Component 'MapData' has been changed or removed!
    .end if
    .select any new_component from instances of C_C where (selected.Name == "TabletCore")
    .if (empty new_component)
        .print "\nWARNING: Component 'TabletCore' has been changed or removed!"
        WARNING: Component 'TabletCore' has been changed or removed!
    .end if
    .select any new_component from instances of C_C where (selected.Name == "User")
    .if (empty new_component)
        .print "\nWARNING: Component 'User' has been changed or removed!"
        WARNING: Component 'User' has been changed or removed!
    .end if
.assign oldComponentFound = 0;
.select many new_components from instances of C_C
.for each new_component in new_components
        .if(new_component.Name == "Battery")
            .assign oldComponentFound = 1;
        .elif (new_component.Name == "Car")
            .assign oldComponentFound = 1;
        .elif (new_component.Name == "GPS")
            .assign oldComponentFound = 1;
        .elif (new_component.Name == "MapData")
            .assign oldComponentFound = 1;
        .elif (new_component.Name == "TabletCore")
            .assign oldComponentFound = 1;
        .elif (new_component.Name == "User")
            .assign oldComponentFound = 1;
        .end if
    .if (oldComponentFound == 0)
        .print "\nWARNING: New Component '${new_component.Name}' Added!\n"
        WARNING: New Component '${new_component.Name}' Added!
    .end if
    .assign oldComponentFound = 0;
.end for
    .select any new_interface from instances of C_I where (selected.Name == "BatteryInterface")
    .if (empty new_interface)
        .print "\nWARNING: Interface 'BatteryInterface' has been changed or removed!"
        WARNING: Interface 'BatteryInterface' has been changed or removed!
    .else
        .select any new_signal related by new_interface->C_EP[R4003]->C_AS[R4004] where (selected.Name=="batteryUpdated")
        .if (empty new_signal)
            .print "\nWARNING: Signal 'batteryUpdated' in interface 'BatteryInterface' has been changed or removed!"
            WARNING: Signal 'batteryUpdated' in interface 'BatteryInterface' has been changed or removed!
        .end if
        .select any new_signal related by new_interface->C_EP[R4003]->C_AS[R4004] where (selected.Name=="tabletOn")
        .if (empty new_signal)
            .print "\nWARNING: Signal 'tabletOn' in interface 'BatteryInterface' has been changed or removed!"
            WARNING: Signal 'tabletOn' in interface 'BatteryInterface' has been changed or removed!
        .end if
        .select any new_signal related by new_interface->C_EP[R4003]->C_AS[R4004] where (selected.Name=="rechargedBattery")
        .if (empty new_signal)
            .print "\nWARNING: Signal 'rechargedBattery' in interface 'BatteryInterface' has been changed or removed!"
            WARNING: Signal 'rechargedBattery' in interface 'BatteryInterface' has been changed or removed!
        .end if
    .end if
    .select any new_interface from instances of C_I where (selected.Name == "CarInterface")
    .if (empty new_interface)
        .print "\nWARNING: Interface 'CarInterface' has been changed or removed!"
        WARNING: Interface 'CarInterface' has been changed or removed!
    .else
        .select any new_signal related by new_interface->C_EP[R4003]->C_AS[R4004] where (selected.Name=="connectToCar")
        .if (empty new_signal)
            .print "\nWARNING: Signal 'connectToCar' in interface 'CarInterface' has been changed or removed!"
            WARNING: Signal 'connectToCar' in interface 'CarInterface' has been changed or removed!
        .end if
        .select any new_signal related by new_interface->C_EP[R4003]->C_AS[R4004] where (selected.Name=="disconnectFromCar")
        .if (empty new_signal)
            .print "\nWARNING: Signal 'disconnectFromCar' in interface 'CarInterface' has been changed or removed!"
            WARNING: Signal 'disconnectFromCar' in interface 'CarInterface' has been changed or removed!
        .end if
```

Fig. 7 Part of generated consistency test CTestII

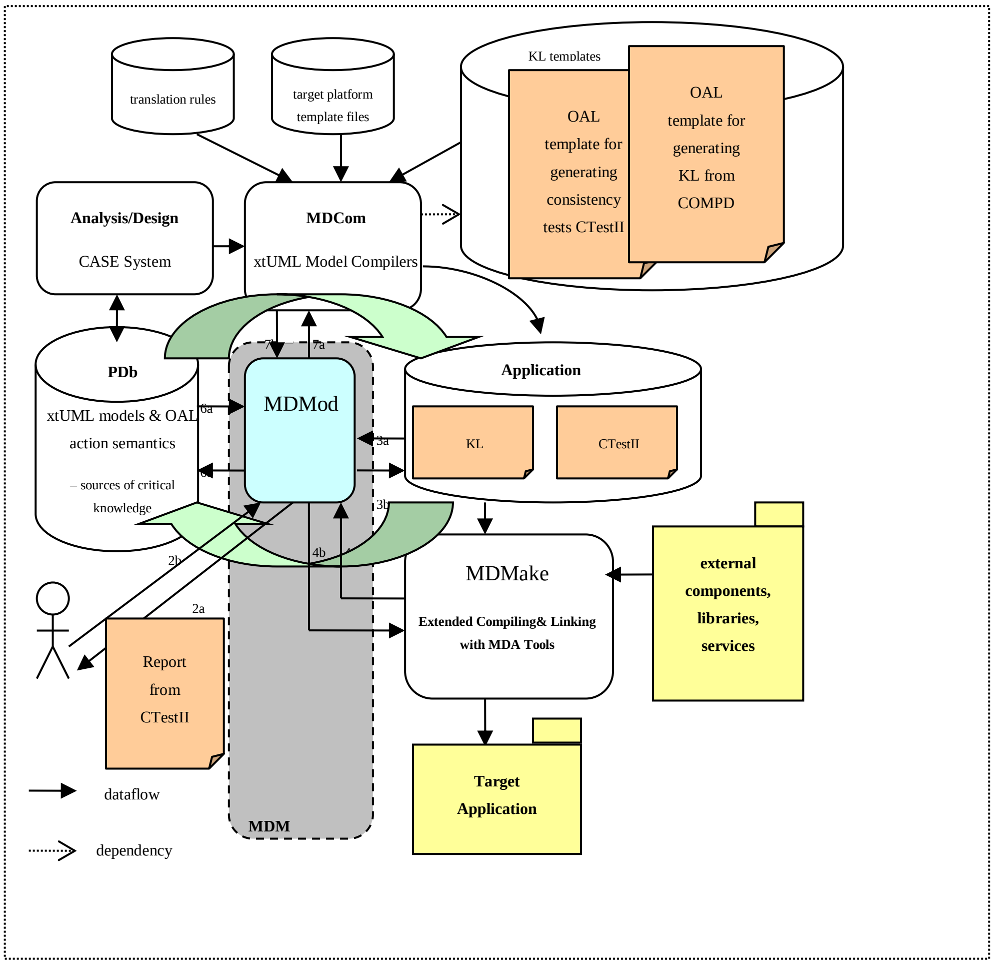
translation rules
target platform template files
KL templates
OAL template for generating consistency tests CTestII
OAL template for generating KL from COMPD
Analysis/Design
CASE System
MDCom
xtUML Model Compilers
PDb
xtUML models & OAL action semantics
– sources of critical knowledge
MDMod
Application
KL
CTestII
MDMake
Extended Compiling& Linking with MDA Tools
external components, libraries, services
Report from CTestII
Target Application
MDM
7
7a
6a
6
3a
3b
4b
2b
2a
dataflow
dependency

Fig. 8 Experimental ARSA implementation with KL on source level with use of BridgePoint and xtUML

4.4. Consistency Test Execution

Result of consistency test generation CTestII is implemented in OAL and is executed automatically when the MDG processes take place. I.e., when a new version of the system is being built from the models, the results of the generation are being confronted with the KL storing the knowledge about the previous version of the system (OAL representation of this application logic is kept in the file as presented in Fig. 10). An example output of a consistency check is presented in Fig. 9.

```
Tue Dec 13 13:02:45 2016
xtumlmc_build -home C:/BridgePoint_v5.3.4/BridgePoint/eclipse/plugins/org.xtuml.bp.mc.c.source_5.3.4/ -l3s -e -d code_generation
-O ../../src/
Upgrading translation workspace:  code_generation
Enabling detection of empty handles for component(s) *.
Action statement tracing enabled for component Battery.
Action statement tracing enabled for component Car.
Action statement tracing enabled for component GPS.
Action statement tracing enabled for component MapData.
Action statement tracing enabled for component TabletCore.
Action statement tracing enabled for component User.
Enabling state transition tracing for component(s) *.
Analyzing model and making optimizations....
22 attributes read
21 attributes written
translating control statements
translating other statements
rolling up statements into action bodies
done translating statements
NOTE: Domain code 0 allocated for this import.
NOTE: The domain code you have entered has already been used/allocated, but will allow this import to proceed normally.
sys.arc: 9:  INFO:  starting Tue Dec 13 13:02:46 2016
sk.tuke.dci.consistency.arc: 1:  INFO:
########## MODEL COMPILER CONSISTENCY TESTING EXTENSION LOADED ##########
Created by Branislav Rajnak v1.0 (c)2015
Modified by Csaba Szabo v1.1 (c)2016
Modified by Csaba Szabo and Veronika Szaboova v1.2 (c)2016

sk.tuke.dci.consistency.arc: 7:  INFO:  File '../../consistency-tests/sk.tuke.dci.consistency.output.txt'  UNCHANGED.
sk.tuke.dci.consistency.arc: 11:  INFO:  File '../../consistency-tests/sk.tuke.dci.consistency.test.main.arc'  REPLACED.
sk.tuke.dci.consistency.xmi-gen.main.arc: 1:  INFO:  Class diagram XML generation started.
sk.tuke.dci.consistency.xmi-gen.main.arc: 115:  INFO:  File 'Tablet.xmi'  CREATED.
sys.arc: 92:  INFO:  System level marking complete.
q.domain.bridges.arc: 17:  INFO:  File '_ch/LOG_bridge.h'  CREATED.
q.domain.bridges.arc: 23:  INFO:  File '_ch/LOG_bridge.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/TabletCore_TrackPoint_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/TabletCore_TrackPoint_class.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/TabletCore_TabletCore_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/TabletCore_TabletCore_class.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/MapData_MapHandler_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/MapData_MapHandler_class.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/MapData_EachMap_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/MapData_EachMap_class.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/GPS_GPS_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/GPS_GPS_class.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/Car_Car_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/Car_Car_class.c'  CREATED.
q.classes.arc: 19:  INFO:  File '_ch/Battery_Battery_class.h'  CREATED.
q.classes.arc: 24:  INFO:  File '_ch/Battery_Battery_class.c'  CREATED.
q.components.arc: 30:  INFO:  File '_ch/User.h'  CREATED.
q.components.arc: 87:  INFO:  File '_ch/User.c'  CREATED.
q.components.arc: 30:  INFO:  File '_ch/TabletCore.h'  CREATED.
q.components.arc: 74:  INFO:  File '_ch/TabletCore_classes.h'  CREATED.
q.components.arc: 87:  INFO:  File '_ch/TabletCore.c'  CREATED.
q.components.arc: 30:  INFO:  File '_ch/MapData.h'  CREATED.
q.components.arc: 74:  INFO:  File '_ch/MapData_classes.h'  CREATED.
q.components.arc: 87:  INFO:  File '_ch/MapData.c'  CREATED.
q.components.arc: 30:  INFO:  File '_ch/GPS.h'  CREATED.
q.components.arc: 74:  INFO:  File '_ch/GPS_classes.h'  CREATED.
q.components.arc: 87:  INFO:  File '_ch/GPS.c'  CREATED.
q.components.arc: 30:  INFO:  File '_ch/Car.h'  CREATED.
q.components.arc: 74:  INFO:  File '_ch/Car_classes.h'  CREATED.
q.components.arc: 87:  INFO:  File '_ch/Car.c'  CREATED.
q.components.arc: 30:  INFO:  File '_ch/Battery.h'  CREATED.
q.components.arc: 74:  INFO:  File '_ch/Battery_classes.h'  CREATED.
q.components.arc: 87:  INFO:  File '_ch/Battery.c'  CREATED.
sys.arc: 205:  INFO:  File '_ch/Tablet_sys_main.c'  CREATED.
sys.arc: 264:  INFO:  File '_ch/sys_xtuml.h'  CREATED.
sys.arc: 271:  INFO:  File '_ch/sys_xtuml.c'  CREATED.
sys.arc: 297:  INFO:  File '_ch/Tablet_sys_types.h'  CREATED.
sys.arc: 303:  INFO:  File '_ch/sys_user_co.h'  CREATED.
sys.arc: 309:  INFO:  File '_ch/sys_user_co.c'  CREATED.
sys.arc: 316:  INFO:  File '_ch/TIM_bridge.h'  CREATED.
sys.arc: 322:  INFO:  File '_ch/TIM_bridge.c'  CREATED.
sys.arc: 340:  INFO:  ending Tue Dec 13 13:02:47 2016
Code generation complete.
Tue Dec 13 13:02:47 2016
Testing consistency with last compiled version...

            Checking class 'Battery' with new model
            Checking class 'Car' with new model
            Checking class 'GPS' with new model
            Checking class 'EachMap' with new model
            Checking class 'MapHandler' with new model
            Checking class 'TabletCore' with new model
            Checking class 'TrackPoint' with new model
```

Fig. 9 Usage of KL during consistency checking

```
.print "\n########## MODEL COMPILER CONSISTENCY TESTING EXTENSION LOADED ##########\n"
.//
.// LOAD CONSISTENCY TESTS IF AVAILABLE
.//
.//
.include "../../consistency-tests/sk.tuke.dci.consistency.test.main.arc"
.emit to file "../../consistency-tests/sk.tuke.dci.consistency.output.txt"
.//
.//
.include "${arc_path}/sk.tuke.dci.consistency.test-gen.main.arc"
.emit to file "../../consistency-tests/sk.tuke.dci.consistency.test.main.arc"
.//
.//XMI generation from new models
.//
.//COMPONENT CLASSES
.include "${arc_path}/sk.tuke.dci.consistency.xmi-gen.main.arc"
.//
.//
```

Fig. 10 Autoreflexion logic including KL actualization at test and executable level of the system

4.5. Evaluation of metrics for reliability and efficiency of software modification

We measured reliability and efficiency in our experimental ARSA implementations according to the formulas (1) and (2). In our experiments, we used specific SwS, which implemented the following task:

A mobile application for car driving navigation, which automatically starts displaying gasoline/petrol pumps when the fuel level decreases to critical and, which might automatically turn off selected features when the battery level decreases to selected critical levels (levels are defined depending on the features).

Our measurement method was simple. We implemented an incremental development life cycle (Khan et al. 2015) including steps back to check reliability of the procedures for maintaining the KL in ARSA. Table 1 summarizes our results.

Based on the results presented in Tab. 1, we can evaluate the reliability metrics for each change type category applying formula (1). The average of the calculated reliabilities is 65.90%. Where, we calculated *e* based on the number of lines of code (LOC) that needed modification.

We used the scale: 0-no change, 1-very small 1-5 LOC. 2-small 6-10 LOC, 3-medium 11-15 LOC, 4-large 16-20 LOC, and 5-very large with more than 20 LOC).

Table 1: Measurement results of experimental ARSA implementation

Task description	N_m	N_{am}	N_{mm}	N_{ni}
Change in use-case diagram (non-critical model): rename use-case package	9	0	9	6
Change in the package/component diagrams (critical model): rename package	18	17	1	0
Change in the package/component diagrams (critical model): change interfaces	20	8	12	0
Change in the class state machines (critical model): rename actions or states	40	21	19	19
Change in the class state machines (critical model): change flow	19	19	0	14
Change in the sequence diagrams (critical model): rename instances	6	6	0	0
Change in the sequence diagrams (critical model): change messages	11	11	0	14
Change in the class diagrams (critical model): change interfaces	20	16	4	8
Change in the class diagrams (critical model): change data types	10	0	10	2
Change in the class diagrams (critical model): change external entities	18	15	3	4

Efficiency of software process of modification of SwS (E_m) we simply calculated based on formula (2) for the categories of changes, and then we achieved the average value for the efficiency as 65.03%.

4.6. Related Work

Presented approach (Fritz et al., 2014) for the maintaining and the use of the knowledge about implementing code of SwS for the effective maintenance of SwS is based on developer's individual knowledge of the code and their changes connected to this code. This knowledge is usually low-level knowledge of programmers based on the knowledge about the programming language and the requirements for the system implementation. This knowledge can be very useful but maintaining this kind of knowledge in consistency with code needs interactive changes with probability of human mistake.

Unlike about mentioned approach our solution based on the design models stored in KL in ARSA can use with advantage reversibility of abstract models and equivalent code (or template of code) for eliminating human mistake in coding changes and describing this change in the process of maintaining. Advantage for better understanding of code of SwS with ARSA can be also high-level abstract knowledge about SwS defined by the critical design models of the architecture, structure and behavior of SwS stored in KL and connected to the parts of the code. Combining individual knowledge of the code (Fritz et al. 2014) and the critical abstract models of SwS in the integrated knowledge layer in the architecture ARSA can be way how to do effective sustainability of SwS.

5. Conclusions

We used our mobile application example to demonstrate novel architecture ARSA implementation, with emphasis on the KL (XMI format implementation). The KL was used to report the changes in the SwS as an extension to the built-in tracking feature of the used BridgePoint utility. The counts of changes divided into change type categories were used to determine measurement values for the formulas defined in the introductory part of the paper.
From the point of evaluation of our goals mentioned in the introduction of the paper:
1. We have used model transformations based on MDA approach in KL Generator implemented in OAL for creating KL consisting of critical models of example SwS (component diagram, class diagram, state transition diagram).
2. We have used formal languages XMI, xtUML, OAL and open-source CASE tool BridgePoint for implementing of ARSA.
3. We have proposed and used metrics -formulas (1), (2) described in section 2.6 - to measure of reliability and efficiency of modification of our example software. Result of use of these metrics are in Table 1. These software maintenance metrics have also more general use for measuring reliability and efficiency of modification of software.
From the point of evaluation of our research questions mentioned in the introduction of the paper, our experiment did not reveal any necessary intervene into ARSA methodology to support the development of self-healing systems, by contrast, we found that modeling and generation of self-healing system architecture is simple using the methodology of ARSA (it is not easier without the same as ARSA). Both the substance and implementation of the navigation device have not required intervention by the proposed system because of the use of ARSA. It was only necessary to introduce more detail into each model including OAL scripts for procedural logic – action semantics objects in order to generate the target system run without refilling the generated code before the actual translation. Amendment was needed only for the implementation of communication with sensors of fuel and batteries.

We can conclude that the reliability and the efficiency of the modifications are various, and they depend on the type of the applied change.

Discovered benefits of ARSA are:

- easy to use, it needs one-time integration and addition of a new directory to the project workspace,
- no real increase in development workload as it is integrated and autonomous,
- in addition to the original tool, it adds reporting interface on changes made since the last generation of the SwS.

Discovered disadvantages are:

- severe additional work for each starting project,
- much more generator output, which might be a problem when using automated reflection tools (these would require severe modifications or extensions).

Further directions of our research include a series of larger case studies to ensure ARSA applicability, inclusion of another UML models to extend the KL (and the scope of the critical models). With the extensions, new consistency testing strategies might be needed as well. We will also focus on these consistency-checking mechanisms.

References

Bergmann, G. et al. (2015). VIATRA 3: A Reactive Model Transformation Platform, Theory and Practice of Model Transformations, Volume 9152 of the series Lecture Notes in Computer Science, pp. 101-110, ISBN: 978-3-319-21154-1 (Print) 978-3-319-21155-8 (Online).

Bocciarelli, P. et al. (2015). A Model-driven Framework for Distributed Simulation of Autonomous Systems. In: SpringSim-TMS/DEVS'15 Proc. of the Symp. on Theory of Modeling and Simulation: DEVS Integrative Modeling and Simulation Symp., April 12-15, 2015, Alexandria, VA, USA, Society for Modeling and Simulation International (SCS), San Diego, CA, USA, pp. 213-220, ISBN: 978-1-5108-0105-9.

Davis, J. E. and Chang, E. (2011). Lifecycle and generational application of automated updates to MDA based enterprise information systems. SoICT 2011. Proc. of the 2nd Symposium on Information and Communication Technology, 2011 Hanoi, Vietnam. ISBN 978-1-14503-0880-9/11/10.

Dévai, G. et al. (2014). Textual, executable, translatable UML, In: 14th International Workshop on OCL and Textual Modeling (workshop of the ACM/IEEE 17th International Conference on

Model Driven Engineering Languages and Systems MODELS 2014), Sep 28 – Oct 3, 2014, Valencia, Spain.

Dukan, P. (2013). Cloud-based smart metering system. In 2013 IEEE 14th International Symposium on Computational Intelligence and Informatics, pp. 499–502.

Edwards, C. and Gruner, S. (2013). A new tool for URDAD to Java EE EJB Transformations. In: SAICSIT'13 Proc. of the South African Institute for Computer Scientists and Information Technologies Conf., ACM, pp. 144-153, ISBN: 978-1-4503-2112-9.

France, R. and Rumpe, B. (2007). Model-Driven Development of Complex Software: A Research Roadmap, In: Future of Software Engineering 2007 at 29th International Conference on Software Engineering (ICSE), Minneapolis, pp. 37-54, IEEE, May 19-27, 2007.

Fritz, T. et al. (2014). Degree-of-knowledge: Modeling a developer's knowledge of code. ACM Transactions on Software Engineering and Methodology (TOSEM). Volume 23 Issue 2, March 2014. ACM New York, NY, USA. doi:10.1145/2512207.

Gregorics, B. et al. (2015). Textual Diagram Layout Language and Visualization Algorithm, In: ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems, MODELS 2015, Ottawa, ON, Canada, Sep 30 – Oct 2, 2015.

Haeubl, Ch. Et al. (2014). Trace Transitioning and Exception Handling in a Trace-Based JIT Compiler for Java. In ACM TRANSACTIONS ON ARCHITECTURE AND CODE OPTIMIZATION, Volume: 11, Issue: 1, Article Number: 6, DOI: 10.1145/2579673. ISSN: 1544-3566, eISSN: 1544-3973.

Havlice, Z. (2009). Knowledge-based Software Engineering. In: Computer Science and Technology Research Survey, Vol. 4, 2009, pp. 5-14, Technical University of Kosice, Faculty of Electrical Engineering and Informatics, Kosice, Slovakia.

Havlice, Z. (2013). Auto-Reflexive Software Architecture with Layer of Knowledge Based on UML Models. In: International Review on Computers and Software (IRECOS). Vol. 8, no. 8 (2013), p. 1814-1821. -ISSN 1828-6003.

Havlice, Z. at al. (2014). Knowledge-Layer Integration into Information System. In: Computer Science and Technology Research Survey: Volume 7. - Košice: TU, 2014 S. 62-67. - ISBN 978-80-553-1857-8.

Hog, Ch. E. et al. (2011). AWS-WSDL: A WSDL Extension to Support Adaptive Web Service. In: iiWAS2011, Prof. of the 13th Internat. Conf. on Information Integration and Web-based Applications and Services, ACM, pp. 477-480, ISBN: 978-1-4503-0784-0.

Jouault, F. and Kurtev, I. (2006). Transforming models with ATL. In MoDELS'05 Proceedings of the 2005 international conference on Satellite Events at the MoDELS 2005 Conference. Springer-Verlag Berlin, Heidelberg, 2006, pp. 128–138, ISBN:3-540-31780-5, 978-3-540-31780-7 doi>10.1007/11663430_14.

Jouault, F. et al. (2008). ATL: A model transformation tool. Sciences of Computer Programming, Elseiver, Vol. 72, Issues 1-2, June 2008, pp 31–39, ISSN 0167-6423.

Jouault, F. et al. (2014). fUML as an assembly Language for MDA. MiSE 2014 Proc. of the 6th Intern. Workshop on Modeling in Software Engineering, ACM, ISBM 978-1-4503-2849-4.

Khan, T. et al. (2013). eCITY: A Tool to Track Software Structural Changes using an Evolving City. In 2013 29TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), pp. 492-495. DOI: 10.1109/ICSM.2013.80.

Khan, T. et al. (2015). An Interactive Approach for Inspecting Software System Measurements. In 15th IFIP TC.13 International Conference on Human-Computer Interaction (INTERACT), Lecture Notes in Computer Science Volume: 9298, pp. 1-8, DOI: 10.1007/978-3-319-22698-9_1.

Kovari, A. (2020). Study of Algorithmic Problem-Solving and Executive Function. Acta Polytechnica Hungarica, 17(9), 241–256.

Kunstar, J. et al. (2009). Principles of model utilization in software system life cycle. In: Acta Electrotechnica et Informatica, Vol. 9, No. 3, Technical University of Kosice, Faculty of Electrical Engineering and Informatics. Kosice, 2009, pp. 48-53, ISSN 1335-8243.

Kunstar, J. et al. (2009a). The use of development models for improvement of software maintenance. In: Acta Universitatis Sapientiae, Informatica. Vol. 1, no. 1 (2009), p. 45-52. - ISSN 1844-6086.

Lahiani N. and Bennouar, D. (2015). A Model Driven Approach to Derive e-Learning Applications in Software Product Line. In: IPAC'15 Prof. of the Internat. Conf. on Intelligent Information Processing, Security and Advanced Communication, ACM, Article No. 78, 6 p., ISBN: 978-1-4503-3458-7.

Li, Z. et al. (2013). Application of knowledge-based approaches in software architecture: A systematic mapping study. Information and Software Technology 55 (2013), pp. 777-794, ISSN: 0950-5849.

Liang, P. and Avgeriou, P. (2009). Tools and technologies for architecture knowledge management. In: Ali Babar, M., Dingsøyr, T., Lago, P., H. van der Vliet, H. (Eds.); Software Architecture Knowledge Management, Springer, Berlin Heidelberg, 2009, pp. 91–111, ISBN 978-3-642-02374-3.

Maalal S. and Addou, M. (2011). A new approach of designing Multi-Agent Systems, With a practical sample, (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 2, No. 11, 2011, ISSN: 2156-5570 (Online), 2158-107X (Print), DOI: 10.14569/issn.2156-5570.

Mellor, S. J. and Balcer, M. J. (2002). Executable UML: A Foundation for Model driven Architecture. Indianapolis 2002, ISBN 0-201-74809-5, 385 p.

Mens, T. and Van Gorp, P. (2006). A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science, Elsevier Science Publishers B. V. Amsterdam, The Netherlands, vol. 152, pp. 125–142, 2006, doi>10.1016/j.entcs.2005.10.021.

Mezghani E. et al. (2013). A Model Driven Methodology for enabling Autonomic Reconfiguration of Service Oriented Architecture. In: SAC'13 Prof. of the 28th Annual ACM Symposium on Applied Computing, ACM, pp. 1772-1773, ISBN: 978-1-4503-1656-9.

OAL. (2015). Object Action Language Reference Manual. Retrieved November 28,2015 from http://www.ooatool.com/docs/OAL08.pdf

OMG. (2015). MDA - The Architecture of Choice for a Changing World. Retrieved November 28,2015 from http://omg.org/mda

Polák, M. and Holubová, I. (2015). Advanced REST API Management and Evolution Using MDA. DChanges 2015, Proc. of the 3rd Inernat. Workshop on Document Changes: modeling, storage and visualization. Lausanne, Switzerland, pp.11-18. ISBN 978-1-4503-3714-4/15/09.

Porkolab, Z. and Sinkovics, A. (2011). Domain-specific Language Integration with Compile-time Parser Generator Library. In ACM SIGPLAN NOTICES, Volume: 46, Issue: 2, pp. 137-146, DOI: 10.1145/1942788.1868315. ISSN: 0362-1340.

Rajnak, B. (2015). Use of MDA Principles and MDG Technology in Software Life Cycle. Diploma thesis. 2015. Technical University of Kosice, Faculty of Electrical Engineering and Informatics, Kosice, Slovakia.

Robillard, M. P. (2016). Sustainable Software Design. FSE 2016 Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Pages 920-923. Seattle, WA, USA — November 13 - 18, 2016. ACM New York, NY, USA. ISBN: 978-1-4503-4218-6. doi:10.1145/2950290.2983983.

Rodriguez-Echevaria et al. (2022). Suggesting model transformation repairs for rule-based languages using a contract-based testing approach. Software and Systems Modeling (SoSyM) Volume 21Issue 1Feb 2022 pp 81–112https://doi.org/10.1007/s10270-021-00891-0

ROX Software. (2005): MC-3020 Model Compiler. User's Guide. Escher Code Generator. Mentor Graphics Corporation, August 2005. Retrieved November 28,2015 from http://roxsoftware.com/ug/

Sarjoughian, H. S. et al. (2015). Behavioral DEVS Modeling. In Prof. of the 2015 Winter Simulation Conference, IEEE, pp. 2788-2799, ISBN: 978-1-4673-9743-8.

Sinkovics, A. and Porkolab, Z. (2013). Implementing monads for C plus plus template metaprograms. In SCIENCE OF COMPUTER PROGRAMMING, Volume: 78, Issue: 9, pp. 1600-1621, DOI: 10.1016/j.scico.2013.01.002. ISSN: 0167-6423.

Tang A. et al. (2010). A comparative study of architecture knowledge management tools, Journal of Systems and Software, 83 (2010) 352–370, ISSN: 0164-1212.

Wuerthinger, T. et al. (2011). Safe and Atomic Run-time Code Evolution for Java and its Application to Dynamic AOP. In ACM SIGPLAN NOTICES, Volume: 46, Issue: 10, pp. 825-844. DOI: 10.1145/2076021.2048129, ISSN: 0362-1340.

Weins, D., Iftikhar, U. M. (2022). Providing Assurances for Self-Adaptation in a Mobile Digital Storytelling Application Using ActivFORMS. ACM Transactions on Software Engineering and Methodology. https://doi.org/10.1145/3522585

xtUML.(2015). BridgePoint. Retrieved November 28,2015 from https://xtuml.org

About Authors

Zdeněk HAVLICE received his M. Sc. from Technical University of Košice, Slovakia in 1982 and PhD from Technical University of Košice, Slovakia in 1989. He is associated professor at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. His research interests include methods, tools and methodology of analysis and design of software systems, use of knowledge in software processes and architectures, modeling and prototyping of software systems, CASE systems.

Veronika SZABÓOVÁ received her M. Sc. from Technical University of Košice, Slovakia in 2012 and PhD from Technical University of Košice, Slovakia in 2016. She is a software developer at R-SYS s.r.o., Trenčín, Slovakia – Košice branch office. Her research interests include autonomic computing, software design and component-based software development for the web, especially using progressive technologies such as ReactJS with TypeScript.

Branislav MADOŠ received his M. Sc. from Technical University of Košice, Slovakia in 2006 and PhD from Technical University of Košice, Slovakia in 2009. He is an associated professor at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, Slovakia. His research interests include parallel computer architectures and architectures of computers with data driven computational model.