Computing and Informatics, Vol. 28, 2009, 843–860

TOWARDS CONFIGURABLE INTEGRATED PROJECT SUPPORT ENVIRONMENT

Vladimír Chladný, Zdeněk Havlice, Ondrej Pločica

Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice
Letná 9
041 20 Košice, Slovakia
e-mail: `Vladimir.Chladny@sk.ness.com`,
{`Zdenek.Havlice, Ondrej.Plocica`}`@tuke.sk`

Manuscript received 31 January 2006; revised 25 October 2007
Communicated by Jacek Kitowski

Abstract. This article discusses the challenge of the *Integrated Project Support Environment* development. Such an environment could be successfully applicable to a variety of software development project types. Since there are many software development approaches available it is important to choose and use an appropriate development methodology, methods and tools for a given project. The proposed Integrated Project Support Environment allows developers to configure software technology using formal specification to be used for project development.

Keywords: Integrated project support environment, software technology, software modeling, project management, software lifecycle, prototyping, code and documentation generating

1 INTRODUCTION

The area of software development is characterized particularly by rapid technological change. The contribution of software development costs to total development costs has increased to more than 70 % in several of the most important European industries. The need for further development of software engineering practices is necessary. One of the main problems of software engineering [1] is finding of adequate software technology for developing new software projects. There are critical important decisions about methodology, methods and tools, which have to be made in earlier phases of software life cycle (SwLC). These decisions are usually done on the basis of previous experiences with similar software projects.

Software engineering often involves the use of Computer Aided Software Engineering (CASE) tools or more complex CASE environments [2]. One of important problems, when adopting new methodology, is the selection (or even development) of suitable CASE tools that can successfully support it. The methodologies support systems developers in their work and the CASE tools aid the use of these methods. The methodologies without tools support are not worthy enough and they are not successfully applicable to complex projects. However, it is not economically effective to buy a new CASE tool and to re-educate staff (how to use the CASE tool) each time the methodology has been switched.

Integrated Project Support Environment (IPSE) is our proposed solution to the above problems. IPSE is CASE-based development environment, where software lifecycle, modeling methods and tools can be configured. Software technology specification is used for the CASE configuration and configured CASE is applied in software project development. Software technology specification is reusable and it can be later re-used for each solution that falls within the same software projects category too.

In the next section current trends in software development, tools and methodologies are presented. Section 3 describes usage of the proposed configurable IPSE. Section 4 describes by examples formal languages used for specification of software technology in IPSE. Software architecture of IPSE is presented in Section 5. Experimental implementation of IPSE called the *Integrated CASE system* is discussed in Section 6.

2 CURRENT TRENDS IN SOFTWARE DEVELOPMENT

Software engineering is a young discipline and is still developing. Current trends in software development include prototyping, Agile programming [3], Extreme programming [4], Unified Process development approaches [5], CleanRoom Software Engineering [6], Model-driven development [7] and there are still more traditional development approaches available.

Development tools and environments are also still developing. We can identify the following current trends in the development of CASE tools and CASE environments:

- Effort to develop different CASE tools to support selected activities within the software lifecycle. Numerous open-source projects fall within this area as well.
- Effort to develop as much suitable CASE tools to support the most popular existing methodologies as possible. This also covers the effort to reach the

integration of these CASE tools in more complex IPSE environments, which could be configured to cover all needed services in software life cycle supported by different type of CASE tools.

- Effort to develop and suggest different development approaches and methodologies based on already released CASE tools - to gain maximum from CASE tools features.
- Effort to develop Meta-CASE tools offering the features for the specification and use of customer-defined CASE tools.

Software development processes have attention as well. Major attempts to improve software results can be listed as follows [8]:

- Software Engineering Institute (SEI) [9] – founded by U.S. Department of Defense, in conjunction with Carnegie Mellon University, Pittsburgh, PA.
- SEI developed the Capability Maturity Model (CMM) [10] for assessing software capability.
- ISO 9000 [11] – International Standard Organization 9000 series standards, pertain to improvement of design, development, production activities, not just for software.
- Software Process Improvement Capability dEtermination (SPICE) [12] – an international process improvement initiative, over 40 countries involved.
- CMM, ISO 9000, SPICE – all aimed at improving the software development process.

Today's software industry is driven by intense market forces, including persistent pressure to deliver software on unrealistic time schedules. Rapid evolution of software methodologies is continuing. However, it is unable to adopt and utilize proven methodologies in timely fashion. Even though much is now known about how to improve software production, the overall state is not much better than ever, due to the urgency of meeting unrealistic delivery schedules and the continuing rapid evolution of the software industry [13, 14].

Important factors for future of the software engineering are:

- Market pressures (short production cycles, unrealistic delivery dates, shortages of skilled personnel).
- New methodologies/technologies optimized for a developed system.
- Technology transfer limitations.

3 CONFIGURATION-BASED IPSE USAGE SCENARIO

A typical scenario for using any configurable system consists of two main steps. The first step is the configuration process that is predecessing the second step, during which configured system is used. More detailed scenario for the proposed configurable IPSE usage is illustrated in Figure 1.

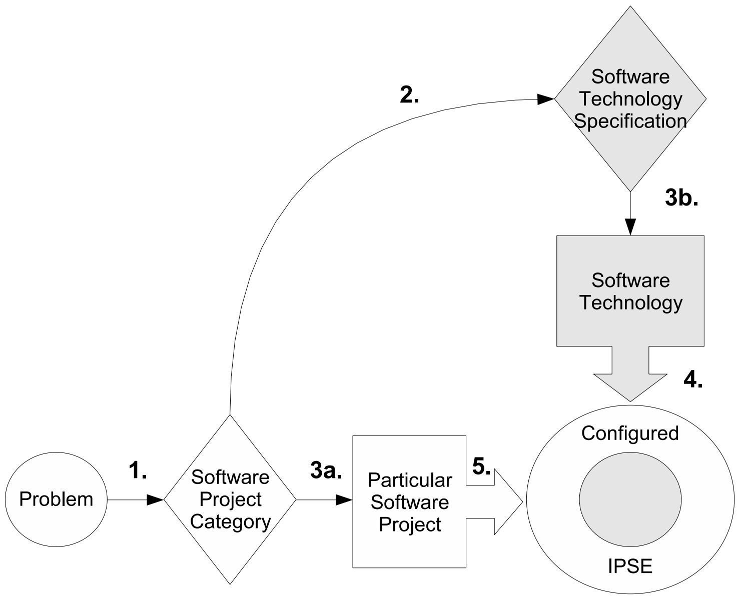
Software Technology Specification
2.
3b.
Software Technology
4.
Configured
IPSE
Problem
1.
Software Project Category
3a.
Particular Software Project
5.

Fig. 1. Configuration-based IPSE usage scenario

3.1 The Configuration Process

Before any (software) project is started, a customer comes to the problem that s/he needs to find solution for. Based on this specific problem, it is possible to identify the software project category of such solution (1). Now it is clear, which particular technologies will be applied. Suitable IPSE is needed. Rather than wasting the time by searching and testing different CASE products from different vendors, we will specify suitable software technology (2). Software technology specification (see 2.2) is reusable and it can be later re-used for each solution that falls within the same software projects category too. Particular software project can be established now (3a). Software technology specification is ready (3b) and is used for the configurable IPSE configuration (4). Finally, configured IPSE is applied on particular software project (5).

3.2 Software Technology Specification

The set of one software lifecycle model specification, zero or more tools specifications (that also cover zero or more configuration scripts for documentation generation and zero or more configuration scripts for code generation) and zero or more so-called external tools references (references to existing software tools), is called *software technology specification.* The software technology specification structure, according to UML Class diagram notation, is shown in Figure 2. This model defines basic relation between software technology components and represents definition of the software technology.

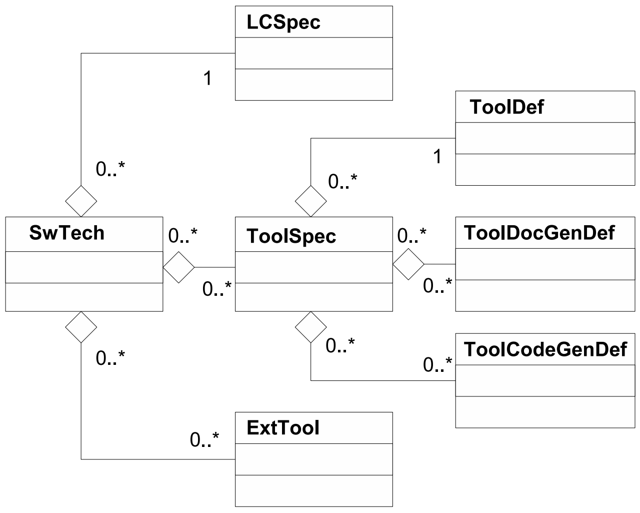

Fig. 2. The software technology specification structure

Main components of the software technology specification (SwTech) are as follows:

LCSpec	The Lifecycle Model Specification
ToolSpec	The Tool Specification
ExtTool	The Reference to External Tool
ToolDef	The Modeling Tools Definition
ToolDocGenDef	The Template for a Tools' Documentation Generator
ToolCodeGenDef	The Template for a Tools'Code Generator.

4 FORMAL TOOLS USED IN SOFTWARE TECHNOLOGY SPECIFICATION

It is necessary to use some formal tools in order to represent the software technology specification in the form suitable for IPSE configuration. The problem is that many tools that came into consideration were too complicated for our purposes. It is caused by the fact that typical user of the IPSE, during the configuration process, is expected to be rather a project manager than a formal specifications specialist.

Thus, with ease of use in mind, we have proposed two formal, but still powerful enough, languages, namely:

- Modeling Tools Description Language
- Lifecycle Configuration Language.

4.1 Modeling Tools Description Language

Modeling Tools Description Language (MTDL) is a meta-language defined primarily for dynamic configuration of the IPSE's diagrams editor (to support various notations) and for taking the part in reconfiguration of the software technology, to modify the CASE system for the best fulfilment of user's (developer's) needs [15].

MTDL is a meta-language for describing other languages (notations). It takes the role of the language, which fills the gap between formal and semiformal languages and is better understandable for CASE users, which don't have strong mathematical background and deep knowledge of formal specifications.

MTDL was constructed as "as simple as possible language", but with all needed aspects for definition of modeling tools as parts of software technologies specifications.

From the most abstract point of view we could say that MTDL is oriented for description of any Node-Oriented Diagram (tool). The built-in mechanisms allow the creator to make specifications using MTDL for definition of tools and for some experiments in this area.

Modeling tool	Diagram
Syntactic properties	types of elements (node, link) rules operations (connection, decomposition, collapse)
Semantic properties	attributes of elements rules for consistency control of operations rules for visual representation

Table 1. Properies of diagram tool

Modeling tool	Matrix (table)
Syntactic properties	dimensions - number of rows and columns
Semantic properties	attribute of columns (all items in this columns) content of headers of columns and lines sources for initialization of matrix

Table 2. Properies of matrix tool

MTDL allows specification of syntactic and semantic properties of the modeling tools (diagrams, matrices) [16]. Overview of syntactic and semantic properties for diagram and matrix modeling tool is given in Tables 1 and 2, respectively. Semantic properties of modeling tools represened by node graps are described by attributes of graph elements, connections between graph models based on decomposition and consistency rules.

Actual version of MTDL was used, and in this way successfully tested, to define the following diagrams: Entity – Relationships Diagram (ERD), Data Flow Diagram (DFD), Use Case Diagram, Class Diagram, State Transitions Diagram, Sequence Diagram, Collaboration Diagram.

Some of the main semantic characteristics of MTDL are as follows:

- MTDL enables work with sets (create set, operations with set, etc.)
- expressions can't change state of the model over which they are computed
- MTDL enables access to the attributes of modeled elements
- MTDL enables simple writing of the connection node to link and decomposition of element into diagram
- MTDL enables uniform access to elements of diagrams and to attributes of elements.

Example of DataFlow Diagram definition in MTDL:

```
// uses external definitions DataStructureDiagram, EntityRelationshipsDiagram
NODEGRAPH DataFlowDiagram                    // type and name of the model
NODES process, datastore, external_entity;   // types of nodes
LINKS flow_in, flow_out, flow_update;        // types of links
SYNONYM
        DataFlowDiagram : dfd;
        DataStructureDiagram : dsd;
        EntityRelationshipsDiagram : erd;
        external_entity : e;
        datastore : s;
        process :  p, q;
        flow_in :  fi;
        flow_out :  fo;
        flow_update :  fio;
DEFINITIONS
        f  = fi + fo + fio;   // flow can be flow_in or flow_out or flow_update
        Lw =  2;              // line width
        R  = 30;              // initial radius
        H  = 30;              // height
        W  = 40;              // width
VISUAL                 // Visual components definition (icon and initial dimensions):
        p:   CIRCLE (circle.ico, Lw, R)       // process
        s:   USR1 (usr1.ico, Lw, H, W)        // data store
        e:   RECTANGLE (rectan.ico, Lw, H, W) // external entity
        fi:  ARROW1 (arrow1.ico, Lw)          // input dataflow
        fo:  ARROW2 (arrow2.ico, Lw)          // output dataflow
        fio: ARROW3 (arrow3.ico, Lw)          // update dataflow
ATTRIBUTES             // Definition of attributes:
        dfd:    CODE id;                  // for dfd
                STRING name;
                TEXT descr;
                CHAR type, importance, level;
        f:      CODE id;                  // for dataflow
                TEXT full_name;
                STRING name;
        p:      CODE id;                  // for process
                STRING name;
                TEXT descr;
        s:      CODE id;                    // for datastore
                STRING name;
                TEXT descr;
        e:      CODE id;                    // for external entity - terminator
                STRING name;
                TEXT descr;
CONNECT                 // Definition of connections:
        Cn1: p, f, s;       // process and datastore can be connected with dataflow
        Cn2: p, f, e;       // process and terminator can be connected with dataflow
        Cn3: p, f, q;       // 2 processes can be connected with dataflow

DECOMPOSITION           // Definition of possible decompositions:
        De1: p << dfd;      // process can be decomposed into DFD
        De2: f << dsd;      // dataflow can be decomposed into DSD
        De3: s << erd;      // datastore can be decomposed into ERD
        De4: s << dsd;      // datastore can be decomposed into DSD
CONSISTENCY             // Definition of consistency rules:
        R1:     FOR Cn1, De2, De3 {
                        f << dsd . leaf_node(id, name, type) <=
                        s << erd . attrib_node(attrib_id, attrib_name, logic_type)
                }
                // leaf nodes of DSDs representing decompositions (De2) of all dataflows
                // connected to datastore (Cn1) have to be subset (<=) of attribute nodes
                // of ERD representing decomposition (De3) of connected datastore
        R2:     FOR Cn1, Cn2, Cn3, De1 {
                        p @ f @ e == p << dfd.e  &
                // external entities connected to processes on decomposition level have
                // to be the same as external entities
                // connected to decomposed prosess on higher level
                        p @ f @ s <= p << dfd.s  &
                // external datastores connected to processes on decomposition level
                // have to be the same as datastores
                // connected to decomposed prosess on higher level
                        p @ f @ q < p << dfd.p
                // external processes connected to processes on decomposition level
                // have to be the same as processes
                // connected to decomposed process on higher level
                }
        R3:     FOR Cn3 {
                        p @ fi != 0
                }
                // set of input dataflows of any process have to be not empty
        R4:     FOR Cn3 {
                        p @ fo != 0
                }
                // set of output dataflows of any process have to be not empty
END
```

4.2 Lifecycle Configuration Language

Lifecycle Configuration Language (LCCL) [15] is a software lifecycle model specification language that enables process engineers to describe software project lifecycle models in a form that can be eventually translated into a variety of representations. Using LCCL, one can describe software lifecycle models in terms of phases, activities, tasks and supporting documents and tools needed to perform the tasks. The resulting software lifecycle model specification is, together with the tools specifications in MTDL for the tools to be applied to perform tasks, a part of the software technology specification, and the resulting software technology specification can then be used to configure an IPSE to satisfy users' needs.

Without proposing the LCCL-based solution we would have to keep forcing an actual software project in the way that its specific lifecycle model would have to be omitted and some IPSE supported lifecycle model would have to be applied on it. And this "forced" way is not a correct approach. We should avoid the practice when the available tools influence a software development. Then the project solution is not reflecting the problem nature but some side factors, and this can cause a lot of trouble or an overall project failure.

The second possibility is, of course, the use of a number of different not integrated tools separately. Then we can design a project-specific lifecycle model well, but we have to take care of its interpretation ourselves. Then we lose all what the tools integration can offer.

LCCL-based solution allows both possibilities mentioned, namely the tools integration together with other IPSE features application and a software lifecycle model customization as well.

We believe that proposed LCCL is a user-friendly formal language in the way that an experienced specialist in formal specifications field is not needed for its use, and it is still expressive enough for our purposes and successfully applicable. We wanted to avoid the same mistake that caused Goodstep project [17] failure – too complicated configuration input for an (less-featured) IPSE.

LCCL script basically describes the order of phases of lifecycle model topology (we call it the "lifecycle template") and transitions between them, together with specifying the types and names of phases and types and names of transitions as well. Also the structure of phases is described – the activities covered by each phase, and deeper, the tasks covered by each activity (according to ISO/IEC 12207 [18]). It also specifies which modeling/documenting/coding/etc. tools (of IPSE itself or any external tools that are outside the IPSE) are used during execution of each phase.

Additional information that can be required to be inserted into the lifecycle model, e.g. scheduled start and finish dates for phases, descriptions/instructions for lifecycle model users in natural language, etc., can be added interactively, using a supporting tool which is a part of an IPSE.

We believe that the proposed combination of formal lifecycle model specification, together with the possibility to extend the basic lifecycle model using user-friendly tool, represents suitable solution for lifecycle model specification in real-world conditions. The basic LCCL constructions (sequence, selection, iteration, simultaneity, decomposition) are sufficient to describe standard software lifecycle models as well as specific lifecycle models.

Example of the Waterfall lifecycle model specification in LCCL:

```
///////////////////////////////////
// waterfall lifecycle model     //
// based on: ISO 12207           //
// by: Vladimir Chladny          //
///////////////////////////////////
```

```
lifecycle Waterfall {

 phase cbegin Acquisition {
    exttool MsWord, UseCase, RequisitePro;
       activity DefineSystemConcept(){
          task TakeInterviews();
          task AnalyzeInterviews();
          task DefineRationals();
       };
       activity ClarifySystemReq(){
          task ModelProblemDomain();
          task AnalyzePDModel();
          task RecordRequirements();
       };
 };

 trans AcquisitionSupply node1 = Acquisition, node2 = Supply {}

 phase Supply {
    exttool MsWord, OutlookExpress;
       activity SubmitProposal(){
          task WriteProposal();
          task SendProposalEmail();
       };
       activity NegotiateContract(){
          task AnalyzeFeedback();
          task UpdateProposal();
          task PresentProposal();
          task FinalizeProposal();
       };
 };

 trans SupplyDevelopment node1 = Supply, node2 = Development {}

 phase Development {
    exttool RationalRose, MsWord, RequisitePro, VisualC;
       activity Develop(){
          task DesignLogicalModels();
          task DesignImplemModels();
          task WriteCode();
       };
       activity Install(){
          task InstallHardware();
          task InstallSoftware();
          task CheckOperation();
       };
       activity Test(){
          task RequirementsTest();
          task AllFunctionsTest();
          task PerformanceTest();
       };
 };

 trans DevelopmentOperation node1 = Development, node2 = Operation {}

 phase Operation {
    activity OperateSystem(){};
 };

 trans OperationMaintenance node1 = Operation, node2 = Maintenance {}

 phase cend Maintenance {
    activity MaintainSystemPart(){
       task IdentifyProblem();
       task DesignPatch();
       task InstallPatch();
    };
    activity RetireSystemPart(){
       task IdentifyFault();
      task RemovePart();
    };
 };
}
```

5 THE SOFTWARE ARCHITECTURE OF CONFIGURABLE IPSE

The whole software architecture of IPSE is illustrated in Figure 3. The configurable IPSE is based on shared extended project database (ExtPDB).

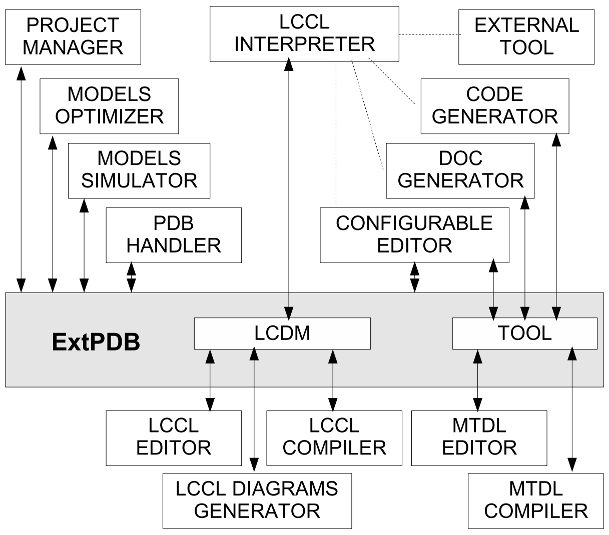

Fig. 3. The architecture of IPSE

Basic parts of architecture are:

- ExtPDB – extended project database
- TOOL – data model for ExtPDB part related to modeling tools specification
- LCDM – data model for ExtPDB part related to lifecycle models specification and use
- CONFIGURABLE EDITOR – configurable editor of diagrams

- MTDL EDITOR – editor of modeling tool specifications in MTDL
- MTDL COMPILER – MTDL scripts compiling tool
- CODE GENERATOR – code generation module
- DOC GENERATOR – documentation generation module
- EXTERNAL TOOL – module allowing external tools invocation
- LCCL EDITOR – tool for editing lifecycle model specifications in LCCL
- LCCL COMPILER – LCCL scripts compiling tool
- LCCL DIAGRAMS GENERATOR – tool for generating graphical representation of lifecycle models from LCCL scripts
- LCCL INTERPRETER – lifecycle models tool interpretation
- PROJECT MANAGER – module for project management features (e.g. team management, cost estimation)
- MODELS OPTIMIZER – module for software project quality improvement and performance tuning
- MODEL SIMULATOR – models simulation module
- PDB HANDLER – module for handling extended project database (e.g. data backup).

5.1 Extended Project Database

An extended project database of the IPSE is an important storage of software technology specification scripts necessary for its use and we call it *the software technology part of ExtPDB*. Another part of ExtPDB contains project-related data (diagrams, team management-related information, etc.) and we call it *the software project part of ExtPDB*. Simplified data model of ExtPDB is presented in Figure 4 and ExtPDB entities are described in Table 3.

The ExtPDB is divided into three parts:

1. Entities LCCL_SCRIPT and LC_TEMPLATE belong to software technology part and are used to store the lifecycle templates. Each template contains the source code in LCCL.
2. Entities EXT_TOOL and TOOL belong to software technology part as well, and are used to store the modeling tools specifications and external tools declarations. Phases of lifecycle template specifications can be related to these tools. This kind of relationship represents the fact that tools should be used during the project lifecycle phase.
3. The PROJECT, PHASE, LC_MODEL, PHASE_TYPE, TRANS, ACTIVITY and TASK entities belong to the software projects part and are used to store the actual project lifecycle model – an instance of lifecycle template.

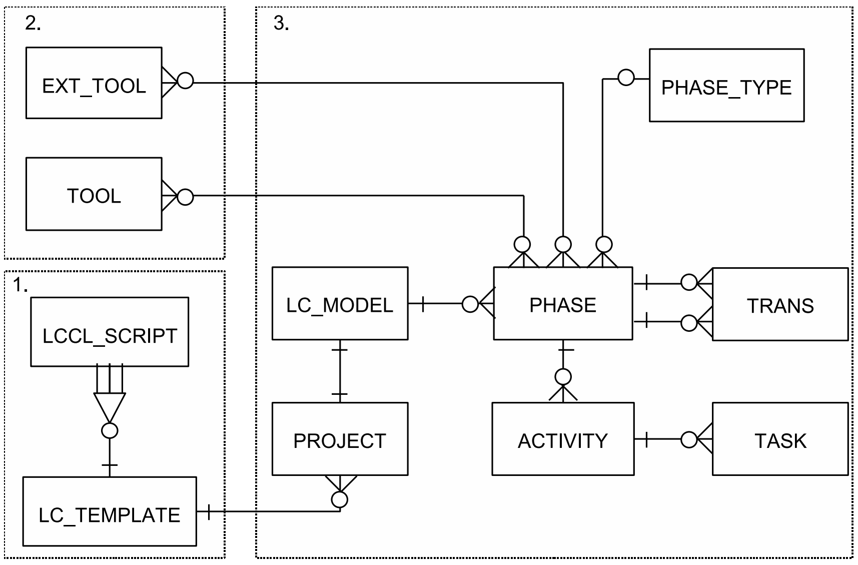

Fig. 4. Data model of ExtPDB

Entity	Description
LCCL_SCRIPT	stores the lines of LCCL script code
LCCL_TEMPLATE	stores information about lifecycle templates
PROJECT	stores information about the projects performed
LC_MODEL	represents a link between projects and their lifecycle models
TOOL	represents tools specification in MTDL
EXT_TOOL	contains reference to external tools
PHASE_TYPE	contains possible phase types
PHASE	represents phase of project lifecycle model
TRANS	represents transitions in project lifecycle model
ACTIVITY	represents an activity in project lifecycle model
TASK	represents a task in project lifecycle model

Table 3. Description of ExtPDB entities

The software technology specification represents the know-how that enables adaptation of IPSE to particular software project conditions. It is smart to store it for later reuse, once it has been specified. Later, after a new project of the same category appears, this know-how can become the advantage of its owner before competitors. The ExtPDB software technology part was designed to serve in the role of software technology storage.

Modern CASE systems usually store all of the information and data that are collected and produced during a software project such as requirements representation, analysis and design models, documentation, etc. Properly configured IPSE stores such information as well [19].

6 EXPERIMENTAL IMPLEMENTATION

In order to test and demonstrate the proposed IPSE software architecture, as well as the proposed configuration-based modeling process, we have made the experimental implementation of IPSE called the "Integrated CASE system" (InCASE). The InCASE interface that allows users to configure it for supporting software technology of their needs has been proposed. The software technology depends on the type of software project on which the IPSE is applied.

An interface has been implemented in the way of tools that user uses to write the software technology configuration scripts. The tools are able to translate these scripts into internal IPSE representation stored in shared project database. The IPSE framework uses the internal representation during its application.

The architecture of InCASE experimental implementation is presented in Figures 5 and 6. InCASE in final representation will be connected to ExtPDB and to the software system (Sw System), which is being developed (analyzed, developed and/or managed) under the given software project.

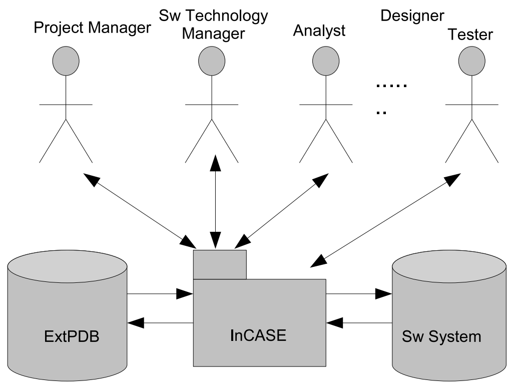

Fig. 5. Architecture of InCASE experimental implementation

The following tools of the InCASE interface have been implemented:

- LCCL editor – the tool for editing lifecycle model specifications in LCCL
- LCCL compiler – the tool for compiling LCCL scripts
- LCCL diagrams generator – the tool for generating graphical representation of lifecycle models from LCCL scripts

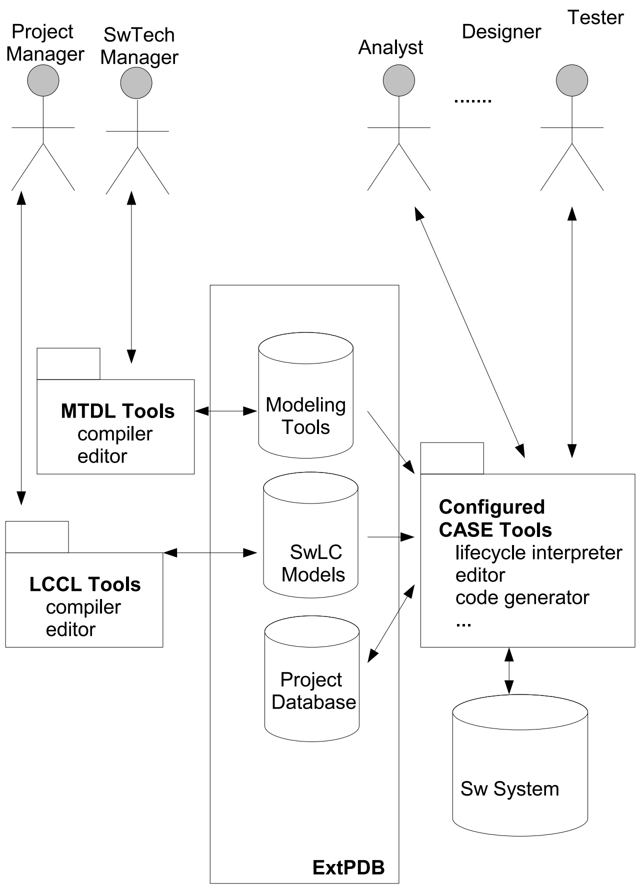

Fig. 6. Components of InCASE architecture

- MTDL editor – the tool for editing modeling tool specifications in MTDL
- MTDL compiler – the tool for compiling MTDL scripts
- template mechanism for code generator configuration
- template mechanism for documentation generator configuration.

The rest of the tools are used during project execution stages. They are:

- configurable diagrams editor (CED)
- configurable code generator
- configurable documentation generator
- configurable lifecycle model interpreter

- project management module
- project database management module
- models optimizer module
- models simulator module
- external tool invocation module.

We also proposed a set of lifecycle configuration script examples that showed how software technology repositories could be built around them as the know-how for finding solutions for development problems. Software technology can serve as template or pattern during the project works. These examples showed that the InCASE could be configured using standardized lifecycle models as well as some project-specific lifecycle models. An example of the proposed solution usage in education process during university courses was covered as well.

The set of modeling tools specification scripts has been designed as well. The CED was configured to aid design of e.g. Entity-relationship diagram, Data-flow diagram, Class Diagram, State-transition diagram, Petri net, etc.

The set of templates for code and documentation generators were created. These cover code and documentation generators for e.g. Java and C++, from the Class diagrams.

All of the mentioned InCASE modules work over the shared project database. The complete design of its data model as well as its implementation was performed too.

7 CONCLUSIONS

The experimental implementation of the proposed IPSE architecture as a prototype solution in C++ was performed successfully. The InCASE system was configured according to standard and specific methodologies as well. It was applied for self description purposes – the development process of InCASE system has been modeled. Software technology for students work has been proposed and used in educational process during university courses. The Eclipse environment [20] is considered as a suitable open source platform for future InCASE implementation.

Acknowledgement

This work was supported by VEGA 1/2176/05 – Technologies for Agent-based and Component-based Distributed Systems Lifecycle Support.

REFERENCES

[1] Somerville, I.: Software Engineering (7$^{\text{th}}$ Edition). Addison-Wesley, Menlo Park (CA), 1999.

[2] Carnegie Mellon Software Engineering Institute web site – Computer-Aided Software Engineering (CASE) Environments. Availaible on: `http://www.sei.cmu.edu/legacy/case/casewhatis.html`.

[3] Ambler, S. W.: Agile Modeling. Effective Practices for Extreme Programming and the Unified Process, John Wiley & Sons, 2002.

[4] Newkirk, J.—Martin, R. C.: Extreme Programming in Practice. Addison Wesley 2001.

[5] Kroll, P.—Kruchten, P.: The Rational Unified Process Made Easy. Addison Wesley 2004.

[6] Broadfoot, G. H.—Hopcroft, P. J.: Introducing Formal Methods Into Industry Using Cleanroom and CSP. Dedicated Systems e-Magazine, 2005.

[7] Beydeda, S.—Book, M.—Gruhn, V. (Eds.): Model-Driven Software Development. Springer 2005, XII.

[8] Mills, E.: Important Trends In Software Engineering. Global e-Business Program, CSSE Dept., Seattle University, 2003.

[9] Carnegie Mellon Software Engineering Institute Homepage. Availaible on: `http://www.sei.cmu.edu`.

[10] Carnegie Mellon Software Engineering Institute – Capability Maturity Models. Available on: `http://www.sei.cmu.edu/cmm/cmms/cmms.html`.

[11] International Organization for Standardization – ISO 9000. Available on: `http://www.iso.ch/iso/en/iso9000-14000/index.html`.

[12] Software Process Improvement and Capability Determination. Available on: `http://www.sqi.gu.edu.au/spice/`.

[13] Horváth, L.—Rudas, I. J.—Bitó, J. F.—Hancke, G.: Intelligent Computing for the Management of Changes in Industrial Engineering Modeling Processes. Computing and Informatics, Vol. 24, 2005, No. 6, pp. 549–562.

[14] Sicilia, M.—Cuadrado-Gallego, J. J.: An Algorithm for the Generation of Segmented Parametric Software Estimation Models and Its Empirical Evaluation. Computing and Informatics, Vol. 26, 2007, No. 1, pp. 1–15.

[15] Chladný, V.: Contribution to Software Systems Modeling. Dissertation thesis, Dept. of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, 2004.

[16] Havlice, Z.: The Integrated CASE System Based on the Modeling Tools Description Language. Proceedings of Scientific Conference with International Participation "Computer Engineering and Informatics", October 1999, Košice-Herľany, pp. 68–73.

[17] The GOODSTEP Project Final Report, GOODSTEP ESPRIT Project 6115, 1995.

[18] ISO/IE12207. Information Technology – Software Life Cycle Processes. ISO/IEC Copyright Office, Geneva, Switzerland, 1995.

[19] Chladný, V.: Computer Aided Software Engineering. Pre-dissertation thesis, Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice, 2001.

[20] `http://www.eclipse.org/`.

Vladimír Chladný worked at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. He received his Ph. D. degree in software andiInformation systems in 2004. His research is focused on software development methodologies, software modeling processes, modeling tools, integrated project support environments, CASE and Meta-CASE tools, software architectures. Currently he works in the Ness Slovakia Company.

Zdeněk Havlice works at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. He received his Ph. D. degree in computer science in 1991. His research is focused on computer aided software engineering methods, tools, systems and on the area of language design and compiler construction.

Ondrej Pločica works at the Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics, Technical University of Košice. His research is focused on the application of artificial intelligence methods and formal methods in Information systems development, language design and compiler construction.