

D2.4

Version: 1.0

Date: 2010-07-31

Dissemination status: PU

Document reference: D2.4



Initial implementation of a compliance language

Project acronym: COMPAS

Project name: Compliance-driven Models, Languages, and Architectures for Services

Call and Contract: FP7-ICT-2007-1

Grant agreement no.: 215175

Project Duration: 01.02.2008 – 28.02.2011 (36 months)

Co-ordinator: TUV Technische Universitaet Wien (AT)

Partners: CWI Stichting Centrum voor Wiskunde en Informatica (NL)

UCBL Université Claude Bernard Lyon 1 (FR)

USTUTT Universitaet Stuttgart (DE)

TILBURG UNIVERSITY Stichting Katholieke Universiteit Brabant (NL)

UNITN Università degli Studi di Trento (IT)

TARC-PL Telcordia Poland (PL)

THALES Thales Services SAS (FR)

PWC Pricewaterhousecoopers Accountants N.V. (NL)

This project is supported by funding from the Information Society Technologies Programme under the 7th Research Framework Programme of the European Union.





Project no. 215175

COMPAS

Compliance-driven Models, Languages, and Architectures for Services

Specific Targeted Research Project

Information Society Technologies

Start date of project: 2008-02-01 Duration: 36 months

D2.4 Initial implementation of a compliance language

Revision 1.0

Due date of deliverable: 2010-07-31

Actual submission date: 2010-07-31

Organisation name of lead partner for this deliverable:

TILBURG UNIVERSITY, Stichting Katholieke Universiteit Brabant(NL)

Contributing partner(s):

UCBL, Université Claude Bernard Lyon 1 (FR)

TARC-PL, Telcordia Poland (PL)

TUV, Vienna University of Technology (AT)

Project funded by the European Commission within the Seventh Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

History chart

Issue	Date	Changed page(s)	Cause of change	Implemented by
0.1	2010-06-29	All Sections	New document	TILBURG, UCBL, TUV, THALES, TARC-PL
0.2	2010-06-30	Section 2	Update	TILBURG
0.3	2010-07-01	Section 3	Section included	TILBURG
0.4	2010-07-06	Section 3 and 4	Update	TILBURG
0.5	2010-07-09	Section 4	Update	TILBURG
0.6	2010-07-13	Section 3	Update	TILBURG
0.7	2010-07-14	All sections	Internal review	TILBURG
0.8	2010-07-21	All sections	Partner internal Review	UCBL, TARC-PL, TILBURG
0.9	2010-07-27	Section 5 is added	Partner internal review	TUV, Tilburg
1.0	2010-07-31		Approve & Release	TUV

Authorisation

No.	Action	Company/Name	Date
1	Prepared	TILBURG	2010-06-29
2	Approved	TUV	2010-07-31
3	Released	TUV	2010-07-31

Disclaimer: The information in this document is subject to change without notice. Company or product names mentioned in this document may be trademarks or registered trademarks of their respective companies.

All rights reserved.

The document is proprietary of the COMPAS consortium members. No copying or distributing, in any form or by any means, is allowed without the prior written agreement of the owner of the property rights.

This document reflects only the authors' view. The European Community is not liable for any use that may be made of the information contained herein.

Contents

1. Introduction	5
1.1. Purpose and scope	5
1.2. Document overview	6
1.3. Definitions and glossary	7
1.4. Abbreviations and acronyms	7
2. Integration within the COMPAS Architecture	7
3. Rule Modeller Prototype Description and Design	9
3.1. Summary of the main features	11
3.2. Expression Elements	13
3.2.1. CRL meta-model for pattern expressions	14
3.2.2. Basic Expression Elements	15
3.2.3. Basic Patterns	16
3.2.4. Advanced Patterns	16
3.2.5. Boolean Operators	18
3.2.6. Scope	19
4. Example Pattern Expressions	19
5. How to Install and Use?	22
6. Reference documents	24
6.1. Internal documents	24
6.2. External documents	24

List of figures

Figure 1 Overall COMPAS architecture	9
Figure 2 Rule Modeller Components	10
Figure 3 Graphical Expression Editor	12
Figure 4 Example LTL formulas generated	13
Figure 5 CRL conceptual model for pattern expressions	14
Figure 6 Examples of pattern expressions for the scenario	20
Figure 7 RuleModeler installation wizard	22
Figure 8 New Project Window	23
Figure 9 Domain-Specific Language Designer Wizard	23

Abstract

This deliverable presents the initial implementation of the prototype D2.4 for the compliance request language. In particular, the tool supports the design of visual representations of compliance requirements using patterns and automated generation of formal compliance rules based on the compliance request language meta-model. It describes how the prototype is integrated with the COMPAS Architecture and briefly presents its main features and design. It also describes the current status of the implementation together with the technology choices.

1. Introduction

1.1. Purpose and scope

The major goal of work package 2 (WP2) is to develop a language for service user requests and provider constraints, and a language for regulatory compliance concerns, as examples of high-level compliance languages based on the concepts from WP1. This mainly involves the design of the compliance language modules and supporting infrastructure.

Deliverable 2.1 (M6) presents an overview of the state-of-the-art in compliance languages with the main focus on the languages for regulatory and legislative provisions. Mandatory features that should exist in a compliance language are also reported. One of the main findings of this study is that the compliance requirements should be based on a formal foundation of a logical language to pave the way for automatic reasoning techniques that assist in verifying business process compliance during design-time phase of the business process lifecycle.

Deliverable 2.2 (M11) examines the languages that can be used as the basic building blocks for a comprehensive Compliance Request Language (CRL) and reports a comparative analysis between these candidate formalisms (also published in [ETH+10a]). For this deliverable [D2.2], we analysed a wide range of compliance legislations and frameworks, and studied a variety of relevant works on the specification of compliance requirements. We identified structural patterns of frequently recurring (compliance) requirements imposed on the business processes (published in [ETH+10b]).

Deliverable 2.3 (M18) presents a meta-model for the CRL and proposes extensions to address the limitations identified in D2.2. The CRL integrates property specification patterns, Linear Temporal Logic (LTL) formalism and higher-level compliance concepts for the specification of business process compliance requirements for design-time. D2.3 also presents the main features and architecture of a software environment for the CRL, which is further elaborated by the successive deliverable D2.6.

Deliverable 2.6 (M23) presents the initial version of the prototype [M23] for the run-time environment (Compliance Request Language Tools – CRLT¹) and the Compliance Requirements Repository (CRR), where data maintained by the CRLT resides. CRLT enables users to:

- Define compliance requirements in various abstractions together with related aspects such as compliance risks, sources, controls and rules.

¹Compliance Request Language Tools (CRLT) web site: <http://eriss.uvt.nl/compas/>

- Formulate compliance requests during design-time of business process compliance for verifying compliance targets (mainly business processes) against defined compliance requirements.

This prototype deliverable (D2.4 –Initial implementation of a compliance language) has the following short brief in the [DoW]: “This deliverable will provide a prototype covering tools and techniques to translate service compliance specifications and associated requirements into a set of candidate service compositions specified in BPEL dynamically.” On the other hand, as discussed above, our research reported in D2.1 and D2.2 identified a need for a logical language based on a formal foundation to be used to specify compliance requirements to enable the design-time verification of process compositions specified in BPEL. Research results pointed out that the transformation of abstract compliance requirements originating from diverse compliance sources into formal compliance rules plays a key role in ensuring design-time compliance of business processes. Grounded on these research findings, the primary objective of the compliance language (which involved the translation of compliance specifications and associated requirements into service compositions) moved towards representing compliance requirements in several abstraction levels starting with the business-level compliance requirements and relevant concepts to the formal representations transformed from abstract representations, while maintaining bidirectional traceability.

In D2.6, we presented the main features of CRLT by going through ‘use cases’ describing the interaction between the user and CRLT via its user interface. One of the important features identified for the CRLT is the support for the transformation of high-level and abstract compliance requirements/controls into *formal compliance rules* (refer to [D2.6] section 3.1.8). In addition, we found out that the use of formal languages for the specification of compliance requirements creates difficulties for the end-users, particularly in terms of usability and comprehensibility. This problem is one of the main obstacles for the utilization of sophisticated verification and analysis tools associated with these languages. To help to overcome these difficulties, we adapted the property specification patterns as an integral part of CRL (refer to [D2.3] section 5). The property specification patterns (or Dwyer’s property patterns [DAC98]) are high-level abstractions of frequently used logical formulas, which help non-technical users to abstractly represent desired properties and constraints.

Consequently, we identified a need for a feature of the CRLT that allows the ‘*compliance expert*’ to *visually represent a particular compliance constraint by using property specification patterns and business process constructs (operands), and to automatically generate LTL formulas derived from these pattern-based expressions*. However, we have to point out that the target transformation logical language is not limited to LTL. Other target languages can also be incorporated (e.g. Computational tree logic- CTL [PNU77], ForSpec Temporal logic- FTL [AFF+02], etc.). This is enabled by a standalone component of the CRLT, called the ‘Rule Modeller’. This deliverable introduces the initial implementation of the prototype for the Rule Modeller. It describes its main features and design, and its integration with the other components of the COMPAS architecture.

1.2. Document overview

The remainder of this document is structured as follows: Section 2 describes the integration between the Rule Modeller and other components of the COMPAS Architecture defined in [DA.1]. Section 3 describes the overall design and implementation of the Rule Modeller including its main features. Finally, Section 4 describes the examples of compliance requirements to illustrate the use of the software application.

1.3. Definitions and glossary

The most important terminology concerning the COMPAS project is listed on the public COMPAS Web-Site [D7.1] available at <http://www.compas-ict.eu> section terminology. This helps to make the overall COMPAS approach more comprehensive for the general public.

1.4. Abbreviations and acronyms

CDT	Contrary To Duty
CRL	Compliance Request Language
CTL	Computational Tree Logic
CRLT	Compliance Request Language Tool
CRR	Compliance Requirements Repository
FTL	ForSpec Temporal Logic
LTL	Linear Temporal Logic
WP	Work package

2. Integration within the COMPAS Architecture

COMPAS introduces the notion of process fragments (in WP4) that implement compliance requirements by means of activities and control dependencies among them (please refer to D4.1). Process fragments can be integrated into a business processes with the intention of making the process compliant to a certain compliance requirement. However, business processes still requires a formal verification against these compliance requirements in order to ensure that the process fragments have been integrated in the correct manner and at the correct place, and that the overall composition complies with the requirements. To achieve that, the process model is checked against formal rules that represent the compliance requirements. (The details of the overall approach summarized here are presented in a joint paper [STK+10].)

In this scheme, CRLT serves two main purposes. It enables users to define compliance requirements together with their *formal representations*, and enables verification of the business processes by using the process verification tools (adapted by WP3). CRR stores compliance requirements in various levels of abstraction. It is integrated with the ‘Model Repository’, where models including the definitions of compliance targets; such as, business processes, business process activities, and web services, reside. The Model Repository is also integrated with the ‘Process Fragments Repository’ (cf. [D4.4]). Repositories can be accessed through a set of web services maintained by WP1.

CRLT is used to populate CRR with compliance requirements and related concepts (such as compliance sources, risks, controls and rules²) in line with the COMPAS unified conceptual model and to associate them with the compliance targets that reside in the Model Repository (as a part of MORSE central repository hosted and maintained by WP1). In addition, CRLT

²Please refer to the terminology section on the public COMPAS Web-Site [D7.1] available at <http://www.compas-ict.eu> for the definition of the terms (such as compliance target, compliance requirements, compliance rules, compliance expert, business process owner, and etc.) used in this document.

can also serve runtime verification developed by WP5 (D5.4), which is mainly concerned with protocol monitoring that uses LTL formulas as its logical foundation.

Rule Modeller –the main subject of this deliverable- is a standalone component of the CRLT. It is used to define visual representations of the controls as pattern-based expressions and to automatically generate LTL formulas based on these expressions to enable design-time compliance verification.

Figure 1 depicts the COMPAS Architecture that provides relevant prototypes, tools, and technologies, with the use cases provided by industry partners in order to demonstrate the pragmatic use of COMPAS contributions. The components enclosed in the figure represent CRLT's context together with the Rule modeller and the CRR. The next section elaborates more on the specifics of the integration between the Rule Modeller and the CRR.

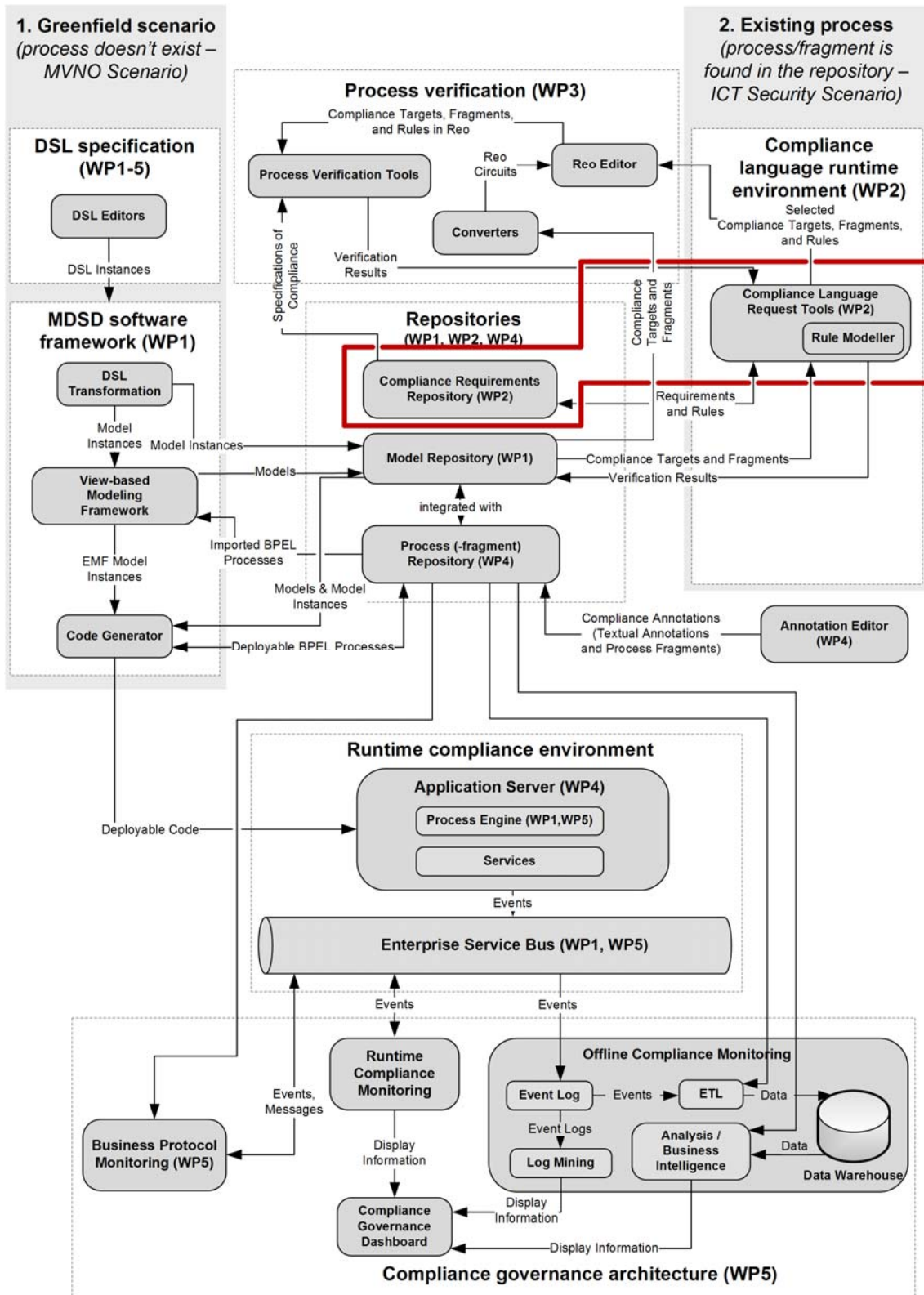


Figure 1 Overall COMPAS architecture

3. Rule Modeller Prototype Description and Design

The internalization and definition of compliance requirements and relevant concepts, and establishing their internal and external relationships within a repository (CRR for our

case) are among the key elements of business process compliance. Besides other uses, CRLT in the COMPAS Architecture is developed mainly to serve this purpose.

Higher-level concepts related to compliance requirements are defined by using the web-based interface of the CRLT (<http://eriss.uvt.nl/compas>). (Please refer to [D2.6] section 3.1 for the main features of the CRLT). Having internalized the compliance constraints and defined concrete requirements, compliance risks, sources and controls, *compliance expert* uses the *Rule Modeller* to visually represent the controls as pattern expressions and generate compliance rules as LTL formulas.

The Rule Modeller is developed as a standalone application in Microsoft Visual Studio³ development environment using C# programming language. It works on Windows environments⁴.

Figure 2 presents the components that comprise the Rule Modeller and its relationship with CRR. First, the visual representations of the expressions are built through the ‘Graphical Expression Editor’ component of the Rule Modeller. These graphical pattern expressions are stored as XML documents.

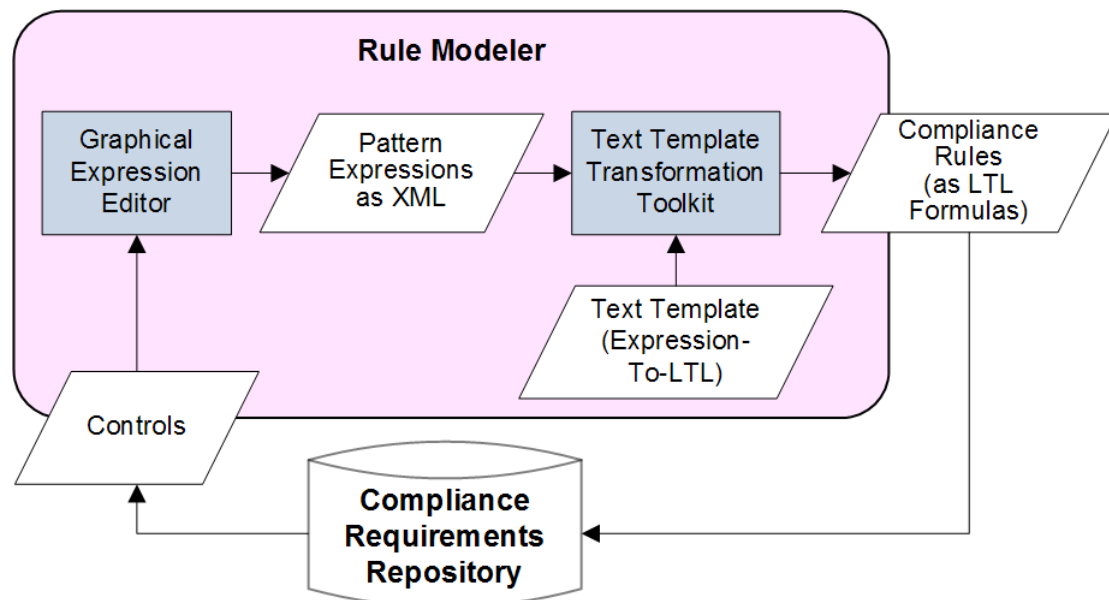


Figure 2 Rule Modeller Components

The automatic transformation of pattern expressions into LTL formulas is implemented via Microsoft Visual Studio 2008 *Text Template Transformation Toolkit* (also known as T4) as a part of Domain-Specific Language Tools. T4 can be used to generate code based on a *text template*, which is a file that contains a mixture of text blocks and control logic. When a text template is transformed, the control logic combines the text blocks with the data in a model to produce an output file. Text templates can be used to create text artefacts. For our purpose, the output of the text template transformation is an XML document that contains Compliance rules as LTL formulas and all other related information.

³Microsoft Visual Studio 2008 with Visual Studio 2008 SDK supporting visual domain-specific languages, <http://msdn.microsoft.com/en-us/vstudio>.

⁴Current version of the software is tested successfully on Windows Vista and Windows 7 operating systems having Microsoft Visual Studio 2005 or above installed.

Generated LTL formulas are stored as compliance rules in the CRR. These rules can also be accessed, updated and searched through CRLT's web interface (refer to [D2.6] section 3.1.6).

The implementation of the Graphical Expression Editor components of the Rule Modeller is completed. The 'text template' that enables the automatic transformation of *basic* and *advanced* pattern expressions to LTL formulas is also implemented. However, the work on the transformation of complex expressions that involve multiple sub-expressions is ongoing. The integration between the Rule Modeller and CRR (mainly transfer of compliance rules to CRR database) also requires further work that will be completed for the final version of the integrated CRLT (due M35).

3.1. Summary of the main features

This section briefly presents the main features supported by the Rule Modeller. Figure 3 displays the main interface of the Graphical Expression Editor component of the Rule Modeller. On the left part of the interface the toolbox is positioned, which presents the primary elements (constructs) that are used to build the pattern-based expressions. These elements include patterns (basic and advanced), basic expression elements (such as activity, role, relationships, etc.), scope elements, and Boolean operators. The details regarding these elements are presented in the next section.

The main working field, where the visual pattern expressions are constructed, is positioned in the centre of the interface (Figure 3). The expressions are built by dragging-and-dropping necessary expression elements from the toolbox onto the drawing canvas. The Graphical Expression Builder controls the type and direction of the relationships that can be established between each expression element and allows only valid relationships based on a predefined meta-model.

In the current version of the Rule modeller, the transformation of the visual representations of the pattern-based expressions into compliance rules (in LTL) based on the transformation rules is initiated by selecting the 'Transform All Templates' command menu in the Solution Explorer toolbox. Figure 4 gives lists examples of generated LTL formulas based on pattern-based expressions given in Figure 3.

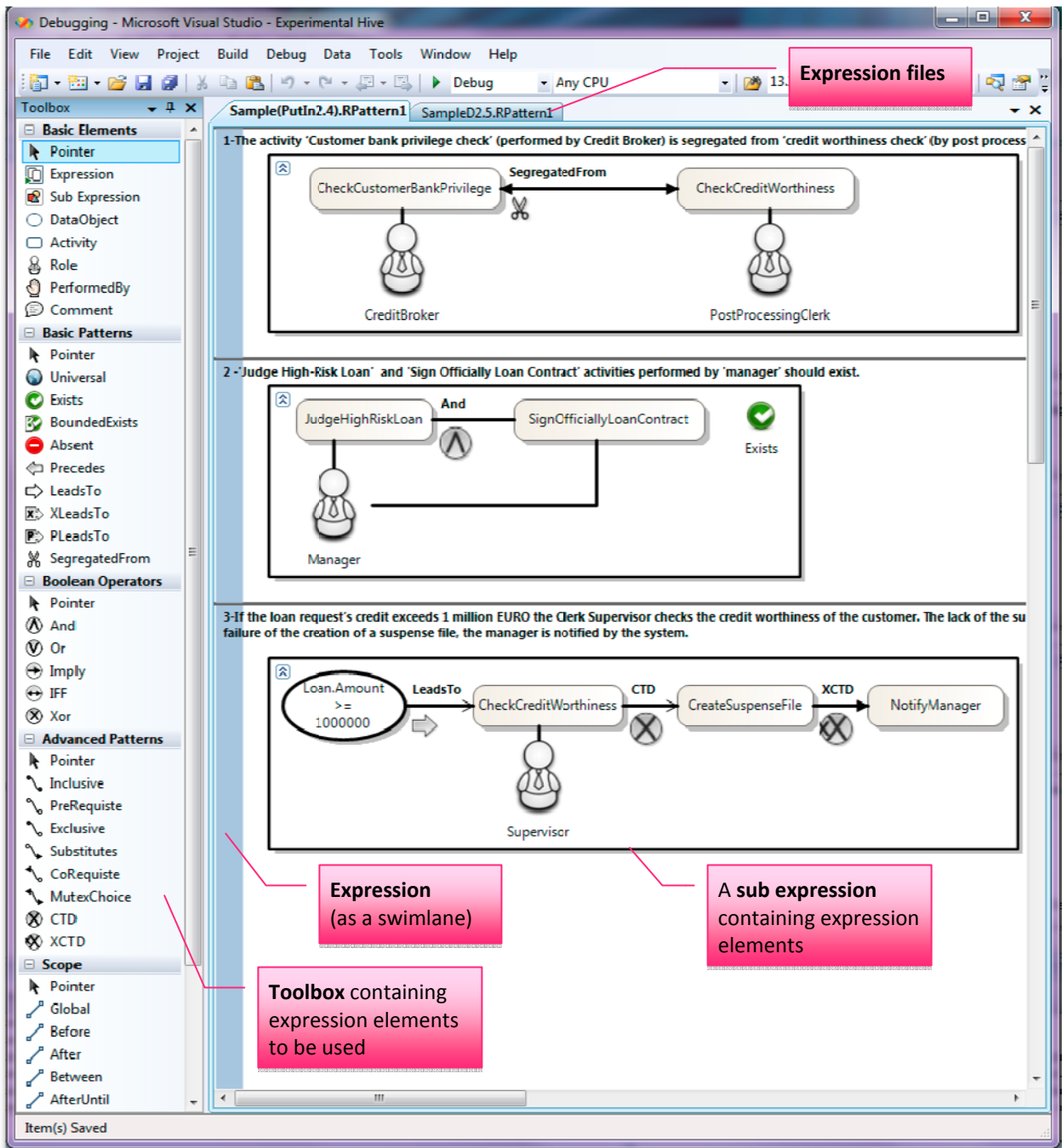
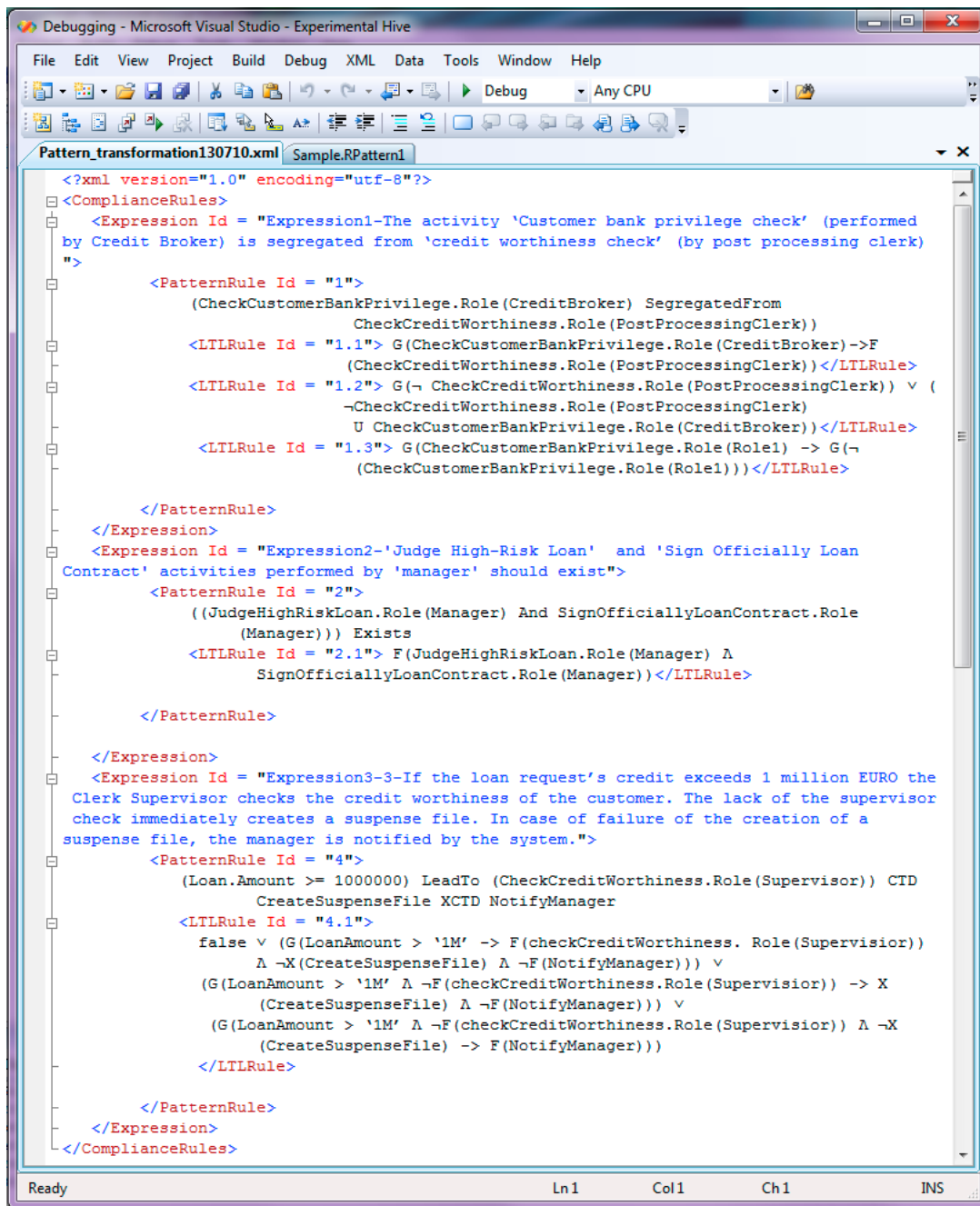


Figure 3 Graphical Expression Editor

The details of the generation rules from pattern expressions to LTL formulas implemented in text templates, as well as the overall approach for the pattern based representation of compliance requirements are presented in [ETH+10c].



```

Debugging - Microsoft Visual Studio - Experimental Hive
File Edit View Project Build Debug XML Data Tools Window Help
Debug Any CPU
Pattern_transformation130710.xml Sample.RPattern1
<?xml version="1.0" encoding="utf-8"?>
<ComplianceRules>
  <Expression Id = "Expression1-The activity 'Customer bank privilege check' (performed
  by Credit Broker) is segregated from 'credit worthiness check' (by post processing clerk)
  ">
    <PatternRule Id = "1">
      (CheckCustomerBankPrivilege.Role (CreditBroker) SegregatedFrom
      CheckCreditWorthiness.Role (PostProcessingClerk))
      <LTTLRule Id = "1.1"> G(CheckCustomerBankPrivilege.Role (CreditBroker)->F
      (CheckCreditWorthiness.Role (PostProcessingClerk))</LTTLRule>
      <LTTLRule Id = "1.2"> G(¬ CheckCreditWorthiness.Role (PostProcessingClerk)) ∨ (
      ¬CheckCreditWorthiness.Role (PostProcessingClerk)
      U CheckCustomerBankPrivilege.Role (CreditBroker))</LTTLRule>
      <LTTLRule Id = "1.3"> G(CheckCustomerBankPrivilege.Role (Role1) -> G(¬
      (CheckCustomerBankPrivilege.Role (Role1)))</LTTLRule>
    </PatternRule>
  </Expression>
  <Expression Id = "Expression2-'Judge High-Risk Loan' and 'Sign Officially Loan
  Contract' activities performed by 'manager' should exist">
    <PatternRule Id = "2">
      ((JudgeHighRiskLoan.Role (Manager) And SignOfficiallyLoanContract.Role
      (Manager))) Exists
      <LTTLRule Id = "2.1"> F(JudgeHighRiskLoan.Role (Manager) ∧
      SignOfficiallyLoanContract.Role (Manager))</LTTLRule>
    </PatternRule>
  </Expression>
  <Expression Id = "Expression3-3-If the loan request's credit exceeds 1 million EURO the
  Clerk Supervisor checks the credit worthiness of the customer. The lack of the supervisor
  check immediately creates a suspense file. In case of failure of the creation of a
  suspense file, the manager is notified by the system.">
    <PatternRule Id = "4">
      (Loan.Amount >= 1000000) LeadTo (CheckCreditWorthiness.Role (Supervisor)) CTD
      CreateSuspenseFile XCTD NotifyManager
      <LTTLRule Id = "4.1">
        false ∨ (G(LoanAmount > '1M' -> F(checkCreditWorthiness.Role (Supervisor))
        ∧ ¬X(CreateSuspenseFile) ∧ ¬F(NotifyManager))) ∨
        (G(LoanAmount > '1M' ∧ ¬F(checkCreditWorthiness.Role (Supervisor)) -> X
        (CreateSuspenseFile) ∧ ¬F(NotifyManager))) ∨
        (G(LoanAmount > '1M' ∧ ¬F(checkCreditWorthiness.Role (Supervisor)) ∧ ¬X
        (CreateSuspenseFile) -> F(NotifyManager)))
      </LTTLRule>
    </PatternRule>
  </Expression>
</ComplianceRules>
Ready Ln1 Col1 Ch1 INS

```

Figure 4 Example LTL formulas generated

3.2. Expression Elements

This section elaborates the elements (constructs) used to build pattern-based expressions, which comprise the primary building blocks of the CRL. The first subsection presents the CRL meta-model regarding the pattern expressions. Subsequent sections describe the elements and their use in detail by going through the elements listed on the 'Toolbox' (Figure 3).

3.2.1. CRL meta-model for pattern expressions

Figure5 presents the meta-model for the particular segment of the CRL. The model is based on original *property specification patterns*, which can be used for the specification of certain compliance requirements.

As shown in Figure5, an *expression* comprises *property specification patterns* (*patterns* in short), *operands* and *scope*. Expressions can combine multiple (sub) expressions by using Boolean operators. For example; the expression “((P *Precedes* Q) *And* (R *Exists*)) *Globally*” comprises two sub-expressions, where *Precedes* and *Exists* indicates patterns; *P*, *Q* and *R* are operands; ‘*And*’ represents the conjunction Boolean operator, and ‘*Globally*’ indicates the scope of the expression.

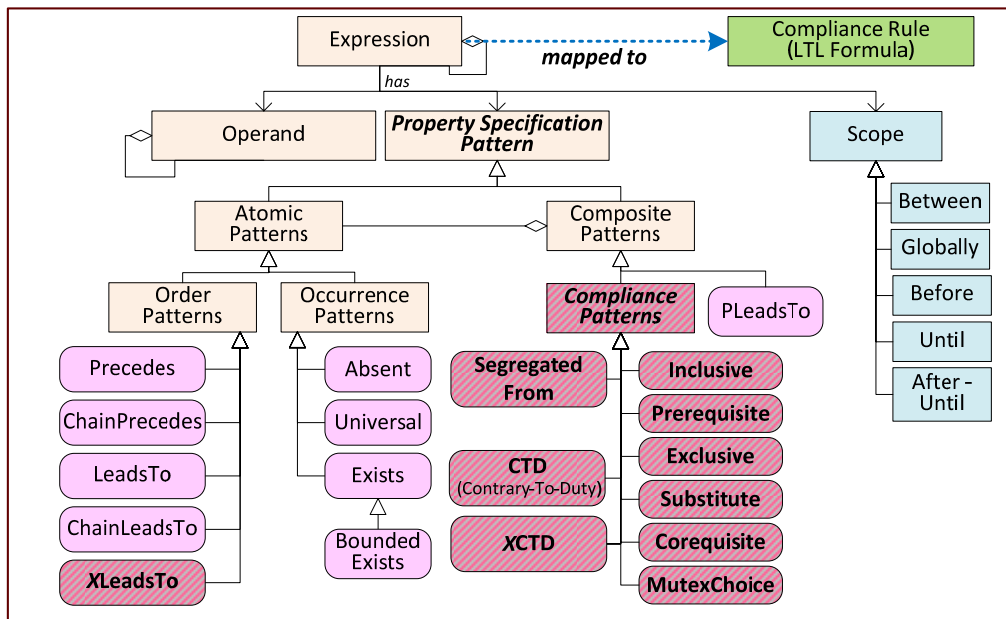


Figure5CRLconceptual model for pattern expressions

Operand take the form of a business process construct (such as activities, roles, etc.), or a particular condition regarding its attributes. For example; the expression “CreateOrderLeadsToApproveOrder” has two operands (activities in this case) connected via the *LeadsTo* pattern. “Loan.Risk = ‘High’” represent a particular condition regarding the ‘*risk*’ attribute of the of the ‘*Loan*’ data object. Similar to expressions, operands can also be combined and nested via Boolean operators.

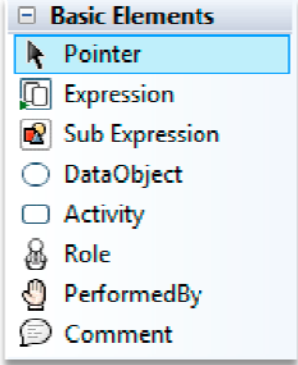
Pattern is a generalized description of a commonly occurring requirement and describes the essential structure of some aspect of a system’s behaviour. Patterns are categorized into two main classes, namely; *atomic* and *composite*. **Atomic patterns** deal with the *occurrence* and *ordering* of operands. **Composite patterns** are built up from combinations of two or more atomic patterns. They utilize Boolean logical operators, including *Not*, *And*, *Or*, *Xor*, *Imply*, and *Iff* to enable the nesting of patterns and allow the definition of complex requirements in terms of other patterns. For example, the *PLeadsTo* pattern, shown in Figure5, is an ‘*And*’ composition of ‘*PPrecedesQ*’ *And* ‘*PLeadsToQ*’, which indicates that *P* and *Q* should occur and must occur sequentially.

Scope defines a starting and an ending *state*⁵ for an expression. For example: *scope global* indicates that the expression must hold during the complete system execution; while *before Q* indicates that the expression must hold up to the occurrence of a given state *Q*. With respect to the compliance domain, it may be desirable that a given property holds throughout the entire system execution. Hence, for simplicity, the *scope* can be omitted in the expression assuming that it is ‘*global*’. Scope types that can be used for pattern expressions are described in Section 3.2.6.

3.2.2. Basic Expression Elements

The toolbox located on the left hand side of the Graphical Expression Editor (Figure 3) is structured in categories of expression elements. This section describes the “basic expression elements” comprising the meta-level constructs (expression, sub-expression & comment), operands (data object, activity & role) and their relationships (performed by). Table 1 lists the description of these elements.

Table 1 Basic Expression Elements⁶

Element	Description	
Expression	Dragging and dropping an expression element creates a swimlane on the drawing canvas where various expression elements are placed and connected, and form a pattern-based expression.	
Sub-Expression	An expression contains one or more sub-expressions, which can be connected via Boolean operators (section 3.2.5) or binary patterns (sections 3.2.3 and 3.2.4) to compose the expression. Sub-expressions act as a container for the expression elements. An element should be placed in a sub-expression, to which it belongs.	
Data object	Data object is an <i>operand</i> used to represent the properties relevant to a specific business/data object in a process. This element is mainly used to represent a certain condition relevant to a specific attribute of a business/data object. E.g.: “Loan.Amount > 1M”, “Order.Status= ‘Approved’”, etc.	
Activity	Activity <i>operand</i> represents a task/activity in a business process. E.g.: Create Order, Approve Credit, etc.	
Role	Activities are assigned to a particular entity (human, software system, etc.) responsible for their execution. Role <i>operand</i> represents these entities. E.g.: Manager, Clerk, Supervisor, etc.	
Performed By	Activities are assigned to roles by using ‘performed by’ <i>relationship</i> type.	
Comment	Comment element is used to link information relevant to the expression or to any of its element.	

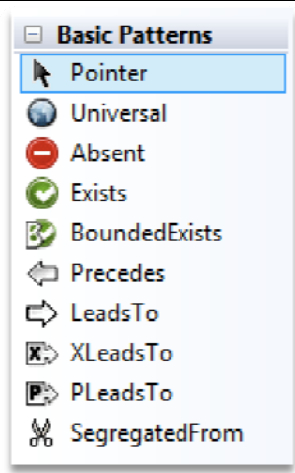
⁵ A *state* represents a node in finite state automata (in our case the formal representation of the business process model). In our context, it indicates a certain business process activity or a condition on any related artefact. *CheckCreditWorthiness* activity and ‘*Loan.Amount > 1M*’ branching condition are examples of states.

⁶ ‘Pointer’ icons on the toolbox do not represent a specific expression element but are used to change the properties of the shape of the expression elements (size, position, etc.) on the drawing canvas.

3.2.3. Basic Patterns

Frequently used patterns are grouped into “basic patterns” category. These patterns involve both *atomic/composite* and *unary/binary*⁷ patterns. Table 2 presents the descriptions and their usage (given P and Q operands).

Table 2 Basic Patterns

Basic Pattern	Type	Usage	Description	
Universal	Atomic, Unary	$P \text{ Universal}$	Indicates that state P occurs throughout the scope	
Exists	Atomic, Unary	$P \text{ Exists}$	Describes a condition that contains an instance of certain states within a specific scope	
BoundedExists	Atomic, Unary	$P \text{ BoundedExists} \leq k$	Indicates that state P must occur at least/exactly/at most k times within the scope	
Absent	Atomic, Unary	$P \text{ Absent}$	Describes a condition that necessitates a portion of a system to be free of certain state	
Precedes	Atomic, Binary	$P \text{ Precedes } Q$	A given state P must always precede a given state Q within the scope	
LeadsTo	Atomic, Binary	$P \text{ LeadsTo } Q$	State P must always be followed by Q within the scope	
XLeadsTo	Atomic, Binary	$P \text{ XLeadsTo } Q$	Represents a strict case of the <i>LeadsTo</i> pattern, which requires state P to be directly followed by state Q	
Segregated-From	Composite, Binary	$P \text{ SegregatedFrom } Q$	P and Q should be assigned to different roles	
PLeadsTo	Composite, Binary	$P \text{ PLeadsTo } Q$	P and Q should occur and must occur sequentially	

3.2.4. Advanced Patterns

Patterns that allow the representation of relatively complex requirements are grouped into “advanced patterns”. Patterns in this group are composite and binary. Table 3 presents the descriptions and their usage.

⁷Unary patterns involve one operand (or a collection of operands as sub-expressions) (e.g.: *ActivityXExists*). Binary operators involve two operands (e.g.: *ActivityXLeadsToActivityY*). Unary patterns are placed in a sub-expression and become valid across all the elements placed in that particular sub-expression as a whole (refer to Section 4 for examples of unary patterns and their usage).

Table 3 Advanced Patterns

Advanced Pattern	Usage	Description	
Inclusive	$P \text{ Inclusive } Q$	The presence of P mandates that Q is also present	
Prerequisite	$P \text{ Prerequisite } Q$	The absence of P mandates that Q is also absent	
Exclusive	$P \text{ Exclusive } Q$	The presence of P mandates the absence of Q and the presence of Q mandates the absence of P	
Substitute	$Q \text{ Substitute } P$	Q substitutes the absence of P	
Corequisite	$P \text{ Corequisite } Q$	Either activities P and Q should exist together or to be absent together	
MutexChoice	$P \text{ MutexChoice } Q$	Either P or Q exists but not both	
CTD (⊗) / XCDT (x⊗)	$P \text{ X/LeadsTo } Q \text{ x/} \otimes Z \text{ x/} \otimes Y \text{ x/} \otimes U \dots$	Refer to the description in the paragraph below (section 3.2.4.a).	

a) Contrary To Duty and X-Contrary To Duty patterns

Contrary to duty (CTD) and x-contrary to duty (XCTD) patterns require further explanations for their purpose and usage. The *CTD* operator (⊗) is used to specify the violations and the obligations arise as a response to violations. It is used to prioritize a set of propositions (obligations) such that; one and only one obligation should be true by respecting their order. Although logically not equivalent, the formation is analogous to ‘if...then...else’ statements.

In the general case, the rule takes the form: $P \otimes \otimes \otimes$. The underlying semantics of the rule is as follows: If P is true then Y is the primary obligation that should take place; if Y cannot be satisfied, then Z should take place (which compensates the violation of Y), and if Z is violated then L should take place and so on. The entire rule is evaluated to false if none of these obligations are satisfied, and is evaluated to true if one and only one obligation (respecting their order) is satisfied. $\otimes \otimes \otimes$ is called a reparational chain for the violation of the primary obligation Y .

We introduce the corresponding *CTD* and *XCTD* compliance patterns to further ease the formalization process and enhance usability. *XCTD* represents a more strict case of the *CTD* pattern, which mandates that a certain obligation should take place exactly in the next future state in the business process model. *CTD* and *XCTD* patterns should be used in conjunction with *LeadsTo* or *XLeadsTo* patterns to capture the semantics of the *CTD* statements. Section 4 gives an example compliance requirement, where *CTD* pattern is used to reflect the semantics in the expression.

b) ChainPrecedes&ChainLeadsTopatterns

ChainPrecedes&ChainLeadsTo patterns (seeFigure5)can be represented by using patterns that have already been implemented.

ChainLeadsTo indicates that a sequence of states P_1, \dots, P_n must always be *followed by* a sequence of states Q_1, \dots, Q_m . This pattern can be represented by using PLeadsTo and LeadsTo patterns. E.g.:

(P_1, P_2, P_3) ChainLeadsTo (Q_1, Q_2, Q_3) :

$(P_1, \text{PLeadsTo } P_2 \text{PLeadsTo } P_3)$ LeadsTo $(Q_1, \text{PLeadsTo } Q_2 \text{PLeadsTo } Q_3)$

Similarly, **ChainPrecedes** indicates that a sequence of states P_1, \dots, P_n must always be *preceded* by a sequence of states Q_1, \dots, Q_m . This pattern can be represented by using PLeadsTo and Precedes patterns. E.g.:

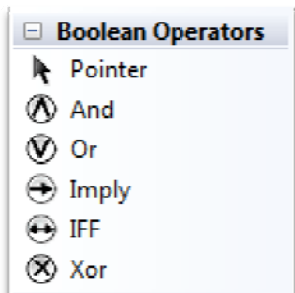
(P_1, P_2, P_3) ChainPrecedes (Q_1, Q_2, Q_3) :

$(P_1, \text{PLeadsTo } P_2 \text{PLeadsTo } P_3)$ Precedes $(Q_1, \text{PLeadsTo } Q_2 \text{PLeadsTo } Q_3)$

3.2.5. Boolean Operators

Boolean operators are used to logically connect operands or sub-expressions. In particular, the following operators are supported: And, Or, Imply, IFF, Xor& Not⁸. Table 4 gives the descriptions for these operators.

Table 4 Boolean Operators

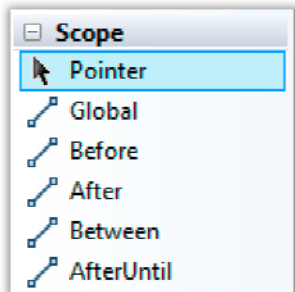
Boolean Operator	Description	
AND	The proposition (p AND q) is true if both p is true and q is true	
OR	The proposition (p OR q) is true if p is true or if q is true (or both)	
IMPLY	The proposition (p IMPLIES q) [also written (IF p THEN q) or ($p \rightarrow q$)] is true if p is true, then q is also true (it is <i>also</i> true if p is false)	
IFF	(If And Only If) The proposition (p IFF q) [also written ($p \leftrightarrow q$)] is true if both p and q are true, or if both p and q are false; otherwise, the proposition is false.	
XOR	(Exclusive Or) The proposition (p XOR q) is true if exactly one of the propositions p and q is true	

⁸ ‘Not’ operator (which represents the logical negation of the proposition) is unary, and implemented as a specific attribute of operands. Operands (activities & data objects) can be negated by updating their ‘negated’ attribute on the ‘properties’ toolbox.

3.2.6. Scope

A scope indicates the certainsection of a system execution (in our case the formal representation of a business process model) through which the expression will be valid. Table 5 describes five basic types of scopes for given Q and R as certain *states* (such as; *ApproveOrder* activity, *Order.Approved*= 'Yes' branching condition, etc.).

Table 5 Scope Elements

	Description	
Global	The expression must hold during the complete system execution	
Before Q	The expression must hold up to the occurrence of a given state Q	
After Q	The expression must hold after the occurrence of Q	
Between Q And R	The expression must hold from the occurrence of a given state Q to the occurrence of a given state R	
After Q Until R	Has the same meaning as 'Between' but the expression holds even if R never occurs	

As discussed in Section 3.2.1, the absence of the scope element in the visual expressions assumes the scope of the expression to be 'global'.

4. Example Pattern Expressions

This section exemplifies the usage of the Rule Modeller by going through a set of compliance requirement examples from the Loan Origination/Approval Case Scenario [D6.2]. Table 6 lists an excerpt of the compliance requirements that are slightly modified (from the original requirements in [D6.2]) to better illustrate the use of specific patterns for the representation of more complex requirements.

Table 6 An excerpt of compliance requirements

ID	Control	Compliance Requirement	Compliance Source
1	The activity ' <i>Customer bank privilege check</i> ' is segregated from ' <i>credit worthiness check</i> '	Duties are adequately segregated	- Sarbanes-Oxley Sec. 404 - ISO 17799-10.1.3
2	If the loan request's <i>credit exceeds 1 million EURO</i> the Clerk Supervisor <i>checks the credit worthiness</i> of the customer. The lack of the supervisor check immediately creates a suspense file. In case of failure of the creation of a suspense file, the manager is notified by the system.		
3	The branch office <i>Manager</i> checks whether risks are acceptable and makes <i>either the final approval or rejection</i> of the request		
4	The customer receives an automated email notification when his personal data is collected by the "Credit Bureau service".	Customer's personal data is handled confidentially	- 95/46/EC (Data protection directive)

Figure6 shows the pattern-based expressions of the examples listed in Table 6. Each swimlane represents an expression (for a control). Expression constructs are placed into one or more sub-expressions within a swimlane. For example; the first control in Table 6 is visualized in the upper part of the interface in the first swimlane. For this example, the expression elements are situated into a sub-expression to represent the control that mandates ‘segregation of duties’ among certain activities that take place in the business process.

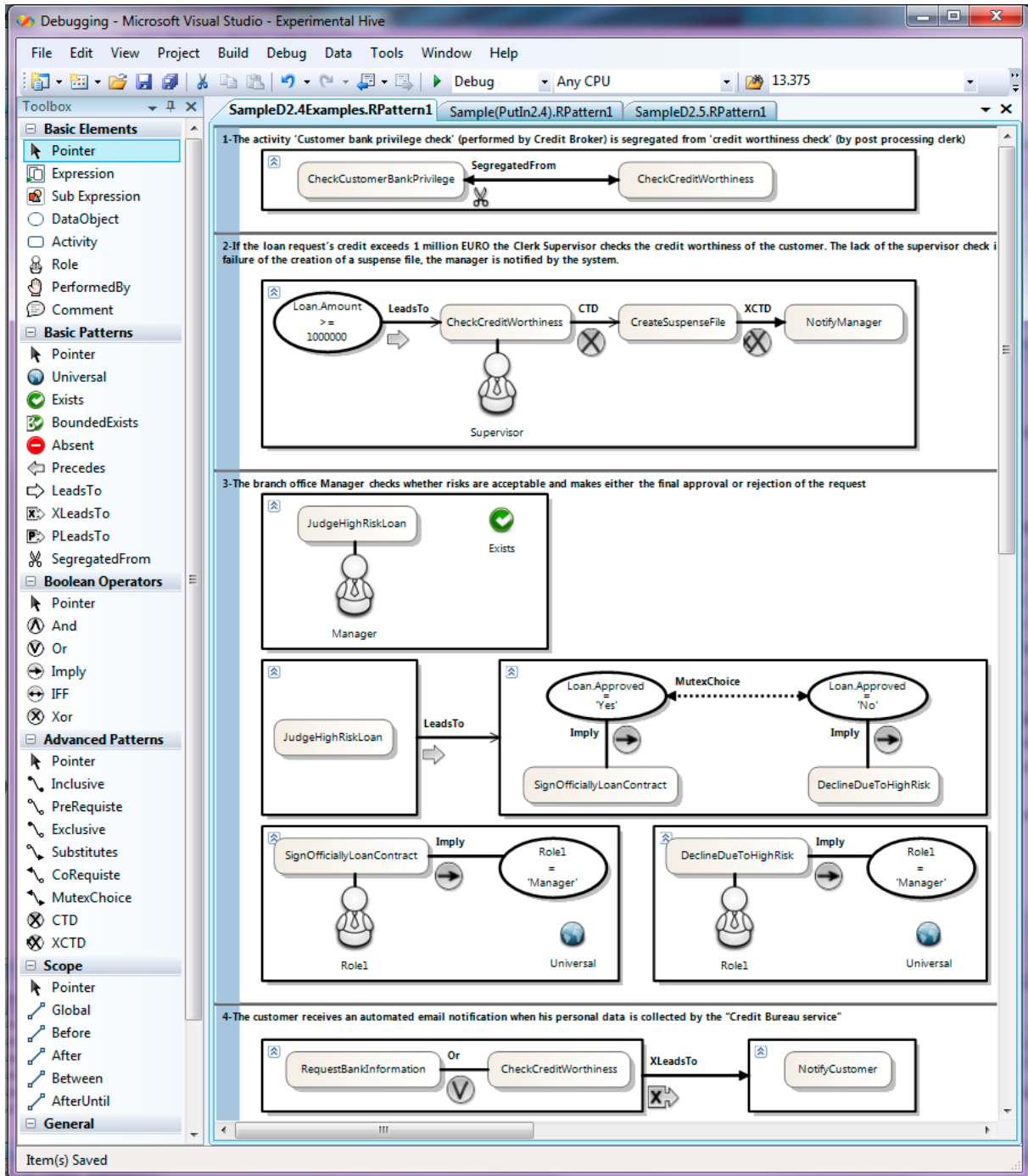


Figure6 Examples of pattern expressions for the scenario

Sub-expressions within a swimlane (i.e., expression) can be connected by using binary patterns or Boolean operators. The sub-expressions that are not explicitly connected with other sub-expressions are considered to be connected with ‘and’ logical operator. The

expressions 3 and 4 (in Figure6) give examples of sub-expressions that are either connected with a binary pattern or left detached.

The pattern expressions depicted in Figure6 are transformed into LTL statements based on transformation rules (as discussed in Section 3.1). Each sub-expression yields one or more compliance rules as LTL formulas. Table 7 lists the LTL formulas and the text-based pattern expressions to be generated from the visual representations of the requirements.

Table 7 An excerpt of compliance requirements

ID	Control	Pattern Expression (Text-based)	Compliance Rules (in LTL)
1	The activity 'Customer bank privilege check' is segregated from 'credit worthiness check'	1.1': (CheckCustomerBankPrivilege SegregatedFrom CheckCreditWorthiness)	1.1.1'': $G(\neg \text{CheckCreditWorthiness } W \text{ CheckCustomerBankPrivilege})$ 1.1.2'': $G(\text{CheckCustomerBankPrivilege} \rightarrow F(\text{CheckCreditWorthiness}))$ 1.1.3'': $G((\text{CheckCustomerBankPrivilege.Role}(\text{Role1}) \rightarrow G(\neg(\text{CheckCreditWorthiness.Role}(\text{Role1}))))$
2	If the loan request's credit exceeds 1 million EURO the Clerk Supervisor checks the credit worthiness of the customer. The lack of the supervisor check immediately creates a suspense file. In case of failure of the creation of a suspense file, the manager is notified by the system.	2.1': Loan.Amount \geq '1M' LeadsTo CheckCreditWorthiness.Role(Supervisor) XCTD CreateSuspenseFile CTD NotifyManager	2.1.1'': $false \square (G(\text{LoanAmount} > '1M' \rightarrow F(\text{checkCreditWorthiness.Role}(\text{Supervisor}))) \wedge \neg X(\text{CreateSuspenseFile}) \wedge \neg F(\text{NotifyManager})) \square (G(\text{LoanAmount} > '1M' \wedge \neg F(\text{checkCreditWorthiness.Role}(\text{Supervisor}))) \rightarrow X(\text{CreateSuspenseFile}) \wedge \neg F(\text{NotifyManager})) \square (G(\text{LoanAmount} > '1M' \wedge \neg F(\text{checkCreditWorthiness.Role}(\text{Supervisor}))) \wedge \neg X(\text{CreateSuspenseFile}) \rightarrow F(\text{NotifyManager}))$
3	The branch office Manager checks whether risks are acceptable and makes either the final approval or rejection of the request	3.1': (JudgeHighRiskLoan.Role('Manager')) Exists 3.2': (JudgeHighRiskLoan LeadsTo ((Loan.Approved = 'Yes' Imply SignOfficiallyLoanContract) MutexChoice (Loan.Approved = 'No' Imply DeclineDueToHighRisk))) 3.3': ((SignOfficiallyLoanContract.Role(Role1) Imply Role1 = 'Manager') And (DeclineDueToHighRisk.Role(Role1) Imply Role1 = 'Manager')) Universal	3.1.1'': $F(\text{JudgeHighRiskLoan.Role}('Manager'))$ 3.2.1'': $G(\text{JudgeHighRiskLoan} \rightarrow (F((\text{SignOfficiallyLoanContract} \wedge G(\neg \text{DeclineDueToHighRisk}))) \square F((\text{DeclineDueToHighRisk} \wedge G(\neg \text{SignOfficiallyLoanContract}))))$ 3.3.1'': $G((\text{SignOfficiallyLoanContract.Role}(\text{Role1}) \rightarrow \text{Role1} = 'Manager') \wedge (\text{DeclineDueToHighRisk.Role}(\text{Role1}) \rightarrow \text{Role1} = 'Manager'))$
4	The customer receives an automated email notification when his personal data is collected by the "Credit Bureau service".	4.1' - ((RequestBankInformation \square CheckCreditWorthiness) XLeadsTo NotifyCustomer)	4.1.1'': $G((\text{RequestBankInformation} \square \text{CheckCreditWorthiness}) \rightarrow X(\text{NotifyCustomer}))$

5. How to Install and Use?

This Section presents a brief description of how to install and use the RuleModeler prototype. At a minimum, you must have Visual Studio 2005 Standard Edition installed on your computer. To install RuleModeler, click on Rule_modeler.msi windows installation file located in the main folder where you extracted the prototype. The wizard window shown in Figure 7 pops up. Click on 'Next' button, and then follow the wizard.

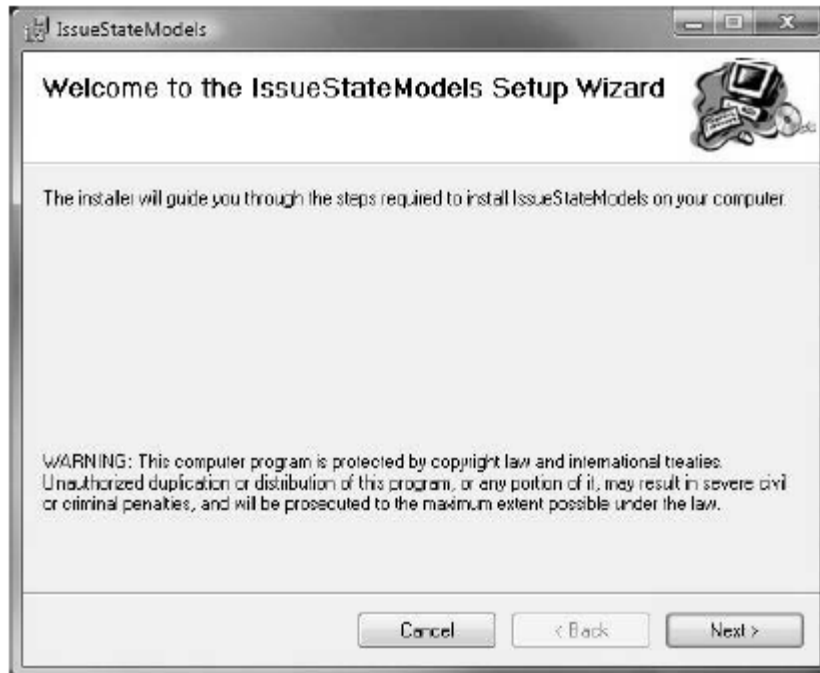


Figure 7 RuleModeler installation wizard

After the installation completes successfully, you can use RuleModeler as a Domain specific language template, such that it acts as the base of your new project. To do this, open Microsoft Visual Studio, then click on 'File' menu -> New -> Project. The 'new project' window shown in Figure 8 appears.

Click on 'Other Project Types' -> 'Extensibility' -> 'Domain-Specific Language Designer'. Then in the *Name* and *Location* fields, choose a name and location where your new project is stored. Then click on 'Ok' button. The Domain-Specific Language Designer Wizard appears. Click on 'RuleModeler' in "Which template would you like to base your domain-specific language on?". Then choose a name for your domain-specific language. Click on 'Next' button, and then follow the wizard.

When your new project opens, click on 'transform all templates' in the Solution Explorer window, then press on F5 to debug your new project. The Microsoft Visual Studio-Experimental Hive opens, where you can start building new compliance expressions and automatically generate corresponding LTL formulas for verification as explained before in the previous Sections.

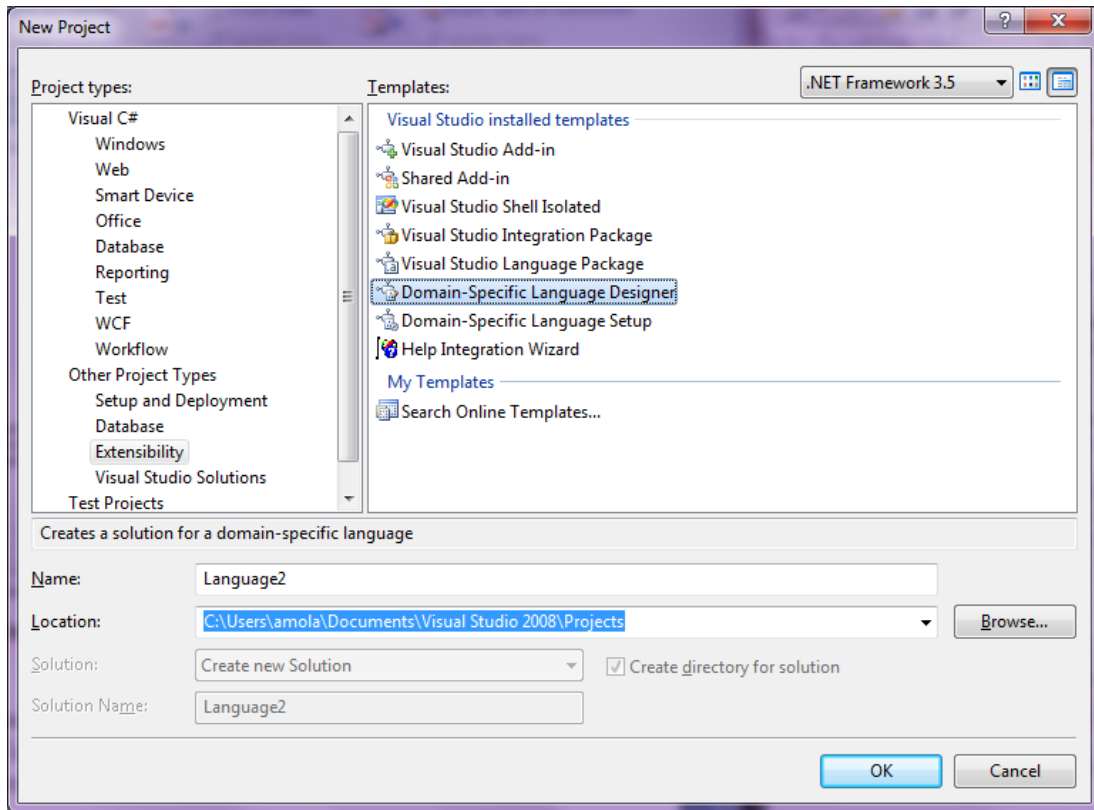


Figure 8 New Project Window

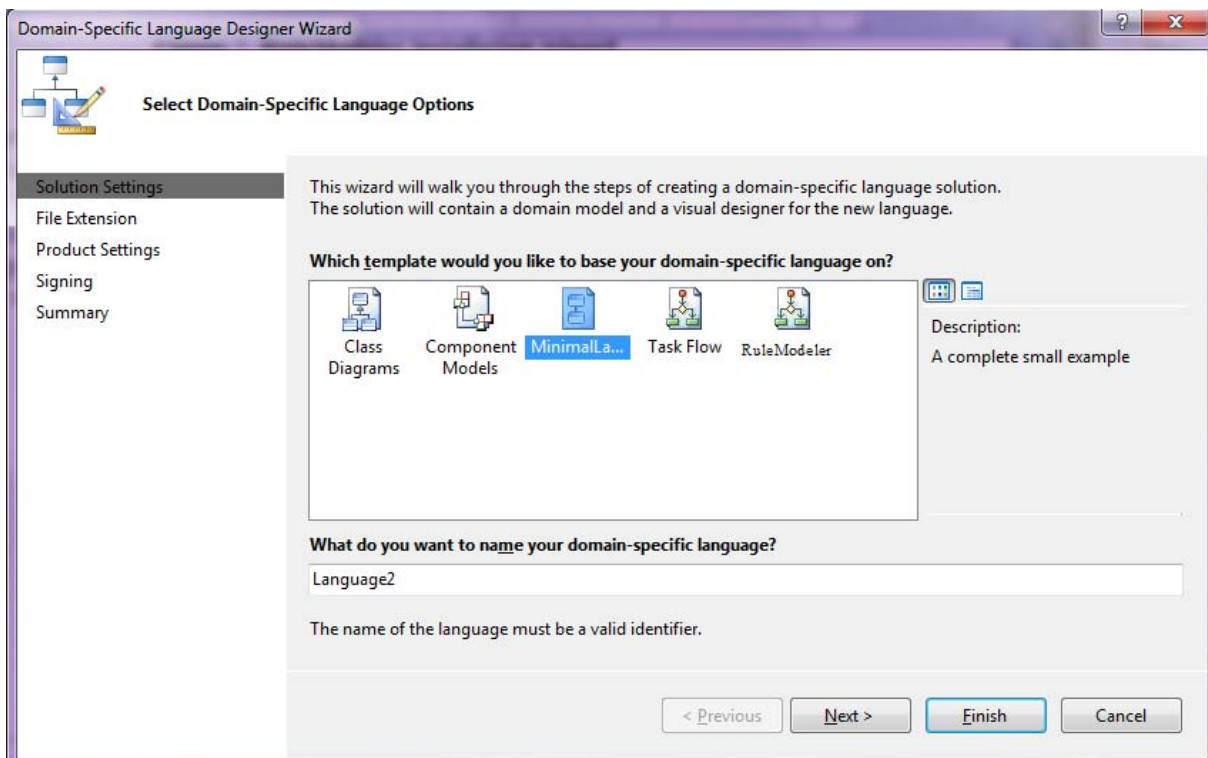


Figure 9 Domain-Specific Language Designer Wizard

6. Reference documents

6.1. Internal documents

- [DoW] Description of Work for COMPAS, 2008-02-01.
- [DA.1] COMPAS Architectural Walkthroughs and Evaluation Metrics
- [D2.1] State-of-the-art in the field of compliance languages
- [D2.2] Initial Specification of Compliance Language Constructs and Operators
- [D2.3] Design of Compliance Language Run-time Environment and Architecture
- [D2.6] Implementation of an integrated prototype handling interactive user specified compliance requests in a compliance language
- [D4.1] Report on the existing approaches to improving reusability of processes and service compositions
- [D4.4] Supporting infrastructure – process engine, process artefact repository, process generation tool
- [D5.4] Reasoning mechanisms to support the identification and the analysis of problems associated with user requests.
- [D6.2] Application implementation and case study prototypes
- [D7.1] “Public Web-Site”, <http://www.compas-ict.eu>

6.2. External documents

- [AFF+02] R.Armoni, L. Fix, A.Flaisher, R.Gerth, B. Ginsburg, T.Kanza, A.Landver, S.Mador-Haim, E.Singerman, A.Tiemeyer, M.Vardi, Y.Zbar: The ForSpec Temporal Logic: A New Temporal Property-Specification Language. Lecture Notes In Computer Science, Vol. 2280 (2002).
- [DAC98] M. Dwyer, G. Avrunin, J. Corbett: Property Specification Patterns for Finite-State Verification. Int. Workshop on Formal Methods on Software Practice, 1998, pp. 7-15.
- [ETH+10a] A. Elgammal, O. Turetken, W-J. van den Heuvel, M. Papazoglou: On the Formal Specification of Business Contracts and Regulatory Compliance. 4th Workshop on FLACOS.EPTCS, Pisa, Italy, 2010 (to appear).
- [ETH+10b] A. Elgammal, O. Turetken, W. van den Heuvel, M. Papazoglou: Root-Cause Analysis of Design-time Compliance Violations on the basis of Property Patterns. 8th International Conference on Service-Oriented Computing (ICSOC10), USA, 2010 (to appear).
- [ETH+10c] A. Elgammal, O. Turetken, W-J. van den Heuvel, M. Papazoglou: Towards a Comprehensive Pattern-based Business Process Compliance Language. International Conference on Cooperative Information Systems (CoopIS'10). Create, Greece, 2010 (submitted).

- [PNU77] A. Pnueli: The Temporal Logic of Programs: 18th IEEE Symposium on Foundations of Computer Science, pp. 46–57 (1977).
- [STK+10] D. Schumm, O. Turetken, N. Kokash, A. Elgammal, F. Leymann, W-J. van den Heuvel: Business Process Compliance through Reusable Units of Compliant Processes. In: Workshop on Engineering SOA and the Web (ESW'10). LNCS, Austria, 2010 (to appear).