Universidade Federal Fluminense

Joel André Ferreira dos Santos

Multimedia and hypermedia document validation and verification using a model-driven approach

NITERÓI

2012

UNIVERSIDADE FEDERAL FLUMINENSE

Joel André Ferreira dos Santos

Multimedia and hypermedia document validation and verification using a model-driven approach

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Orientador:
Profa. Débora Christina Muchaluat Saade, D.Sc.

Co-orientador:
Prof. Christiano de Oliveira Braga, D.Sc.

NITERÓI

2012

Multimedia and hypermedia document validation and verification using a model-driven approach

Joel André Ferreira dos Santos

Dissertação de Mestrado submetida ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do título de Mestre. Área de concentração: Redes e Sistemas Distribuídos e Paralelos.

Aprovada por:

Niterói, 26 de Março de 2012.

A dúvida é o princípio da sabedoria.
(Aristóteles)

aos meus pais Joel e Fátima, por terem tornado essa caminhada possível;

à minha irmã Maria Rosa, por ser um exemplo;

à minha namorada Alina, por seu apoio constante.

Agradecimentos

Aos meus pais e irmã: Joel, Fátima e Maria Rosa, que estiveram ao meu lado durante a realização deste trabalho. Em alguns momentos tentando até entender o que eu estava fazendo. À minha namorada, Alina, pelo apoio constante.

Às amizades construídas na Universidade Federal Fluminense, em especial no laboratório MídiaCom, pelo excelente convívio e pelas trocas de conhecimento.

À Professora Débora Christina Muchaluat Saade, mais uma vez, pela sua enorme paciência e excelente orientação. Ao Professor Christiano de Oliveira Braga, por suas contribuições e orientação.

A todos aqueles que contribuíram para a conclusão deste trabalho. Em especial à Júlia Varanda, por sua ajuda no desenvolvimento e manutenção da API aNa e ao Roberto Menezes, por sua grande ajuda no desenvolvimento dos invariantes OCL e no uso do TCLib. Agradeço também às Professora Loana Tito e Maria Luiza e aos amigos Diego Passos e Rafael Carvalho por seus excelentes conselhos.

Finalmente, a você, por estar lendo esta dissertação.

Resumo

Este trabalho discute a validação e verificação de documentos hipermídia e multimídia. Com a validação e verificação de documentos hipermídia e multimídia, é possível indicar para o autor possíveis pontos inconsistentes em sua definição. Dessa forma o autor tem a possibilidade de corrigir essas inconsistências, garantindo a boa definição da aplicação antes desta ser disponibilizada ao usuário final.

A validação e verificação apresentada neste trabalho é dividida em um conjunto de propriedades que um documento deve apresentar. Estas propriedades foram propostas baseadas em trabalhos relacionados publicados na literatura. A implementação da validação e verificação usa uma abordagem dirigida a modelos.

Este trabalho também apresenta uma ferramenta, chamada aNaa (API for NCL Authoring and Analysis), capaz de realizar a validação e verificação apresentada, baseada no conjunto de propriedades propostas, garantindo a consistência de documentos especificados com a linguagem NCL (Nested Context Language).

Palavras-chave: Validação de Documentos Multimídia, Verificação de Documentos Multimídia, Aplicações Multimídia Interativas, Sincronização baseada em Eventos, Autoria Multimídia, NCM, NCL, aNa, aNaa.

Abstract

This work discusses the validation and verification of hypermedia and multimedia documents. With the validation and verification of hypermedia and multimedia documents, it is possible to indicate to the author possible inconsistent points in his definition. It also provides the author the possibility of correcting those inconsistencies, guaranteeing that the application is well-defined before it is made available for the final user.

The validation and verification here presented are divided into a set of desirable document properties, which are proposed based on related works published in the literature. The implementation of the validation and verification uses a model-driven approach.

This work also presents a tool, named aNaa (API for NCL Authoring and Analysis), capable of performing the validation and verification here presented in order to guarantee the consistency of NCL (Nested Context Language) documents, based on the set of properties proposed.

Keywords: Multimedia Document Validation, Multimedia Document Verification, Interactive Multimedia Applications, Event-based synchronization, Multimedia Authoring, NCM, NCL, aNa, aNaa.

Acronyms

aNa	:	API for NCL Authoring
aNaa	:	API for NCL Authoring and Analysis
API	:	Application Programming Interface
AST	:	Abstract Syntax Tree
caT	:	context aware Trellis
DOM	:	Document Object Model
DTV	:	Digital Television
FDT	:	Formal Description Technique
HMBS	:	Hypermedia Model Based on Statecharts
HTML	:	HyperText Markup Language
IPTV	:	IP Television
ITU	:	International Telecommunications Union
ITU-T	:	ITU Telecommunication Standardization Sector
JMF	:	Java Media Framework API
LOTOS	:	Language Of Temporal Ordering Specification
LSM	:	Language Structure Metamodel
LTL	:	Linear Temporal Logic
MDA	:	Model-driven Architecture
NCL	:	Nested Context Language
NCM	:	Nested Context Model
OCL	:	Object Constraint Language
OMG	:	Object Management Group
RT-LOTOS	:	Real-Time LOTOS
SHM	:	Simple Hypermedia Model
SMIL	:	Synchronized Multimedia Integration Language
TDD	:	Test-driven Development
UML	:	Unified Modeling Language
W3C	:	World Wide Web Consortium
XHTML	:	eXtensible HyperText Markup Language

XML	:	eXtensible Markup Language
XSLT	:	XML Stylesheet Language Transformations

List of Figures

List of Listings

List of Tables

Contents

Chapter 1

Introduction

Efforts like the ITU-T H series [ITU 1997], focus on the convergence of Digital TV (DTV) systems, such as: IPTV, Terrestrial DTV, etc. Such a convergence will result in a common middleware layer for different DTV systems. A common middleware for DTV systems will increase interest in multimedia systems[1]. In order to support multimedia document authoring and presentation in different platforms, validation and verification of multimedia documents is very important, guaranteeing that applications are well-defined before deployed. Thus, an approach for analyzing multimedia documents is necessary, so authors can identify possible specification problems and correct them.

The use of a declarative language simplifies the definition of multimedia documents since it emphasizes the description of a problem rather than its decomposition into the necessary steps to solve it.

Although declarative languages make the creation of interactive multimedia applications easier, by allowing the document author to focus on authoring aspects, when an application becomes more complex, for example, with many media objects and user interaction, the multimedia document that describes it gets bigger, with many XML [W3C 2008a] code lines.

Commonly, the authoring of large multimedia documents is more error-prone, since the author tends to reuse previously defined specifications and structures copying code. Besides, sometimes the author forgets to define some crucial relationships for the desired behavior of the application being created. This may cause non-termination and/or unreachability of parts of a given document. Another very common error is the definition of conflicting relationships, leading an application to present an undesirable behavior.

[1]In the literature, the terms *multimedia* and *hypermedia* are sometimes used with the same meaning. Sometimes, the term *hypermedia* is used to represent interactive *multimedia* documents. In this work, both terms are used as synonyms.

One approach to avoid this problem is to enrich the specification of a document with temporal and/or spatial semantics in some language elements, facilitating the specification of relationships among application components. One example of this approach is seen in SMIL (Synchronized Multimedia Integration Language) [W3C 2008b], which provides a set of containers with embedded temporal semantics (*<seq>, <par>* and *<excl>*).

Another approach is the use of authoring facilities that abstract the authoring language expressiveness, defining an authoring layer where the language elements present spatial and/or temporal embedded semantics. One example of this approach is the use of hypermedia composite templates [Santos and Muchaluat-Saade 2011] to embed spatio-temporal semantics into NCL (Nested Context Language) [ITU 2009] contexts.

These approaches provide an important contribution for the creation of interactive multimedia applications. However, they are not capable of preventing the creation of documents with specification problems, either because the authoring language can not be enriched at a point that avoids it or because a composite template may be defined incorrectly by an author.

1.1 Motivation

The multimedia application authoring process, in this work, is defined as the necessary steps for the creation of a multimedia document before its distribution to the final user. Typically, this process is divided in two steps, the authoring of the multimedia document and its execution in order to verify its behavior. Figure 1.1 presents this process.

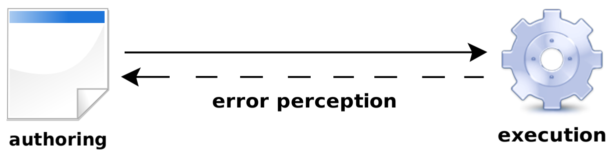


Figure 1.1: Typical authoring process

In the first step, the author defines the multimedia document that describes the desired temporal scenario. A temporal scenario is the description of a set of activities that are in some way related in time [Pérez-Luque and Little 1996]. It will be presented in more details in Chapter 3. In order to verify if the created document corresponds to the desired temporal scenario, the author executes the application in the authoring language presentation engine. In case the application behavior is not the desired one, the author goes back to the document definition searching for code parts potentially responsible for that behavior. This iteration continues until the specification problems are solved and the application works properly. This process is not efficient since, as stated by [Eidenberger 2003], it is not always clear if an undesired behavior is due to a conflict generated by an author's wrong definition or a bug in the presentation engine implementation. Additionally, it is necessary several simulations to cover all possible situations of a document execution. Bugs in a presentation engine implementation would not occur if a well-defined set of benchmark multimedia documents for a specific multimedia language were available.

In order to allow an efficient authoring process, some works in the literature propose models, languages and tools that intervene themselves in the authoring-execution path, providing the analysis of the multimedia document being created, alerting the author of possible specification problems. So, the execution of the application is not necessary in order to verify its behavior. This approach is illustrated in Figure 1.2.

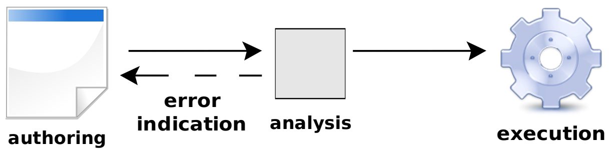


Figure 1.2: Authoring process with analysis

This work evolves from the need to create consistent multimedia documents. We consider a document consistent, when it does not present specification problems or an undesired behavior. With a common middleware for DTV systems and the simple approach for the creation of multimedia documents provided by declarative languages, users with different skills can create interactive applications. It is important that authors are able to analyze the created multimedia document in order to guarantee that it is in accordance with the desired behavior. The success of those applications for final users depends on their consistency. Multimedia document specification problems may lead to application bugs, reducing the application distribution and use by final users.

This work proposes an approach to check the consistency of declarative multimedia documents. The consistency checking, proposed in this work, is achieved through model validation and verification. *Validation* means to demonstrate that a property is valid for a particular case, whether *verification* means to demonstrate that a property is valid for a general case. In order to make this text simpler, the remaining of this dissertation will use *analysis* to represent the *validation* and/or *verification* of multimedia documents.

1.2 Objectives

The main objective of this work is to propose a solution for analyzing multimedia documents specified with the NCM model, using the NCL language, in order to avoid authoring errors and guarantee that they work as desired. This analysis will be done after the document authoring is complete, to test the multimedia document and present its possible specification problems to the author, allowing him to correct them.

Several works present approaches for the analysis of multimedia document consistency. From the analysis of those works, it is possible to identify desirable properties that, when satisfied in a multimedia document, it can be considered consistent. Those properties should be generic enough to make their use possible for the analysis of documents described with different multimedia languages. One objective of this work is the proposition of that set of properties. Another objective is to present a generic method for the analysis of multimedia documents following the set of generic properties here defined.

The multimedia document analysis presented in this work is divided in two parts: the validation of the document structural definition, which investigates if the multimedia document created by the author satisfies the syntactic rules defined by the declarative authoring language grammar, and the verification of the document behavioral definition, which investigates if the multimedia document created describes a temporal scenario possibly free of errors.

The analysis follows an MDA (Model-driven Architecture) [OMG 2003] approach. The validation of the document structural definition is achieved by representing the document as an instance of a metamodel that represents the authoring language structure and the representation of the authoring language syntactic rules as invariants. Metamodels and metamodel instances representation uses UML (Unified Modeling Language) [OMG 2010] and invariants are represented by OCL (Object Constraint Language) invariants [Warmer and Kleppe 1999]. The verification of the document behavioral definition is achieved by representing the document in a general model that represents multimedia documents behavior, and investigating multimedia documents temporal properties over its induced transition system. The multimedia documents temporal properties representation uses LTL (Linear Temporal Logic) [Pnueli 1977] and the automatic investigation of those properties is done by a model checker [Clarke et al. 2000]. The properties, as well as the representation of the multimedia document to be verified and the investigation of the temporal properties are implemented using Maude [Clavel et al. 2007].

As a test case, this work develops an analysis tool for documents defined by the NCM model (Nested Context Model) [Soares et al. 2000], using the NCL language. NCL was chosen since it is a standard for DTV systems. The developed tool is called aNaa - API for NCL Authoring and Analysis.

1.3 Contributions

This work contributions are:

- The definition of a set of properties that, when satisfied in a multimedia document, it can be considered consistent;
- The use of a model-driven approach for the analysis of multimedia documents, which brings the following contributions:
  - The definition of a general method for the validation of multimedia document structural definitions;
  - The definition of a model that represents a multimedia document spatio-temporal specification for NCL documents;
  - The definition of a general method for the verification of multimedia document behavioral definition.
- The development of an API called aNa (API for NCL Authoring) for representing NCL documents, which brings the following contributions:
  - The creation of a data model specifically for representing NCL documents;
  - The implementation of a common core for NCL authoring tools, making possible to exchange object-oriented data among different tools without the need to generate XML code;
- The development of an API called aNaa (API for NCL Authoring and Analysis), making possible for authoring tools to analyze an NCL document.

Those contributions will be presented during this text and highlighted in this dissertation conclusion.

1.4 Dissertation structure

The remainder of this dissertation is structured as follows.

Chapter 2 discusses related work revisiting analysis techniques published in the literature. The related works presented are divided regarding their use or not of formal description approaches.

Chapter 3 presents a multimedia background. It describes some multimedia concepts to clarify the text comprehension and presents the different classes of temporal synchronization models. To illustrate the models presented, this chapter also presents an example of temporal scenario and its description in the different temporal synchronization models. This chapter also gives an overview of the NCM model and the NCL language. Since NCM and NCL are used for the test case analysis tool implementation, those concepts facilitate understanding this work.

Chapter 4 presents an MDA background. It describes some MDA concepts and how the OCL validation and the model checking work. This chapter also introduces linear temporal logic, rewriting logic and its use for model checking.

Chapter 5 defines the proposed validation and verification properties to be satisfied in a multimedia document. The properties are divided into static properties and dynamic properties. The static properties are used for the multimedia document definition structural validation, while the dynamic properties are used for the document behavioral verification. It also presents a comparison among related work regarding the validation and verification properties.

Chapter 6 presents the proposed method for the validation of the structure of multimedia documents. It describes the modeling of the multimedia language structure and how the static properties are represented for that model. It also shows the representation of a multimedia document following the language structure metamodel. This chapter content is described over the NCL language.

Chapter 7 presents the proposed method for the verification of the behavior of multimedia documents. It proposes a generic model for multimedia documents and presents the formalization of the verification properties for the document verification.

Chapter 8 presents the implementation of the analysis here presented by aNaa. This chapter describes the tools used for achieving NCL document analysis and the APIs created.

Chapter 9 shows some NCL document examples and their analysis using aNaa. It also presents some performance tests and a discussion of the tool limitations.

Chapter 10 concludes this dissertation, restating this work contributions and presents future and ongoing works.

Chapter 2

Related work

Many previous works in the literature proposed models, languages and tools to improve the multimedia authoring process by providing analysis of multimedia documents and alerting the author about possible problems in its specification. However, great part of those works is not recent. In recent years there is a lack of available tools for multimedia document analysis published in the literature.

Those works, taking into account their functionality, are divided in: approaches for structural validation [Araújo et al. 2008, Honorato and Barbosa 2010] and approaches for behavioral verification [Santos et al. 1998, Na and Furuta 2001, Furuta and Stotts 2001, Oliveira et al. 2001, Felix 2004, Bossi and Gaggi 2007, Bertino et al. 2005, Elias et al. 2006, Ma and Shin 2004]. The following sections present those related works.

2.1 Approaches for structural validation

In [Araújo et al. 2008], the authors presented a validation process for hypermedia documents specified with the NCL language. The validation process is divided in four steps: (1) lexical and syntactic validation, (2) structural validation, (3) contextual and reference validation and (4) semantic validation.

The lexical and syntactic validation investigates the lexical and syntactic structure of the XML document. The structural validation investigates if all element attributes are valid, all the required attributes are present and the element children are correct and have the correct cardinality. The contextual and reference validation checks if some element referenced in an attribute exists and if it has the type the attribute demands. It also checks if the elements are in the same context. The semantic validation investigates if document parts are not reached or if there are alternatives that will never be chosen.

The NCL-validator tool is an implementation of that validation process. However, the semantic validation step is not implemented, since, according to the paper, it does not endanger the NCL document validation. NCL-validator is used as a library for NCL document validation by the NCL-Eclipse [Azevedo et al. 2009] and NCL Composer [Lima et al. 2010] authoring tools. When authoring problems are found, the tool returns error messages to the author identifying the correspondent problem.

In [Honorato and Barbosa 2010], the authors presented the NCL-Inspector tool. This tool, based on other tools for code quality critique, supports the authoring of NCL applications. It supports the author in terms of code quality. With this tool, besides the possibility of analyzing the NCL code searching for coding problems, it is possible to suggest modifications regarding best programming practices.

The code analysis, or inspection, is done following a set of rules, forming a rule repository. Each rule presents an NCL code pattern and an action realized when that pattern is found. Also, each rule is implemented as a plugin, so the system as a whole may be extended by adding new rules. Another benefit of that approach is the possibility of sharing those rules with other NCL-Inspector users.

The specification of a rule may be done using XSLT (XML Stylesheet Language Transformations) [W3C 1999] and Java languages. During a rule creation, the author may test it through a mini-test framework available. This framework follows a Test-driven Development (TDD) approach.

For the inspection of an NCL document, NCL-Inspector parses the document. After that step, the tool creates an AST (Abstract Syntax Tree) that represents the NCL document being inspected. Then it walks through the AST searching for violations of the existent rules. The violations found are presented to the user so he can correct the application code. Those violations are presented as error or warning messages. Besides the AST, an inspector (part of the tool that inspects a specific rule) may act directly with the application code with some textual abstraction. This way, it is possible to investigate specific text details, for example, the use of the tabulation character (\t) for code indenting.

2.2 Approaches for behavioral verification

In [Santos et al. 1998], the authors presented a way to detect possible undesired behaviors created by the combination of conflicting temporal constraints in a multimedia document.

The approach presented in the paper did not assume, a priori, a specific model to express and compose the temporal constraints, but used generic authoring models. The multimedia document to be analyzed is translated, automatically, into a formal specification, in that case, into generic FDTs (Formal Description Techniques). With the specification of an FDT document, it is possible to apply general validation techniques for the analysis of the multimedia document consistency. In that paper, RT-LOTOS was chosen as the formalism to be used.

In order to translate the multimedia document into RT-LOTOS processes, general mapping rules were used. Also, the definition of RT-LOTOS process libraries, specifying the behavior of reusable document parts, were used. The modularity and hierarchy of RT-LOTOS allows the combination of processes specifying the document presentation with other processes modeling the available platform.

The verification consists in the interpretation of the minimum reachability graph, in order to prove if the action corresponding to the presentation end can be reached from the initial state. Each node in the graph represents a reachable state and each edge, the occurrence of an action or temporal progression. When a possible undesired behavior is found, the tool returns an error message to the author, so he can repair it.

The paper presented different possible undesired behavior situations to be analyzed, which are: qualitative, if they do not depend on an object duration, and quantitative, if they depend on an object duration. The possible undesired behaviors can also be intrinsic if they do not depend on the platform where the document is presented and extrinsic if they depend on it. In the last case, it is considered if platform resources are blocking or non-blocking. Blocking resources are the ones that can not be used by two objects at the same time. An audio channel, according to the paper, is an example of blocking resource. In addition, presentation component delays were also considered. Regarding those delays, the document behavior may become undesired.

Although in [Santos et al. 1998] the analyzed hypermedia documents were specified according to NCM [Soares et al. 2000], their proposal may also be used to analyze documents specified with SMIL [W3C 2008b].

In [Na and Furuta 2001], the authors presented the caT (context aware Trellis) system, an evolution of Trellis [Furuta and Stotts 2001] that provides application adaptation according to the user context. An author, using an authoring tool based on Petri nets, builds the document structure and associates network places to document media objects.

The work presented Petri nets as a good candidate for modeling multimedia documents, since its synchronization is easy to model and allows the analysis of important properties. Basic Petri nets, however, are not convenient for representing and analyzing complex systems, since their tokens do not have identity. To overcome this limitation, the paper proposes Petri nets with identifiable tokens, called High-Level Petri nets.

The caT system provides the separation among document specification and presentation, allowing multiple presentations for a document specification. To reduce the authoring graphical complexity and improve net reuse, caT incorporates hierarchical Petri nets. The authoring tool supports a tool for the analysis of hierarchical Petri nets, through its reachability graph.

The analysis tool builds the reachability tree of the analyzed document. The author defines limit values for the occurrence of dead links (transitions that may not be triggered), places with token excess, besides other options, as the analysis maximum time. Then the tool investigates the existence of a terminal state, that is, if there is a state where no transitions are triggered. It also investigates the limitation property, that is, if no place in the net has an unlimited number of tokens and the safeness property, that is, if each place in the net has a token. The limitation analysis is important since tokens may represent scarce system resources.

In [Oliveira et al. 2001], the authors presented HMBS (Hypermedia Model Based on Statecharts). That model uses statecharts for the authoring of multimedia documents. Statecharts are extensions of finite state machines and HMBS is a generalization of hypertext models based on hypergraphs. The paper presents the use of models for the authoring of multimedia documents as a way to encourage a structured development, since the document structure is defined before content is added to the model.

An HMBS hypermedia application is described by a statechart that represents its structural hierarchy, regarding nodes and links, and its *human-consumable* components. Those components are expressed as information units, called pages and anchors. The statechart execution semantics provide the application navigation model. A statechart state is mapped into pages and transactions and events represent a set of possible link activations.

During the execution process, the statechart assumes a new state configuration correspondent to the set of current basic states. The substates of an *OR* decomposition (state with sequential temporal semantics) are not activated simultaneously, while the substates of an *AND* decomposition (state with parallel temporal semantics) are simultaneously activated, as long as its parent state remains active.

The statechart reachability tree for a specific configuration may be used to verify if any page can not be reached. For this, the occurrence of a state s in one of the generated configurations is verified. If s does not occur, the information associated to that state will not be visible when the application navigation starts in the initial state considered. In a similar manner, it is possible to determine if a certain group of pages may be seen simultaneously searching state configurations containing the states associated to those pages. The reachability tree also allows the detection of configurations from which no other page may be reached or that present cyclical paths.

The reachability tree also allows determining the maximum number of simultaneous windows necessary to present the application. Analyzing the tree and determining the maximum number of active states, it is possible to determine a better layout for the application.

An obvious overload of using caT and HMBS systems is that the author interacting with those systems should be familiarized with the models and their basic formalism. Although that need can be discarded if the formalism is "hidden" in a more friendly metaphor, those systems diminish the author freedom, not allowing the choice of another multimedia model to be used for the application creation.

In [Felix 2004], the author presents an approach for the verification of temporal properties of multimedia documents through the application of model checking techniques. Since the work did not present a name, here it will be identified by NCL-FA (formal analysis). The work presents a notation used for the description of NCL relevant characteristics, in the case its temporal characteristics.

Such a description is transformed into a timed automata net that indicates the document temporal behavior. The transformation creates a state machine for each media node and a synchronizer machine for each link. A synchronizer machine is used to tie together the occurrence of transitions in the media node state machines.

The work also presents a tool where the author can define temporal-logic formulas for verification of the temporal properties. The temporal verification is done with a model-checker. It is worth mentioning that the work does not define any temporal-logic formula, forcing the author to know the formalism used.

In [Bossi and Gaggi 2007], the authors proposed an authoring system that includes a semantic analysis module for multimedia document temporal behavior evaluation. This is obtained by defining a formal semantics for the SMIL language [W3C 2008b]. Since that approach did not present a name, here the system will be identified by SMIL-EA (enriched with assertions). The proposed semantics is defined through a set of inference rules inspired by Hoare logic. The main characteristic of Hoare logic is that it describes how a command, or code part, changes the computation state. That way, the SMIL structure may be enriched with assertions expressing temporal properties that may be used during the authoring phase. Another application resulting from the defined formal semantics is the concept of equivalence, which guarantees that two sets of SMIL tags may be replaced, without changing the application behavior.

The work presented the choice of inserting temporal assertions in a SMIL document as a way of diminishing the analysis complexity, since this approach does not require the translation of the document being created to some formalism and then perform its analysis. The analysis is done during the authoring phase, whenever the author wants or when he saves the application. This is done to diminish the occurrence of error messages regarding possible specification problems generated during the application creation.

The assertions defined by the semantics proposed in the paper specify the system temporal state before and after the execution of a SMIL tag or set of tags. For the system correctness verification, the tool applies axioms, also defined by the proposed semantics, in order to verify if a tag, or set of tags, correctly changes the system temporal state. Otherwise, the tool presents the author the problem found so it can be corrected.

In [Bertino et al. 2005], the authors proposed an authoring model based on constraints. Since the paper did not indicate a name for the model, it is identified as BFPS (authors' initials). In the BFPS model, a multimedia application consists of several sub presentations, each one representing a topic composed of multimedia objects semantically related. All relations, temporal, layout and structural, are specified in a single step. This way, the author defines a set of high-level constraints that will be used by the system to automatically group the objects into topics. The application generation process is responsible for three main tasks: consistency checking, presentation structure generation and topics generation.

The presentation consistency is checked by applying compatibility rules to each pair of constraints, detecting inconsistencies. Before checking, several inference rules are applied to the initial specification to determine constraints that, even not defined explicitly, are consequences of the constraints defined. If an inconsistency arises, the system applies relaxation techniques, reducing the constraint set until the presentation becomes consistent or, when it is not possible, the author must review the specification.

The presentation structure generation process creates a structure that reflects the given application specification. The structure is represented by a direct graph where each vertex represents a topic and the edges, the connections among them. This process always returns a consistent graph, otherwise, the author should review the specification. After this step, the system relates media objects to topics. According to the constraints, it creates connections among topics and checks the consistency before returning the final generated graph. If any failure occurs, the author is warned about the inconsistencies found.

In [Elias et al. 2006], the authors present an algorithm for dynamic checking spatio-temporal relations. Since the paper does not present a name for the approach, it will be identified by EEC (authors' initials). Dynamic, in the paper, means that the checking is done during presentation specification. The paper extends the work published in [Ma and Shin 2004] proposing new operators to model the temporal and spatial relations of a multimedia document, solving limitations of the operators defined in [Ma and Shin 2004].

Temporal inconsistencies occur when a set of constrains can not be satisfied at the same time. Incompleteness of a constraint set occurs when there is a discontinuity in the presentation, that is, there is a media object set that is not reached during presentation. In case an inconsistency occurs in a constraint set, one of the constraints must be removed in order to obtain a consistent set. That removal is done by relating a priority value to each constraint. In case two inconsistent constraints present the same priority, relaxation techniques are applied to determine the constraint to be removed.

The paper presents two operators *TEMPORAL* and *SPATIAL*, to model temporal and spatial relations, respectively. In *TEMPORAL(A, B,* d_1*,* d_2*, priority)*, d_1 and d_2 are the difference among the begin and end time of media objects A and B, respectively. This operator is used to model any temporal relation. *SPATIAL(A, B,* d_{x1}*,* d_{x2}*,* d_{y1}*,* d_{y2}*, priority)* is used to model any spatial relation. In this operator, d_{x1} and d_{x2} are the distances between the x coordinates of the inferior left and superior right corners, respectively, and d_{y1} and d_{y2} are the distance of the y coordinates of those corners.

The consistency checking is done by finding the minimum spanning tree T for the graph defined by the media objects (vertices) and the relationships among them (edges). In order to maintain the presentation consistency and the acyclic nature of T, a relationship that creates cycles must be removed. The choice is done by taking into account the priority of each relationship. For the completeness checking, all vertices must be found in the set that contains the first media object. If this search returns the vertex set of T, then all presentation media objects are reached directly or indirectly from the initial object. Otherwise, the algorithm presents an error message so the author can define restrictions that make the constraint set complete. With the use of the *SPATIAL* operator, it is possible to determine if A overlaps B and vice versa. The spatial consistency is checked the same way as the temporal one.

2.3 Closing remarks

This chapter presented previous works that provide the analysis of multimedia documents. Those works were divided in: formal approaches and informal approaches. The use of a formal approach brings the benefit of having available tools to support the implementation of the analysis, besides guaranteeing the correctness of the analysis done, since it uses formal descriptions of the multimedia language used for the document authoring. Based on the benefits brought by a formal description, this work has chosen to follow that approach.

The works here presented were used for the definition of the validation and verification properties as it will be presented in Chapter 5. The next chapter presents a background on multimedia models.

Chapter 3

Multimedia background

This chapter introduces some multimedia concepts to facilitate the comprehension of this work. Section 3.1 presents basic multimedia concepts. Section 3.2 presents different classes of temporal synchronization models and examples of their representation of a same temporal scenario. Section 3.3 introduces the NCM model and the NCL language, which are used in the proposed solution implementation and Section 3.4 concludes this chapter with some remarks.

3.1 Multimedia basic concepts

This section presents basic concepts related to multimedia documents: temporal scenario, multimedia documents, multimedia document models and temporal synchronization models. Those concepts are based in the definitions presented in [Pérez-Luque and Little 1996, Boll 2001].

A *temporal scenario* represents a set of activities that are in some way related in time. Those activities are associated to the grouping of continuous and discrete media objects into a logically coherent unit and, possibly, taking into account user interaction.

A *multimedia document* is a document that describes a *temporal scenario*. It is composed of nodes, that represent media objects in the *temporal scenario* and relationships among them. Sometimes, multimedia documents can express relationships taking into account node content subunits, called anchors.

A *multimedia document* is an instance of a *multimedia document model*, which defines the entities used by the *multimedia document* for the description of a *temporal scenario*. A *multimedia document model* also defines how temporal dependencies (relationships) among nodes are represented. That representation can be based in one or more *temporal* *synchronization models*.

A *multimedia document model* is said to have more *expressive power* [Pérez-Luque and Little 1996] than another *multimedia document model*, when it is capable of representing more complex *temporal scenarios*.

3.2 Temporal synchronization models

This section presents different classes of *temporal synchronization models*. Different works published in the literature have proposed different ways to classify *multimedia document models*. [Bulterman and Hardman 2005], for example, proposed a classification of the *multimedia document models* based on the authoring approach. Here, the classification is based on [Blakowski and Steinmetz 1996, Boll 2001], dividing the *multimedia document models* regarding how they specify temporal relations among media objects. The following classes of temporal synchronization models were identified: constraint-based, axes-based, hierarchical-based, formal model-based, event-based and script-based synchronization models.

To illustrate the different temporal synchronization models, Figure 3.1 presents a sample of temporal scenario where a video, an audio, an image and a text are related. This example comes from [Boll 2001].

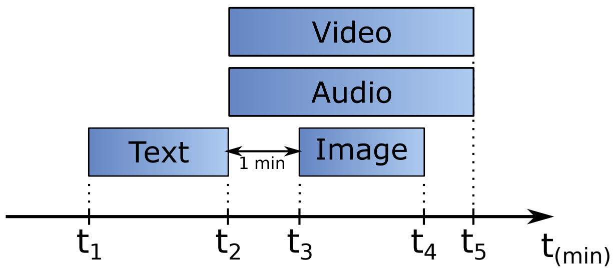


Figure 3.1: Temporal scenario example

The figure describes the presentation of a text, followed by a temporal gap of 60 seconds and then the presentation of an image. In parallel, beginning when the text finishes its presentation, the parallel presentation of an audio and a video starts. The following sections present the classes of temporal synchronization models and present, when possible, how they specify this temporal scenario.

3.2.1 Constraint-based synchronization

Constraint-based synchronization defines constraints about the temporal ordering of media objects. It can be divided in two types: interval-based and reference point-based synchronization.

Interval-based synchronization defines the duration of a node as an interval. The synchronization between two intervals is defined using 13 basic relations presented by [Allen 1983], where some of those relations are invertible. Another approach is to use the 29 relations defined by [Wahl and Rothermel 1994]. One example of an interval-based multimedia authoring system is Madeus [Jourdan et al. 1998].

Although it provides simple definition of relations between nodes (intervals), an interval-based model does not allow the specification of relations regarding node anchors. Those definitions have to be made indirectly. Listing 3.1 presents how Interval-based synchronization is used to specify the sample temporal scenario.

Listing 3.1: Interval-based synchronization

```
video equals audio
text meets video
text meets image delay 1min
```

Reference point-based synchronization uses reference points to define the synchronization of nodes. Reference points may be the beginning and end of a node, or its anchors, presentation. The relation among nodes come from the specification of constraints among reference points, creating synchronization points. That way, anchors participating at the same synchronization point are started or stopped together when that point is reached during the document presentation.

Since it provides the possibility of creating reference points to represent the node internal structure, reference point-based models present more flexibility than interval-based ones. One example of a reference point-based multimedia authoring system is Firefly [Buchanan and Zellweger 2005]. Listing 3.2 presents how Reference point-based synchronization is used to specify the sample temporal scenario.

Listing 3.2: Reference point-based synchronization

```
Point t2:
   text ends and video starts and audio starts and image starts delay 1 min
Point t5:
   video ends and audio ends
```

3.2.2 Axes-based synchronization

Axes-based synchronization, or timeline-based synchronization, maps synchronization events, like the start and end of a node presentation, to a temporal axis. That axis may be global, so every node is attached to the same temporal axis, or it may be virtual. In this case, it is possible to specify different axes to where the nodes may be attached. Examples of systems based on that class are commercial softwares like Apple iMovie [Apple Inc. 2010]. Listing 3.3 presents how Axes-based synchronization is used to specify the sample temporal scenario.

Listing 3.3: Axes-based synchronization

```
text [t1, t2]
video [t2, t5]
audio [t2, t5]
image [t3, t4]
```

3.2.3 Hierarchical-based synchronization

Hierarchical-based synchronization describes the synchronization among nodes using two main synchronization operations: serial or sequential synchronization and parallel synchronization. In that approach, the document synchronization is defined by a tree of temporal containers denoting the serial or parallel presentation of its inner nodes. It is also possible to define a delay for a specific node or even define synchronization constraints among them. SMIL [W3C 2008b] and MPEG-4 XMT [ISO/IEC 2005] are examples of declarative authoring languages based on hierarchical synchronization.

Hierarchical-based models provide a simple definition for node synchronization. Like interval-based synchronization, they do not allow specifying relationships among node anchors. This type of relation is defined indirectly with the use of delays or by dividing a node into different nodes. In addition, the hierarchical structure can not represent some synchronization conditions, like conditions involving more than one type of node event occurrence or state. One example can be the synchronization among two nodes A and B and the occurrence of a user interaction. In this example, the synchronization relationship states that node A will be presented if the user interacts with the application while node B is being presented.

The possibility of manipulating variable state inside multimedia documents is a way of extending hierarchical-based synchronization in order to overcome this limitation, as

SMIL State does [Jansen and Bulterman 2009]. Listing 3.4 presents how Hierarchical-based synchronization is used to specify the sample temporal scenario.

Listing 3.4: Hierarchical-based synchronization

```
seq{
   text
   par{
      video
      audio
      image begin 1min
   }
}
```

3.2.4 Formal model-based synchronization

Formal model-based synchronization uses some kind of formalism to represent nodes and specify synchronization among them. This kind of synchronization takes advantage of the great number of formal model tools available to present the multimedia document described. Among others, two types of formal model-based synchronization can be used: Timed Petri net-based and Statechart-based synchronization.

Timed Petri net-based synchronization uses Petri nets [Peterson 1981] whose places have duration. In a Petri net, a transition will be fired when all its input places contain tokens. Those tokens, when the transition is triggered, are moved to the output places. When a token arrives at a place, it remains blocked for the place duration.

Although it allows the definition of any kind of synchronization, it presents a complex specification for authors and an insufficient abstraction for a node. It does not allow the synchronization among node anchors without the need of dividing the node in several others, as it occurs with hierarchical-based synchronization. Trellis, caT [Furuta and Stotts 2001, Na and Furuta 2001] and HTSPN [Willrich et al. 2001] are examples of that synchronization class.

Statechart-based synchronization uses statecharts to represent nodes and their temporal synchronization. In this approach, states represent media objects and transitions define a state hierarchy. A transition also defines if child states of a common state are presented simultaneously or not. One example of statechart-based synchronization system is seen at [Oliveira et al. 2001].

3.2.5 Event-based synchronization

Event-based synchronization allows the creation of relationships among events that happen during document execution, such as presentation events (start, stop or pause of a node, for example), selection events (user interaction, for example) or attribution events (changing variable values, for example). Examples of event-based synchronization models are the NCM model [Soares et al. 2000] and Labyrinth [Díaz et al. 2001]. Listing 3.5 presents how Event-based synchronization is used to specify the sample temporal scenario.

Listing 3.5: Event-based synchronization

```
text.begin
on text.end do video.begin and audio.begin
on text.end do image.begin delay 1min
```

As stated in [Blakowski and Steinmetz 1996], this type of specification is easily extended to new synchronization types. One extension is the use of events with durations and the possibility of triggering presentation actions, not only by event state transitions, but also by the state of document nodes and values of document variables.

3.2.6 Script-based synchronization

Script-based synchronization uses a textual description for the synchronization of nodes. A script (textual description) defines a set of algorithmic steps that represent the temporal scenario desired. Flash [Adobe Systems 2010] is an example of well-known script use.

Although scripts have a great expressive power, a disadvantage of their use is the fact that the author usually needs to specify all the details for node synchronization, which are already abstracted in a declarative approach. Listing 3.6 presents how Script-based synchronization is used to specify the sample temporal scenario.

Listing 3.6: Script-based synchronization

```
function imageEnd() {
   image.stop();
}

function imageStart() {
   image.start();
   setInterval(imageEnd, (t4−t3)*60000);
}

function videoEnd() {
   video.stop();
   audio.stop();
}

function textEnd() {
   text.stop();
   video.start();
   audio.start();
   setInterval(imageStart, 60000);
   setInterval(videoEnd, (t5-t2)*60000);
}

function textStart() {
   text.start();
   setInterval(textEnd, (t2-t1)*60000);
}

textStart();
```

3.3 The NCM Model and the NCL language

Nested Context Language (NCL) [ABNT 2011, ITU 2009] is based on the Nested Context Model (NCM) [Soares et al. 2000, Soares and Rodrigues 2005], defining XML elements to represent the entities specified by the NCM model. Section 3.3.1 presents the NCM entities and main concepts and Section 3.3.2 presents how NCL represents multimedia documents. Those sections describe only NCM and NCL characteristics that are relevant to this work. References [ABNT 2011, ITU 2009], [Soares et al. 2000] and [Soares and Rodrigues 2005] should be consulted for more details.

3.3.1 NCM

NCM offers two types of nodes: *content nodes* and *composite nodes*. A *content node*, also called media node, represents a media object, for example an audio, a video, a text, etc. A *composite node* represents a set of nodes, which can be content nodes or composite nodes, and a set of links that represent the relationships among those component nodes.

Although a content node represents a media object, it does not have that media object content. It is only a representation of that object inside the document, so relationships can be defined among that media object and other objects inside the document. It indicates the location of the media content, its type and how it will be presented to the final user. In addition, a content node can define interface points for a media object, called *anchors*. An anchor may be a *content anchor* or an *attribute anchor*.

A *content anchor* represents a subset of the media content, possibly the whole media content. Suppose, for example, a content node that represents a video. A content anchor of that video may represents a subpart of that video - a time interval or an amount of sequential frames - or even the whole video. In NCM, a content anchor that represents the whole media content is called *all content anchor*.

An *attribute anchor* represents a node attribute and its value. Supposing the same video node example, an attribute anchor of that video could be its location on the screen, its sound level, etc.

A *composite node*, also called composition or context, represents a set of nodes and links among its component nodes. A composite node also has interface points, which can be *ports* or *attribute anchors*.

A *port* maps a composite node interface point to one inner node interface point. For example, a port may map to a component composite node, a component composite node interface point, a content node or a content node interface point. Since NCM only allows links to be defined among nodes inside the same composition, a port is useful when some node, inside a composition, should be "reached" by a link from outside the composition.

A special type of *composite node* called *switch node* is used to define a set of component nodes that can be alternatively presented. A switch defines a set of nodes and mappings relating those nodes to generic conditions defined by *rules*. The chosen node will be the first one whose rule is evaluated as true. A switch may also have port mappings to component node interface points. The switch port "target" will also be the first one whose node related rule is evaluated as true.

NCM also allows nodes to be reused inside any composite node, however, a composite node can not be recursively contained in itself. In order to identify nodes inside an NCM document, it introduces the notion of *perspective* [Soares et al. 2000]. A node perspective is the whole node path from a node to the most external composite node. The *perspective* of a node N is a sequence $P = (N_m, \ldots, N_1)$, with $m \geq 1$, such that, for $i \in [1, m)$:

- $N_1 = N$;
- N_{i+1} is a composite node;
- N_i is a component node of N_{i+1};
- N_m is not contained in any other node.

In order to define how content nodes will be presented on the screen, NCM offers *regions* and *descriptors*. *Regions* define areas on the screen where visible media objects will be presented. *Descriptors* describe how a media object will be presented. A descriptor may define, for example, the volume of an audio or video object, the transparency level of a figure or even the duration of a media. The descriptor also defines the region where a media object, using this descriptor, will be presented.

NCM is an event-based multimedia model. Every anchor, and therefore content node, in NCM has related events. An event is an occurrence in time with a duration. It has a type and a state. NCM considers seven types of events [Soares and Rodrigues 2005], however, just the *selection*, *presentation* and *attribution* events are relevant to this work. A *selection* event represents a user selection, a *presentation* event represents a content anchor presentation and an *attribution* event represents an attribute anchor value change. During document presentation, each anchor related event has a state.

The state of an event, or event state, is the current state of an event in its associated state machine. The possible event states are: *sleeping*, *occurring* and *paused*. Figure 3.2 presents the NCM event state machine. It also presents the name of the transitions responsible for changing event states.

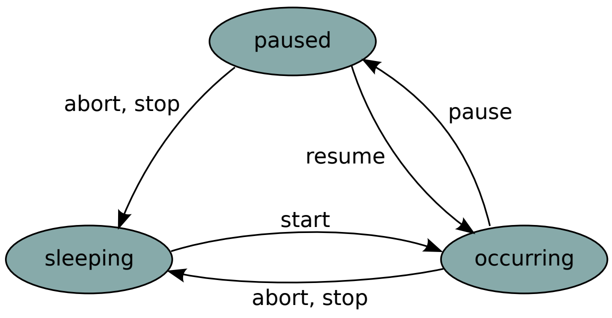


Figure 3.2: NCM event state machine

It is worth mentioning that a content anchor will only be related to presentation and selection events, while an attribute anchor will be related to attribution events.

NCM defines generic relations, represented by *connectors* [Muchaluat-Saade and Soares 2002], which are used in the definition of links (relationships). Connectors may represent a causal relation or a constraint relation, however, just causal relations are relevant to this work. A causal relation has a condition and an action, where the condition, when satisfied, triggers the action. A condition can be defined using event transitions (see Figure 3.2), event states or attribute anchor values. An action triggers the occurrence of a transition in an event state machine. Connector conditions and actions are identified by connector *roles*.

One example of connector is the *onBeginStart* connector. This connector defines two roles: *onBegin* and *start*. This connector specifies a relation stating that "the beginning of the presentation of a node, related to the *onBegin* role, causes the beginning of the presentation of another node, related to the *start* role". The *onBegin* condition will be satisfied when a start transition occurs in its related anchor presentation state, while the *start* action will perform the start transition in its related anchor presentation state. Notice that this connector does not specify which anchors will be attached to its roles, it just defines a relation type.

NCM relationships are defined by *links*. A link uses a relation type defined by a connector and indicates the nodes participating in the relationship. The participant nodes are defined by a set of binds, where each bind attaches a node interface point to a connector role. That way a link ties together the occurrence of transitions in the event states of different node anchors.

In order to exemplify the concepts here presented, Figure 3.3 presents a structural view of a sample NCM document. In the figure, solid circles represent content nodes, dashed circles represent composite nodes, squares represent interface points (ports or anchors), dashed lines represent mappings and arrows represent links.

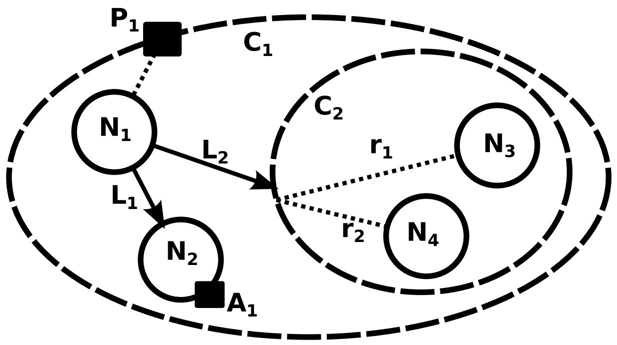


Figure 3.3: Sample NCM document

The document represented in Figure 3.3 has six nodes: content nodes N_1, N_2, N_3 and N_4 and composite nodes C_1 and C_2. Composite node C_1 contains nodes N_1, N_2 and C_2. It also has port P_1 (interface point), which maps to node N_1 and links L_1 and L_2 defining relationships among its nodes. Switch node C_2 contains nodes N_3 and N_4. It will present node N_3, if rule r_1 is evaluated as true, or node N_4, if rule r_2 is evaluated as true. Content node N_2 has one attribute anchor A_1. Regarding node perspectives, node N_3 has the perspective "C_1, C_2, N_3", while, for example, node N_1 has the perspective "C_1, N_1".

Link L_1 defines that anchor A_1 will have its value set to “yes” when node N_1 starts its presentation. Link L_2 defines that switch C_2 will start its presentation when node N_1 ends its presentation. Rules r_1 and r_2 will test if a document global variable named A_1 is equal to “yes” or “no”, respectively. It is worth noticing that anchor A_1 represents that global variable in the NCM document.

When a content node starts its presentation, its *all content anchor* and all its anchors, except temporal anchors, have their presentation event started. A *temporal anchor* is a content anchor defined as a temporal interval. Temporal anchors are started as their start time is reached during the node presentation. An ending or pausing on the node presentation is also performed in every anchor of that node being presented. If, however, the action is done over a node anchor, it is performed only on that anchor.

Regarding composite nodes, every presentation action over a context node is done over its context ports and so over its mapped nodes or anchors. If, however, the target is one context port, the action is done only to the element mapped by that port. The behavior of the switch node is close to the context node, except that the action is done only to the selected component node according to its rules.

It is worth to notice that a node is active, that is, being presented, if at least one of its anchors is active. So a content node is active if at least one anchor is active, a switch node is active if at least one component node is active and the same for the context node. A context node is also considered to be active if at least one link is being evaluated at that moment.

Whenever a content anchor is active, it is able to be selected by the user. An attribution action, however, is performed only over the property anchor indicated in the link bind.

NCM differs documents specification from their execution. NCM defines a *data plan* containing the nodes and links specified in a set of documents. Whenever those documents are executed, NCM defines a *representation plan.* A data object, which composes the data plan, represents an NCM node. Once a data object is associated to its presentation characteristics (specified by a descriptor), it becomes a representation object, which composes the representation plan. If the same data object is associated, for example, to two different descriptors, it will create two distinct representation objects. Figure 3.4 presents NCM data and representation plans considering a sample document.

Notice, in Figure 3.4, that each content node is associated to at least one descriptor,

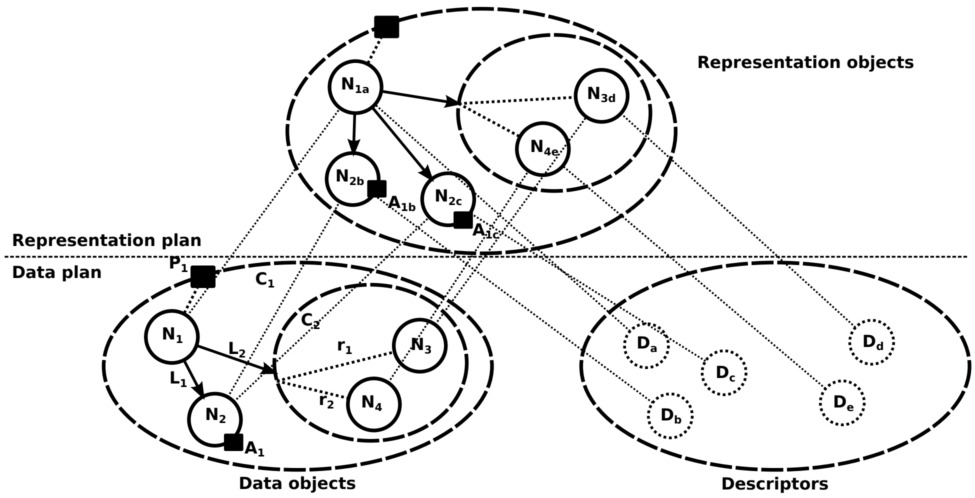


Figure 3.4: NCM data and representation plans

creating a representation object. Node N_2, however, is associated to descriptors D_b and D_c. That way two representation objects N_{2b} and N_{2c} are created.

3.3.2 NCL

NCL is an XML-based language that defines elements to represent NCM entities. An NCL document is divided in two parts: the document *head* and the document *body*. Listing 3.7 presents the structure of an NCL document. Listings 3.8 and 3.9 respectively show the *head* and *body* of an NCL document representing the NCM document illustrated in Figure 3.3.

Listing 3.7: NCL document example

```
<ncl id="runningExample" xmlns="http://www.ncl.org.br/NCL3.0/EDTVProfile">
   <head>
      ... head content ...
   </head>

   <body>
      ... body content ...
   </body>
</ncl>
```

The *ncl* element presented in Listing 3.7 is identified by *runningExample*. It represents an NCL document using the NCL 3.0 namespace *EDTVProfile* [ABNT 2011], which defines the XML elements that can be used in the document.

The document head defines connectors, rules and presentation characteristics that will be used in the document body element. Those definitions are done in bases, which are: *connectorBase*, *ruleBase*, *regionBase*, *descriptorBase* and *transitionBase*. Inside those bases, the NCL document defines connectors (*causalConnector* element), rules (*rule* element), regions (*region* element), descriptors (*descriptor* element) and transitions (*transition* element), respectively. Listing 3.8 presents an NCL document head example.

Listing 3.8: NCL document head example

```
<head>
   <ruleBase>
      <rule id="r1" var="A1" comparator="eq" value="yes" />
      <rule id="r2" var="A1" comparator="eq" value="no" />
   </ruleBase>

   <regionBase>
      <region id="reg1" left="0" top="0" width="100%" height="100%" zIndex="1">
         <region id="reg2" left="50" top="50" width="50" height="50" zIndex="2"/>
      </region>
   </regionBase>

   <descriptorBase>
      <descriptor id="desc1" region="reg1" explicitDur="90s"/>
      <descriptor id="desc2" region="reg2" explicitDur="20s"/>
   </descriptorBase>

   <connectorBase>
      <causalConnector id="onBeginSet">
         <simpleCondition role="onBegin"/>
         <simpleAction role="set" value="yes"/>
      </causalConnector>

      <causalConnector id="onEndStart">
         <simpleCondition role="onEnd"/>
         <simpleAction role="start"/>
      </causalConnector>
   </connectorBase>
</head>
```

The document head presented in Listing 3.8 defines two rules identified by *r1* and *r2*. Rule *r1* tests if the value of global variable *A1* is equal to “yes”, while rule *r2* tests if it is equal to “no”. It also defines two regions *reg1* and *reg2*. Region *reg1* specifies an area that occupies all the screen and region *reg2* defines an area whose upper left corner is (50, 50), width of 50 pixels and height of 50 pixels. Descriptor *desc1* uses region *reg1* and defines a duration of 90 seconds. Descriptor *desc2* uses region *reg2* and defines a duration of 20 seconds. The document head also defines two connectors *onBeginSet* and *onEndStart*. Connector *onBeginSet* defines a generic relation stating that “the beginning of the presentation of a node, related to the *onBegin* role, causes the attribution of the value ‘yes’ to an attribute anchor related to the *set* role” and connector *onEndStart* defines a generic relation stating that “the end of the presentation of a node, related to the *onEnd* role, causes the beginning of the presentation of another node, related to the *start* role".

The document body defines NCL nodes and links that composes the document. Content nodes are defined by the *media* element, context nodes are defines by the *context* element and switch nodes are defined by the *switch* element. Links are defined by the *link* element. Content node anchors are represented by elements *area* (content anchor) and *property* (attribute anchor). A context node port is represented by the *port* element.

Listing 3.9 presents an NCL document body example. Once the document body is a composite node, it represents content node *C1* of the example illustrated in Figure 3.3. The example defines port *P1* that maps to content node *N1* through its *component* attribute. Content node *N1* represent a video that will be presented as defined by descriptor *desc1* and content node *N2* defines an attribute anchor named *A1*, which represents the global variable with the same name inside the document. Switch *C2* defines content nodes *N3* and *N4* representing figures, both presented as defined by descriptor *desc2*. The association between a switch node component and a rule is done by element *bindRule*. The first *bindRule* associates node *N3* to rule *r1* and the second one associates node *N4* to rule *r2*. It is worth to notice that switch *bindRules* are evaluated in the same sequence they are defined.

Listing 3.9: NCL document body example

```
<body id="C1">
   <port id="P1" component="N1"/>

   <media id="N1" src="media/vid.mp4" descriptor="desc1"/>
   <media id="N2" type="application/x-ginga-settings">
      <property name="A1"/>
   </media>

   <switch id="C2">
      <bindRule rule="r1" constituent="N3"/>
      <bindRule rule="r2" constituent="N4"/>
      <media id="N3" src="media/fig1.png" descriptor="desc2"/>
      <media id="N4" src="media/fig2.png" descriptor="desc2"/>
   </switch>

   <link id="L1" xconnector="onBeginSet">
      <bind component="N1" role="onBegin"/>
      <bind component="N2" interface="A1" role="set"/>
   </link>
   <link id="L2" xconnector="onEndStart">
      <bind component="N1" role="onEnd"/>
      <bind component="C2" role="start"/>
   </link>
</body>
```

Listing 3.9 also presents two links *L1* and *L2*. Link *L1* uses the *onBeginSet* connector and associates node *N1* to role *onBegin* and anchor *A1* of node *N2* to role *set*. Link *L2* uses the *onEndStart* connector and associates node *N1* to role *onEnd* and node *C2* to role *start*.

3.4 Closing remarks

This chapter presented basic concepts related to multimedia documents and the different classes of temporal synchronization models. It also presented an overview of the NCM model and the NCL language.

As stated in [Blakowski and Steinmetz 1996], event-based models have a great *expressive power* [Pérez-Luque and Little 1996] to represent a temporal scenario. We believe that a model based on events is general enough to support a representation of other multimedia model temporal scenario descriptions, such as the work presented in [Rodrigues et al. 2002], that proposed a translation from a hierarchical-based synchronization authoring language (SMIL) to an event-based model (NCM).

Thus, the analysis of multimedia documents discussed in this work, that is, the validation of the document structural definition and verification of the document behavioral definition, follows an event-based temporal model. As a test case, this work implements an analysis tool capable of analyzing NCL documents. The choice for NCL was made since it is an international standard and it is also used in the Brazilian Digital TV system.

The next chapter presents a background on MDA (Model-drive Architecture).

Chapter 4

MDA background

This chapter introduces the MDA (Model-driven Architecture) [OMG 2003] concepts necessary to the comprehension of this work. Section 4.1 gives an overview of the MDA process used in this work. Section 4.2 introduces OCL (Object Constraint Language) [Warmer and Kleppe 1999] and how it is used in model validation. Section 4.3 introduces model verification with a model checker. Section 4.4 introduces rewriting logic, using Maude as a concrete syntax and Section 4.5 concludes this chapter with some remarks.

4.1 MDA

MDA [OMG 2003] proposes an approach to simplify software development and maintenance. In that approach, models are not only used for software design and documentation, but also for its development. MDA proposes the transformation of models describing software into some coding language, so any further modification on the software is done over its models. That approach guarantees that models and code are always updated. Once models present a language-independent software description, they also improve software reuse.

A *model*, as stated by [Mellor et al. 2003], is a coherent set of formal elements describing something built with some purpose. The description presented by a model follows a specific structure, which is given by a *metamodel*. A *metamodel* is the description of a modeling language used by a *model*. A *model* may be seen as an instance of a *metamodel*.

Suppose a metamodel of a person description. It defines the important information a model should have to describe a person. All models describing persons represent instances of that metamodel. Figure 4.1 illustrates that example.

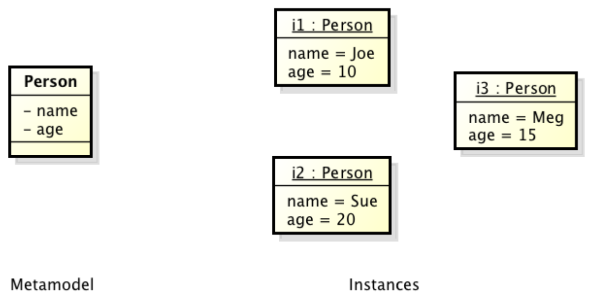


Figure 4.1: Metamodel and model example

A model is said to be *well-formed* with respect to a metamodel when it follows the syntax denoted by the metamodel. A *well-formed* model, however, can define incorrect relations among the modeling language elements. To constrain the use of the modeling language elements, the metamodel defines a set of additional properties through invariants. A model is said to be in *conformance* with a metamodel when the properties of that metamodel hold for the model.

The behavior of a model will be given by a transition system. A transition system is defined by a set of states and transitions defining a relation between states of the transition system.

Under a programing language perspective, a model transformation can be seen as a programing language compilation. A modeling language description has an abstract syntax; one or more concrete syntax; a mapping between the abstract syntax and a concrete syntax; and a description of its semantics. So a metamodel can be seen as the language abstract syntax and a model as a concrete syntax. Also, the metamodel invariants can be seen as the language static semantics and the transition system associated to a model can be seen as the language dynamic semantics. Similar to the process of program transformation, a model-driven development process can be presented as follows:

$$\begin{tikzcd} m \in M_{concrete\ syntax} \arrow[r, "parse"] & \hat{m} \in M_{abstract\ syntax} \arrow[dl, "\tau"'] \\ \hat{m}' \in M'_{abstract\ syntax} \arrow[r, "pretty\ print"] & m' \in M'_{concrete\ syntax} \end{tikzcd}$$

where m, m', $\hat{m}$ and $\hat{m}'$ represent models and M and M' represent the τ transformation source and target metamodels. $m \in M$ denotes that m is well-formed with respect to M. *parse* represents a mapping where a model m produces an instance $\hat{m}$ of M, where $\hat{m}$ is well-formed with respect to M. *pretty print* does the inverse mapping of *parse*.

In this work, UML [OMG 2010] was chosen for the representation of models and metamodels. A metamodel is mapped into a class diagram, while the model, instance of that metamodel, is mapped into an object diagram. Structural metamodel properties, that is, its invariants, are represented by OCL invariants [Warmer and Kleppe 1999]. The transition system associated with a modeling language is represented in rewriting logic. The behavioral properties are specified in temporal logic. Figure 4.2 summarizes the concepts presented in this section.

$$\begin{tikzcd}
\textit{Programming Language} & \textit{MDA} & \textit{This work} \\
\textit{Syntax} \arrow[d, dash] & \textit{Metamodel} \arrow[l] \arrow[d, dash] & \textit{Class Diagram} \arrow[l] \arrow[d, dash] \\
\textit{Static Semantics} \arrow[d, dash] & \textit{Invariants} \arrow[l] \arrow[d, dash] & \textit{OCL Invariants} \arrow[l] \arrow[d, dash] \\
\textit{Dynamic Semantics} & \textit{Transition System} \arrow[l] & \textit{Rewriting Logic} \arrow[l]
\end{tikzcd}$$

Figure 4.2: MDA concepts summary

4.2 OCL validation

OCL [Warmer and Kleppe 1999] is a language used to constrain class diagrams through invariants. An invariant defines a condition on the state of objects in an instance of the class diagram it constrains. OCL expressions are able to describe navigation paths through object associations and tests over object attribute values.

An invariant defines a context and an expression. The invariant context defines the type of the objects to which the invariant will be applied and the invariant expression defines a boolean expression that tests the object state. OCL invariants are defined based on the metamodel they constrain. Taking as example the NCL language, Listing 4.1 presents the definition of an invariant using OCL.

Listing 4.1: Invariant definition example

```
context RegionBase inv noEmptyBase:
    self.regions->notEmpty()
```

Listing 4.1 presents an invariant that is applied to objects of *RegionBase* type. The invariant *noEmptyBase* states that every object of type *RegionBase* should have at least one associated object with a role "*regions*". Operationally, this invariant gets the objects associated with a *regionBase* type object through its "*regions*" association. The expression *notEmpty()* tests if the returned object set is not empty. If this set is not empty, this expression will return true, so the invariant holds. Otherwise the expression returns false and the invariant fails.

OCL invariants describe properties that a model should have in order to be considered in conformance with the metamodel. Every invariant that can be applied over some model, that is, the invariants whose context is present in the model, will be applied. The model will be considered in conformance, if all applied invariants hold. In this work, we call model validation the application of invariants to a particular model.

4.3 Model checking

In this work, behavioral properties are described as temporal logic formulas over transition systems. The verification of behavioral properties is done using a technique called model checking which is essentially a decision procedure for temporal logic.

In this section we recall the fundamental concepts associated with temporal logic specifications. In Section 4.4 we discuss the rewriting logic formalism in which we will implement transition systems, their properties and verify them.

The behavior of a model is given by a transition system. A transition system $\mathcal{M}$ is defined by a set of states S and a relation $\rightarrow$ among the states of the transition system.

$$\begin{aligned} \mathcal{M} &= (S, \rightarrow) \\ \rightarrow &\subseteq S \times S \end{aligned}$$

Figure 4.3 presents an example of a transition system, where p, q and r represent properties.

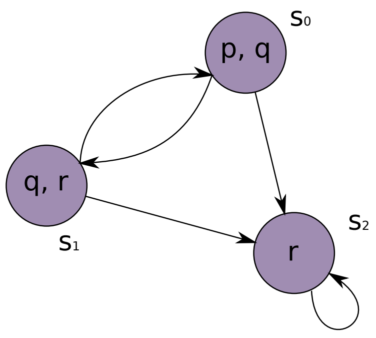


Figure 4.3: Sample transition system

In a transition system $\mathcal{M}$, a path π is an infinite state sequence $s_i \in S$, such that:

- for $i \geq 0, s_i \rightarrow s_{i+1}$;
- s_0 is the initial state of path π;
- $s \in S$ is reachable from s_0 if there is a path π, such that, s and $s_0 \in \pi$.

It is worth mentioning that different paths may have the same initial state. Figure 4.4 presents different paths of the sample transition system presented in Figure 4.3.

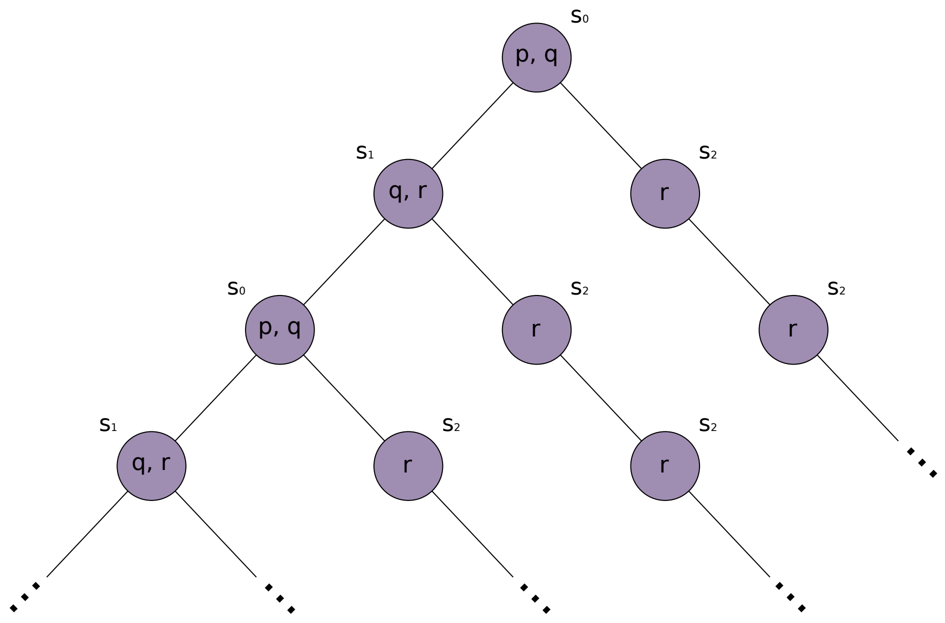


Figure 4.4: Sample transition system paths

LTL (Linear Temporal Logic) [Pnueli 1977] defines a set of logic operators with temporal semantics. The following sections introduce LTL. Section 4.3.1 introduces LTL operators, while Section 4.3.2 presents the verification of LTL formulas over paths.

4.3.1 LTL operators

An LTL formula φ is defined as follows, where X, F, G, U, W and R are called temporal operators.

$$\begin{aligned}\varphi \quad ::= \quad & \top \mid \bot \mid p \mid \neg(\varphi) \mid (\varphi \wedge \varphi) \mid (\varphi \vee \varphi) \mid (\varphi \rightarrow \varphi) \mid \\ & (X\varphi) \mid (F\varphi) \mid (G\varphi) \mid (\varphi\ U\varphi) \mid (\varphi\ W\varphi) \mid (\varphi\ R\varphi)\end{aligned}$$

A concurrent system may have different paths from the same initial state. To illustrate the temporal operators, this section presents their application to a single path.

Operator X (next) states that a property must be valid for the following state. Figure 4.5 presents an example of operator next applied to state s_0.

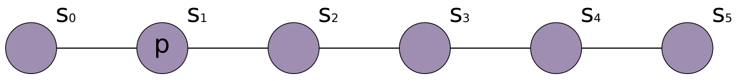


Figure 4.5: Next operator example (Xp)

Operator F (future) states that a property must be valid for some future state. Figure 4.6 presents an example of operator future applied to state s_0.

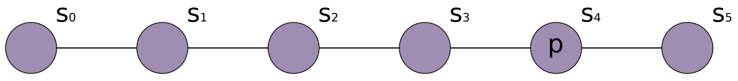


Figure 4.6: Future operator example (Fp)

Operator G (global) states that a property must be valid for all states in a path. Figure 4.7 presents an example of operator global.

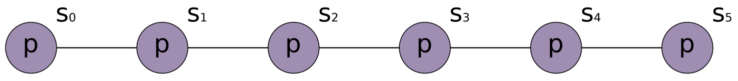


Figure 4.7: Global operator example (Gp)

Operator U (until) states that a property p must be valid until a property q becomes valid. Figure 4.8 presents an example of operator until.

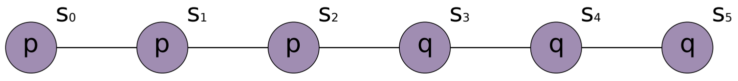


Figure 4.8: Until operator example (pUq)

Operator W (weak until) states that a property p must be valid until a property q becomes valid or p must be valid for all states in the path. Figure 4.9 presents an example of operator weak until.

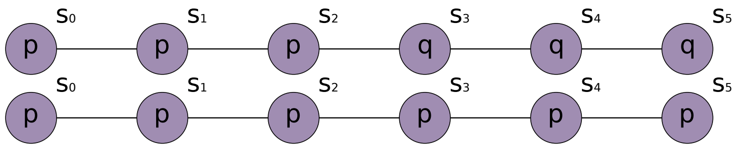


Figure 4.9: Weak until operator example (pWq)

Operator R (release) states that a property p must be valid until a property q becomes valid and both p and q must be valid at the same time for some state. Figure 4.10 presents an example of operator release.

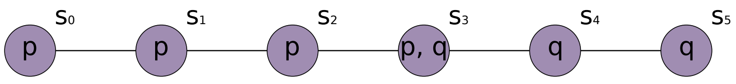
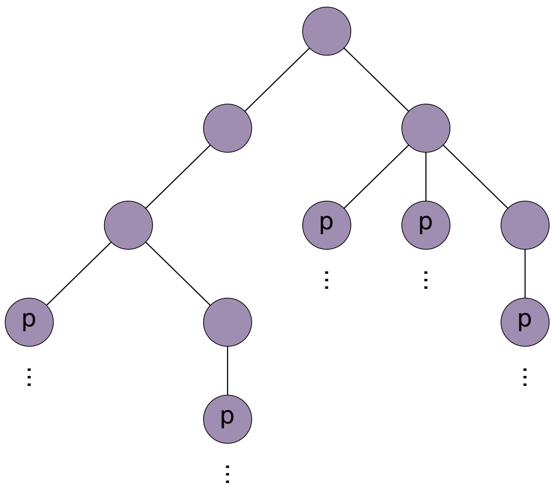


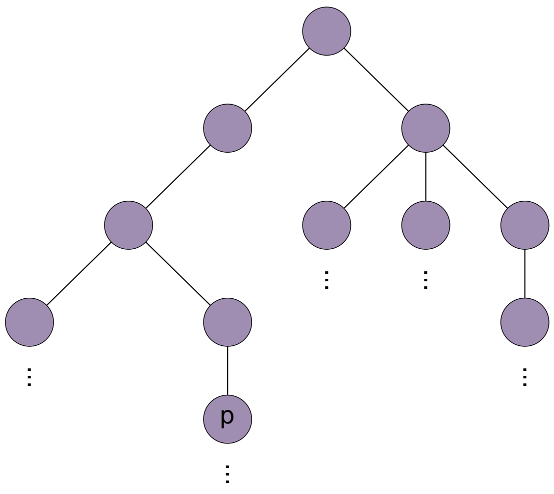
Figure 4.10: Release operator example (qRp)

4.3.2 LTL verification over paths

LTL logical operators define properties verifiable for all paths from the state where the property is tested. Figure 4.11 presents the verification of property Fp over some paths with the same initial state. Figure 4.11a presents a valid example and Figure 4.11b an invalid one.



(a) Valid example



(b) Invalid example

Figure 4.11: Property verification over transition system paths (Fp)

The verification of the existence of a path where a property is valid is achieved through the negation of an LTL formula. The negated formula verification produces a counter-example that exemplifies a possible path where the property holds. Figure 4.12 presents the verification of property $\neg Fp$ over the paths presented in Figure 4.11b.

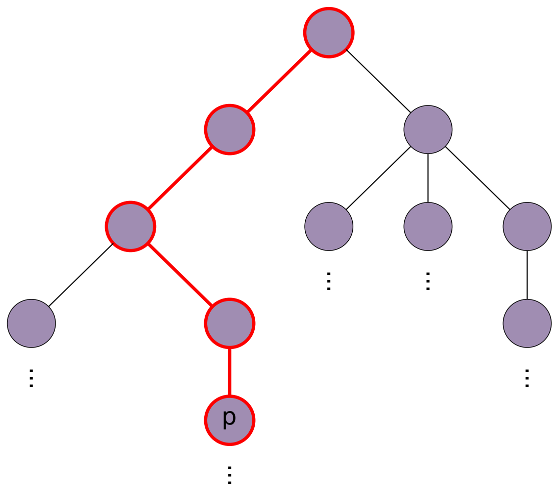


Figure 4.12: Property verification over transition system paths ($\neg Fp$)

4.4 Rewriting logic

In this work we will use rewriting logic and its implementation in the Maude system to represent transition systems and behavioral properties. The LTL model checker implemented in the Maude system will be used to investigate behavioral properties of transition systems.

Rewriting logic consists of a set of rewrite rules defining the concurrent evolution of a system. A theory in rewriting logic, or rewrite theory, is defined as follows.

$$\mathcal{R} = (\Sigma, E, R)$$

A rewrite theory is divided in two parts. The first one (Σ, E) is the equational theory part, while the second one R is the rewrite rules. This section presents the concepts related to rewrite theory using, as a concrete syntax, the Maude language [Clavel et al. 2007]. It is a declarative language and a system for the description of a theory in rewriting logic.

In this section we will present rewrite theories from a transition system perspective as it is the one used in this work.

A rewrite theory is described in terms of *sorts*, *operations*, *equations* and *rules*. *Sorts* are used to define data types and how those types are related through set theory. *Operations* define how elements on those sorts are created and manipulated. *Equations* define a simplification over the elements that compose the system state. Every simplification is applied until the system canonical form is reached, that is, when no other simplification over the system state can be obtained. *Rules* define the system evolution. They are applied, possibly concurrently, changing the system state. A rewrite theory is assumed coherent, which means that no equational reduction or rewrite is missed. Operationally, this is achieved in Maude by first applying the equations and then the rules, that is, rules are applied modulo equations.

Regarding a rewrite theory definition (Σ, E, R), Σ represents the system signature and is described by sorts and operations, while E represents the system equations and R, the system rules. Listing 4.2 presents an example of sorts, operations, equations and rules described in Maude.

Listing 4.2: Maude primitives example

```
--- sorts
sorts A B .
subsort A < B .

--- operations
ops a1 a2 : -> A .
ops b1 b2 : -> B .
op f_ : A -> B .

--- equations
eq f(a1) = b1 .
eq f(a2) = b2 .

--- rules
rl b1 => a2 .
rl b1 => b2 .
```

Listing 4.2 defines two sorts A and B, where $A \subseteq B$. It defines operations for the creation of elements a_1 and a_2 of sort A and elements b_1 and b_2 of sort B. It also defines operation f, of type B which is parameterized by an element of A. The equations define reductions for the f operation, that is, $f(a_1)$ is reduced to b_1 and $f(a_2)$ is reduced to b_2. Rules define a modification in the system state. Every time the canonical form presents element b_1, it can be rewritten to a_2 and b_2 concurrently. Listing 4.3 demonstrates how equations and rules work by presenting the result of Maude `reduce` and `rewrite` commands.

The `reduce` command invokes the equations only. The `rewrite` command applies the rules over equationally simplified terms.

Listing 4.3: Maude commands result

```
Maude> reduce a1 .
result A: a1

```

```
Maude> reduce b1 .
result B: b1

Maude> reduce f(a1) .
result B: b1

Maude> rewrite b1 .
result A: a2

Maude> rewrite f(a1) .
result A: a2
```

Listing 4.2 defines two concurrent rules but in Listing 4.3 only the result on the first rule is presented. That occurs because the `rewrite` command always shows the first solution obtained with one rewrite command. Listing 4.4 presents all possible results of command `rewrite f(a1)`. This is done using a command to search all results from a term.

Listing 4.4: Maude rewrite results

```
Maude> search f(a1) =>* b:B .

Solution 1 (state 0)
b:B --> b1

Solution 2 (state 1)
b:B --> a2

Solution 3 (state 2)
b:B --> b2

No more solutions.
```

Listing 4.4 presents three solutions, the first one where no rule was applied, the second one where the first rule was applied and the third one where the second rule was applied. It is worth noticing that a rule is applied only when the system reaches the canonical form.

Equations and rules are applied when their left side matches at least a fragment of the system state. This is called *pattern matching*. Maude *pattern matching* can be defined over elements, as presented in the previous examples, but also to a general signature. Listing 4.5 presents an example of equation whose left side defines a general signature.

Listing 4.5: Maude pattern matching example

```
op g_ : B -> B .

var varb : B .
```

```

eq g(varb) = varb .
```

Listing 4.5 defines an operation g over set B, a variable $varb$ of type B and a equation the simplifies an element $g(varb)$, where $varb$ represents every element of sort B, to the element represented by $varb$. Listing 4.6 shows examples using that equation.

Listing 4.6: Maude pattern matching result

```
Maude> reduce g(b1) .
result B: b1

Maude> reduce g(b2) .
result B: b2
```

For conditional equations and rules, a condition must be satisfied so the equation or rule is applied. Listing 9.5 presents an example of a conditional equation. The result of that equation application is presented in Listing 4.8

Listing 4.7: Maude conditional equation example

```
ceq g(varb) = varb if varb == b1 .
```

Listing 4.8: Maude conditional equation result

```
Maude> reduce g(b1) .
result B: b1

Maude> reduce g(b2) .
result B: g b2
```

Notice in Listing 4.8 that just the first `reduce` command is applied, since it is the only one that satisfies the equation condition.

A Maude rewrite theory is divided in modules, which can be a *functional module* or a *system module*. *Functional modules* define data types (sorts) and operations over those types. That is, a functional module defines sorts and how those types are related through set theory and operations that may be done over them (Σ part). Functional module equations define a simplification, over the sorts, of the system state (E part). *System modules* define a rewrite theory, by defining data types, operations and equations over them (E part) and rewrite rules (R part).

A possible interpretation of a rewrite theory is to associate a transition system with it, where sorts, operations and equations define the signature of the transition system states and transitions are induced by rules. Because of that characteristic, rewriting logic - and Maude as a concrete syntax - was chosen to support the representation of a transition system modeling a multimedia document behavior. Formally, a transition system $\mathcal{M} = (S, \rightarrow)$ is represented as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ as follows.

$$\begin{aligned} S &= (\Sigma, E) \\ \rightarrow &\subseteq S \times S = R \end{aligned}$$

4.5 Closing remarks

This chapter introduced MDA and how model validation and verification are done. In this work an MDA approach is used to achieve the analysis of multimedia documents specified using the NCM model and NCL language.

The proposed analysis represents the NCL language structure as a metamodel and OCL invariants to represent properties a document must have to be considered consistent. The document itself is represented as a model, instance of the metamodel representing the language structure. The language and document representations and the validation of an NCL document will be presented in more details in Chapter 6.

Besides, this work uses a transition system representation of that document to investigate temporal properties. The transition system is represented using rewriting logic and the temporal properties are represented in LTL. The verification of those temporal properties is achieved by using a model checker implemented in Maude. It is possible to observe, that during model checking, an explosion of the number of states may occur. This concern was taken into consideration when creating the representation of the document, as it will be presented later in this work. The document representation and the temporal properties checking will be presented in more details in Chapter 7.

The next chapter presents the validation and verification properties and a comparison among related works regarding the identified properties.

Chapter 5

Validation and verification properties

This work presents a solution for analyzing a multimedia document in order to guarantee its consistency. We consider a document consistent, when it does not present specification problems or an undesired behavior. The analysis presented here is divided into two parts: the validation of the document structure and the verification of the document behavior, where each part defines a set of properties. Considering the language perspective discussed in Chapter 4, those properties will be called *static* or *dynamic*, making reference to static and dynamic semantics of a modeling language. It is worth mentioning that the properties presented here come from the study of the related works presented in Chapter 2.

Section 5.1 presents the static properties, while Section 5.2 presents the dynamic properties. Section 5.3 discusses related work regarding the static and dynamic properties and Section 5.4 concludes this chapter with final remarks.

5.1 Static validation properties

Static validation properties are the ones responsible for validating the document structure, that is, if the multimedia document created by the author satisfies the syntax rules defined by the authoring language grammar and structural invariants. Although those properties were described in a general manner, their implementation is domain dependent, since the properties must be validated over specific multimedia language elements. The static properties are: lexical and syntactic, hierarchy, attribute, reference, compositionality, composition nesting and element reuse validation properties. The following paragraphs present those properties.

The lexical and syntactic validation property specifies that the multimedia document lexical and syntactic structure should be well-formed with respect to the authoring language specification. For example, when an XML-based language is used, such a property specifies that the XML tags used should be defined in the language namespace and should be correctly closed.

The hierarchy validation property specifies that all language elements should contain valid child elements and in the correct cardinality. It also specifies that required child elements should not be missing.

The attribute validation property specifies that all language elements should contain valid attributes and the required attributes should be defined. In some elements, attribute pairs have related values. For example, suppose an XML element with attributes “type” and “subtype”. In that case the value of the attribute “subtype” must be related to the value of attribute “type”. This validation property specifies that such relation should be correct. It also specifies that each element identifier should be unique.

It is worth to highlight that the XML concepts of well-formed and valid documents [W3C 2008a] are guaranteed by the lexical and syntactic, hierarchy and attribute properties. Those properties can be applied to any document specified using an XML notation like HTML5 [W3C 2011]. Besides, the hierarchy and attribute properties can also be applied to documents where language elements have hierarchy and attributes with related values, like NCL, HTML and SMIL languages.

The reference validation property specifies that, for each attribute that refers to another element, the element whose identifier was referenced should match the required attribute type. For example, suppose an NCL element with attributes “node” and “anchor”. In that case, the element referenced by attribute “anchor” must be a child of the element referenced by attribute “node”.

The compositionality validation property specifies that composition component elements and their attributes should not refer to elements outside the composition. For example, suppose an NCL context port. That port can not refer to a node outside the context it is defined. Another example is the use of a SMIL *par* container *endsync* attribute that can refer to a container child component, determining that the container will end when its child component ends. This container can not refer to a component outside of it.

The composition nesting validation property specifies that a composition should not create a nesting loop. If this property is not valid, one composition alone, or through a set of compositions, nests itself. NCM and caT enables the reuse of composition elements (contexts and switches for NCM and places for caT). In both of them, those elements can not nest themselves, otherwise the multimedia document would become inconsistent.

The element reuse validation property specifies that an element should not create a reuse loop. If this property is not valid, one element alone, or through a set of elements, reuses itself. For example, NCL media elements can not make a reference, through its "refer" attribute, to itself.

5.2 Dynamic verification properties

Dynamic verification properties are the ones responsible for investigating the document behavior. It is worth noticing that those properties investigates if the document presents a possible undesired behavior. The dynamic properties are: reachability, anchor termination, document termination and resource verification properties.

The following paragraphs present those properties with a high level of abstraction. In later chapters, we present the model used to represent the behavioral definition of a multimedia document and the representation of the dynamic properties over that model. With such a detailed presentation of the dynamic properties it will be possible to observe that their verification is decidable.

The reachability verification property specifies that there should be no unreachable document parts. Unreachable document parts may occur when a document does not define relationships referring to some elements or if content control elements have alternative conditions that will never be evaluated as true. This may happen, for example, when a variable responsible for an alternative content evaluation never has its value changed.

The anchor termination verification property specifies that a document element (anchor), once started, should end. A document element may end by its natural end or by an external cause. When an element is ended by an external cause, it is desired that it occurs after the element is started. For example, consider two elements A and B, where A has a natural end and B does not. If the end of element A ends element B, it is important that A ends after B starts. Suppose two temporal intervals in a constraint-based synchronization model. If there is a relation of the type A *finishes* B and A ends before B starts, then B will never end. Figure 5.1 presents an end event that never occurs.

The document termination verification property specifies that the multimedia document, in at least one possible execution, as a whole should end. This specifies that there should be one configuration where all document elements remain finished. This configuration is achieved if all document elements end and there are no loops in the document. Loops occur when an element once stopped, triggers its own presentation begin and no other event aborts its presentation exiting the loop. Examples of loops without exit conditions are: the use of links in NCL that start an element once it ends; or the use of repeat attributes in SMIL elements and the document does not present a possibility of stoping those elements. Figure 5.1 also presents the concept of loops in an event-based document model.

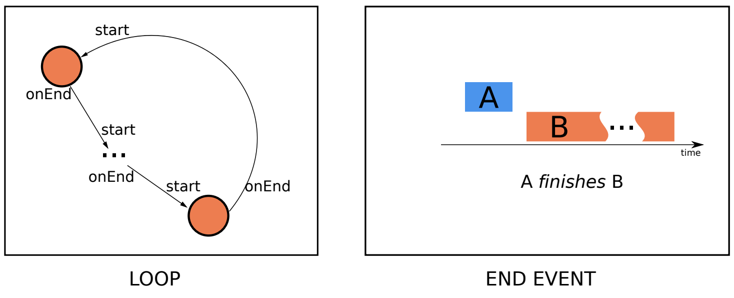


Figure 5.1: Examples of possible undesired behaviors.

The resource verification property investigates if two distinct media nodes use the same presentation resource at the same time. This resource may be, for example, a screen or an audio device. For spatial verification, it investigates if two visual media nodes are presented overlapped. For audio verification, it investigates if two media nodes with audio content (music and video, for example) are presented at the same time.

5.3 Related work comparison

This work proposes a set of desired properties a multimedia document should present, they are the validation and verification properties presented in Sections 5.1 and 5.2. This section uses those properties to evaluate the related work. This comparison is functional, since the great part of the tools is not available for practical tests.

The investigation of the set of possible undesired behaviors proposed by [Santos et al. 1998] meets the verification properties of reachability, anchor termination, document termination and resource. Although it is not the focus of the paper, it does not make clear, however, if, during the translation of the multimedia document into the RT-LOTOS formalism, the document is analyzed regarding any of the static validation properties.

In Trellis [Furuta and Stotts 2001] and caT [Na and Furuta 2001], the Petri net analysis provided meets the verification properties of reachability and document termination.

Although it is not the focus of the caT system, since the authoring of a hypermedia document is done directly in Petri nets, the validation properties of lexical and syntactic, hierarchy, attribute, reference, compositionality and composition nesting are embedded in the authoring tool.

The HMBS model [Oliveira et al. 2001] provides characteristics that meet the reachability, anchor termination and document termination verification properties. As it also provides an analysis of a best application layout, it can be considered as an approach to meet the resource verification property. Like caT, in HMBS, the hypermedia document authoring is done directly using statecharts, so the validation properties of lexical and syntactic, hierarchy, attribute, reference and compositionality are embedded in the authoring tool.

NCL-FA [Felix 2004] creates a timed automata net representing the NCL document temporal behavior. The author defines temporal-logic formulas in order to investigate the temporal properties. All the verification properties can be met, since the author is free to define any temporal-logic formulas. The static validation properties are not met since they are not the focus of that work.

SMIL-EA [Bossi and Gaggi 2007] uses assertions to investigate if the SMIL tags correctly changed the application state. With that assertive analysis, it is possible to meet the reachability, anchor termination and document termination verification properties. The static validation properties are not met since they are not the focus of that work.

In caT and SMIL-EA, it was not identified if the tools could be used to meet the resource verification property.

In BFPS [Bertino et al. 2005], the checking done during the authoring process meets the anchor termination and document termination verification properties. Also, as in BFPS the spatial positioning is done analyzing the space available on the screen at each instant and dividing it in rectangles where objects may be inserted, the resource verification property is also met. The static validation properties are not met since they are not the focus of that work.

EEC [Elias et al. 2006] finds the minimum spanning tree of the graph defined by the application to check its consistency. The acyclic nature of the spanning tree and the completeness checking are sufficient to meet the anchor termination and document termination verification properties. The tool meets the resource verification property with its spatial consistency checking using the *SPATIAL* operator. The static validation properties are not met since they are not the focus of that work.

In BFPS and EEC approaches, it was not identified, however, if the tools could be used to meet the reachability verification property.

With the analysis process presented by NCL-validator [Araújo et al. 2008] it is possible to meet the lexical and syntactic, hierarchy, reference and compositionality validation properties and partially meet the attribute validation property. The dynamic verification properties are not met since they are not the focus of that work.

NCL-Inspector [Honorato and Barbosa 2010] allows the analysis of an NCL application. With the rules used by NCL-Inspector to inspect a document, it is possible to meet the lexical and syntactic, hierarchy and attribute validation properties. It was not possible to determine, if the tool can meet the other validation properties. The dynamic verification properties are not met since they are not the focus of that work.

The related works presented generally focus on either the validation of the document structure or the verification of the document behavior. aNaa (API for NCL Authoring and Analysis), the analysis tool proposed in this work, focuses on both, meeting all static validation and dynamic verification properties.

Table 5.1 summarizes the related work comparison regarding the validation and verification properties. The "-" sign represents partially meet properties and the "e" sign represents properties embedded in the authoring tool.

Table 5.1: Summary of the related work comparison

		RT-LOTOS	caT	HMBS	NCL-FA	SMIL-EA	BFPS	EEC	NCL-validator	NCL-Inspector	aNaa
Static	Lexical and syntactic		e	e					✓	✓	✓
	Hierarchy		e	e					✓	✓	✓
	Attribute		e	e					-	✓	✓
	Reference		e	e					✓		✓
	Compositionality		e	e					✓		✓
	Composition nesting		e								✓
	Element reuse										✓
Dynamic	Reachability	✓	✓	✓	✓	✓					✓
	Anchor Termination	✓		✓	✓	✓	✓	✓			✓
	Document Termination	✓	✓	✓	✓	✓	✓	✓			✓
	Resource	✓		✓	✓		✓	✓			✓

✓ : fully met property;

- : partially met property;

e : embedded property.

5.4 Closing remarks

The set of properties presented in this chapter came from the study of related work. Since the related works here presented comprises different multimedia temporal models, we believe that the set of properties proposed in this work is general enough to be applied to other multimedia models different than NCM.

In addition, it is important to investigate if the set of properties is complete or not. It is possible that the study of related work could not identify all document characteristics that could create specification errors or possible undesired behaviors. This set, however, may be enlarged as different inconsistencies are identified. In order to help the study of the property set completeness, it is important to use those properties to analyze documents specified in different multimedia models, which will be done as future works.

The properties here identified will be used in the document validation and verification as will be presented in the next two chapters.

Chapter 6

NCL static semantics

The proposed static validation properties should be applied to a specific multimedia document in order to be checked. An analysis tool should check those properties focusing on the multimedia language to be validated, since they depend on the language syntax.

In this work, the static validation is done using model validation. To achieve that, the multimedia language syntax to be validated is represented as a metamodel, while the language static semantics is represented by that metamodel properties. In our case, the language metamodel is represented as a class diagram and the metamodel properties as OCL invariants [Warmer and Kleppe 1999]. The model to be validated, that is, the multimedia document, is represented as an object diagram.

It is worth mentioning that OCL invariants represent the set of static validation properties. So, if the OCL invariants hold, it is equivalent to say that the static validation properties also hold and the document is considered valid.

As a test case, this work implemented the validation of the static properties for documents defined by the NCM model [Soares et al. 2000, Soares and Rodrigues 2005], using the NCL language [ITU 2009]. The following sections present the validation of NCL documents.

The remaining of this chapter is structured as follows. Section 6.1 presents an overview of the NCL language syntax metamodel. Section 6.2 presents the OCL invariants used for the static validation. Section 6.3 presents the representation of the document to be validated as an object diagram and Section 6.4 presents some closing remarks.

6.1 NCL Language structure metamodel

The metamodel used for static property validation is defined based on the NCL language structure. Since it is represented as a class diagram, each XML element of the NCL language is represented as a class. Each class contains the same attributes of the XML element it represents. For an XML element that has child elements, the class that represents it will have associations between that element and its children. In this way, hierarchical relations among elements are represented, making it possible to navigate from an element to its children and back.

NCL defines element attributes that refer to other elements. That reference is usually done by defining the attribute value equal to the referenced element id or another attribute. In order to improve navigation over NCL elements, the metamodel structure represents those references as associations between element classes. Listing 6.1 presents an example of reference between XML elements and Figure 6.1 presents how those elements are represented in the metamodel.

Listing 6.1: Example of reference between elements

```
<regionBase>
   <region id="reg1" left="0" top="0" width="100%" height="100%"/>
</regionBase>

<descriptorBase>
   <descriptor id="desc1" region="reg1"/>
</descriptorBase>
```

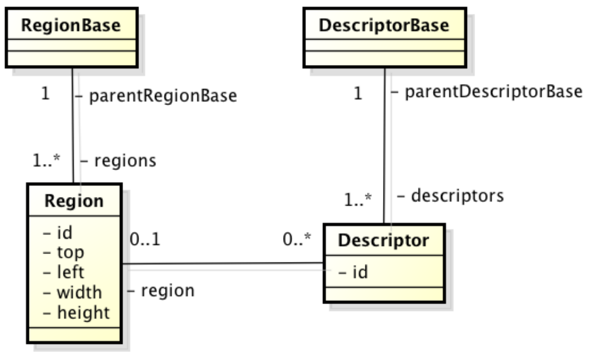


Figure 6.1: Metamodel representation

Listing 6.1 presents a fragment of the NCL document example from Section 3.3.2. That example defines a *regionBase* element, which has *region* elements as children. The *region* element defines a set of attributes, which are: id, top, left, width and height. The id attribute is a string that represents the *region* identification and the top, left, width and height attributes are integers that represent the *region* location on the screen. The example also defines a *descriptorBase* element, which has *descriptor* elements as children. The *descriptor* element defines two attributes, id and region. The id attribute is a string that represents the *descriptor* identification and the region attribute is a string that represents the identification of the *region* element used by that *descriptor* element. So, in this example, descriptor element "desc1" refers to region element "reg1".

In Figure 6.1, it is possible to observe that every XML element is represented by a class. Each element class has the same attributes as the XML element and each element is associated to its children (see *regionBase* and *descriptorBase*). Notice that the *region* attribute of the *descriptor* element is represented as an association between the Descriptor and Region classes. That way, it is easy to follow element references inside a document. Figure 6.2 presents a part of the NCL language structure metamodel (LSM) representing the relations among elements of the NCL document head and Figure 6.3 does the same for the NCL document body.

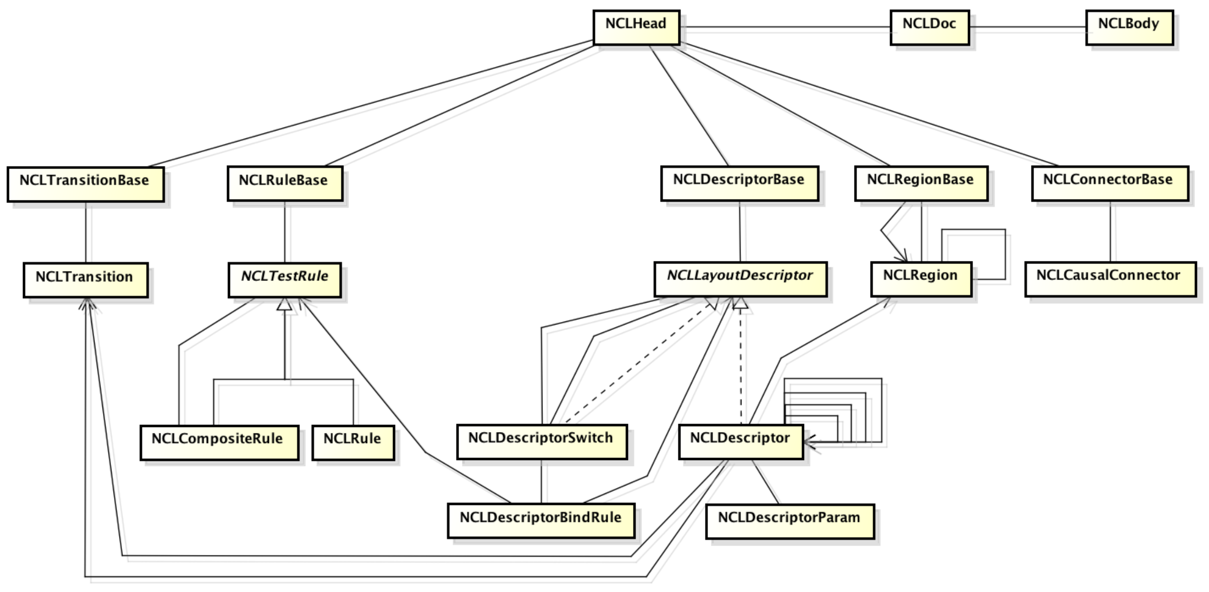


Figure 6.2: Structure metamodel of NCL document head

The NCL LSM here presented is used during the static validation of an NCL document. The following section presents how static validation properties are represented as OCL invariants.

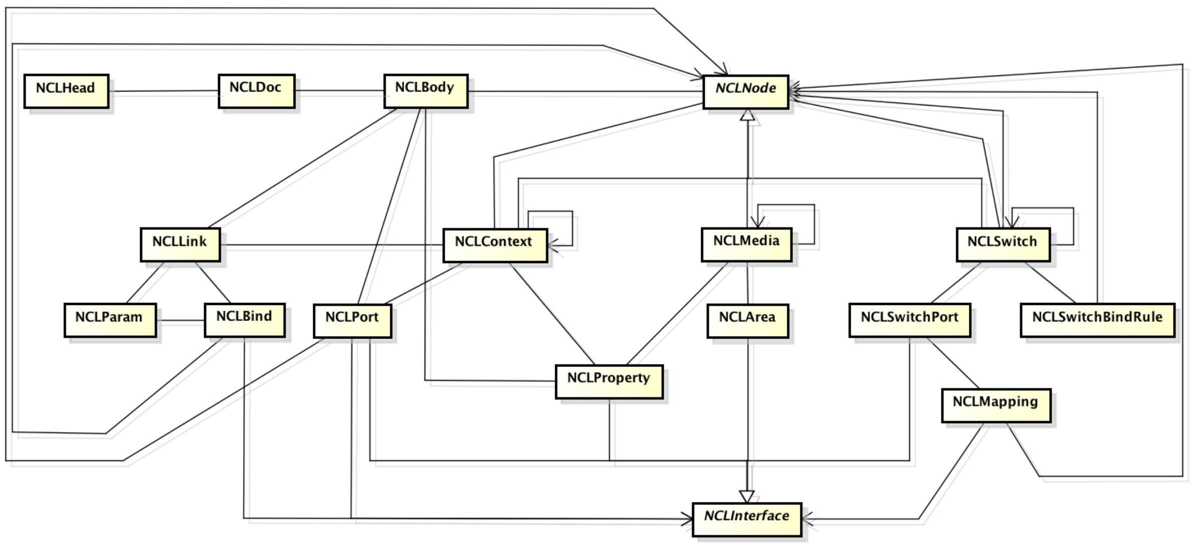


Figure 6.3: Structure metamodel of NCL document body

6.2 Validation invariants

Static validation properties are defined as general properties. In order to be able to investigate those properties, the validation tool has to specialize those properties for the elements in the Language Structure Metamodel (LSM). The OCL invariants used for validating the document will represent the hierarchy, attribute, reference, compositionality, composition nesting and element reuse validation properties. The remaining of this section presents how OCL was used to represent those properties and also give invariant examples.

The hierarchy property is represented by a set of invariants that test the required cardinality of child elements. Notice that while testing the cardinality of the child elements, it already tests if some required child element is missing. The attribute property is represented by a set of invariants that test if required attributes are defined. When representing those two properties with OCL invariants, it was necessary to create a base of OCL invariants with one invariant for each element class to be tested. Since the number of invariants is big, Listing 6.2 presents just two examples of invariants that test the properties described above. Appendix A lists the complete set of OCL invariants.

Listing 6.2: Hierarchy and Attribute properties invariant example

```
context NCLLink inv:
    self.binds->size() >= 2

context NCLLink inv:
    self.xconnector->notEmpty()
```

The first invariant in Listing 6.2 constrains that the Link element should have at least two Bind elements as children. The second invariant constraints that the Link attribute "xconnector" should be defined. Those two invariants implement hierarchy and attribute validation properties, respectively.

The attribute validation property also tests if related attributes have the correct values and XML element identifiers are unique. The related attribute value validation is done with one invariant, for each pair of attributes, that tests if the values of both attributes are related. Regarding the unique identifier, NCL defines domains into which the element identifier must be unique. For some elements, like properties and connector roles, their identifier must be unique among the children elements of the same parent. For the remaining elements that have identifiers, they must be unique inside the whole document. The unique identifier validation is done with one invariant, for each domain, that tests if the element identifier is unique. Listing 6.3 presents an example of invariant that tests related attributes and Listing 6.4, invariants that test if element identifiers are unique.

Listing 6.3: Attribute relation invariant example

```
context NCLSimpleAction inv:
    self.repeatDelay->notEmpty() implies self.repeat->notEmpty()
```

The invariant in Listing 6.3 constrains that if the SimpleAction element defines the "repeatDelay" attribute then the "repeat" attribute must be defined.

Listing 6.4: Unique identifier invariants

```
context NCLMedia inv:
    self.properties->forAll(p1 : NCLProperty, p2 : NCLProperty | p1 <> p2 implies
        p1.name <> p2.name)

context NCLCausalConnector inv:
    self.getRolesFromCausalConnector()->forAll(r1, r2 | r1 <> r2 implies r1.differ(r2)))

context NCLIdentifiableElement inv:
    if not self.oclIsTypeOf(NCLProperty) and self.id->notEmpty() then
        NCLIdentifiableElement.allInstances()->forAll(d |
            if d.id->notEmpty() and d <> i then
                d.id.value <> self.id.value
            else
                true
            endif)
    else
        true
    endif
```

Listing 6.4 presents three invariants that constrains the elements Media, CausalConnector and all IdentifiableElement, respectively. It is worth to notice that, in NCL LSM, every element that has an "id" attribute is an IdentifiableElement. That is why the unique identifier validation could be represented by just one invariant. Those invariant states that the "name" attribute of all properties inside a Media element must be unique, the "role" attribute of all elements inside a CausalConnector element must be unique and that the "id" attribute of all elements that have one, inside the document, must be unique.

The reference property is represented by a set of invariants that test if attributes that refer to other elements refer to the correct element. For each NCL element that has this type of attribute, one invariant will be created. Listing 6.5 presents an example of invariant that implements the reference property.

Listing 6.5: Reference property invariant example

```
context NCLLink inv:
    self.binds.role->forAll(r |
        self.xconnector.getRolesFromCausalConnector()->includes(r))
```

The invariant presented in Listing 6.5 constrains that the "role" attribute of all Bind elements inside a Link element must be a role defined by the connector referenced by the "xconnector" attribute of the link.

The compositionality property is represented by a set of invariants that test if attributes that must refer to other elements inside the same composition make that reference correctly. For each NCL element that has that type of attribute, one invariant will be created. Listing 6.6 presents an example of invariant that implements the compositionality property.

Listing 6.6: Compositionality property invariant example

```
context NCLPort inv:
    if self.parentBody->notEmpty() then
        self.parentBody.nodes->exists(a | a = self.component)
    else
        self.parentContext.nodes->exists(a | a = self.component)
    endif
```

The invariant presented in Listing 6.6 constrains that the Port element must define the "component" attribute referring to a node element inside the composition it is defined. Notice that the invariant tests if the node referred by the "component" attribute is a child element of its parent, which can be a Body or a Context element.

The last two properties, composition nesting and element reuse properties, are represented by a set of invariants that walks through the elements associations, testing if a composition nests itself or if an element reuses itself. Listing 6.7 presents an example of invariant for both properties.

Listing 6.7: Composition nesting and element reuse properties invariant example

```
context NCLSwitch inv:
    self.nodes->forAll(n:NCLNode | n.loops())

context NCLSwitch inv:
    self.refer->forAll(s:NCLSwitch | s.oclAsType(NCLNode).referLoops()))
```

The first invariant in Listing 6.7 constrains that the Switch element must not nest itself. This invariant walks through the association among a Switch element and its parent element and from it to its parent, testing if the Switch is one of those parent elements. This test is done by the *loops()* operation. Listing 6.8 presents the *loops()* operation definition.

Listing 6.8: loops() operation definition

```
context NCLNode::loops():Boolean body:
    if self.hasParent() then
        self.oclAsType(NCLElement).getParent() .verifyLoop(self)
    else
        true
    endif

context NCLElement::verifyLoop(node:NCLElement) :Boolean body:
    if self = node then
        false
    else
        if self.hasParent() then
            self.getParent().verifyLoop(self)
        else
            true
        endif
    endif
```

The second invariant in Listing 6.7 constrains that the Switch element must not refer to itself. This invariant walks through the association among a Switch element and its referred element and from it to its referred element and so on, testing if it is one of those referred elements. This test is done by the *referLoops()* operation. Listing 6.9 presents the *referLoops()* operation definition.

Listing 6.9: referLoops() operation definition

```
context NCLNode::referLoops():Boolean body:
    if self.oclAsType(NCLElement).hasRefer() then
        self.oclAsType(NCLElement).getRefer().verifyReferLoop(self)
    else
        true
    endif


context NCLElement::verifyReferLoop(node:NCLElement):Boolean body:
    if self = node then
        false
    else
        if self.oclAsType(NCLElement).hasRefer() then
            self.oclAsType(NCLElement).getRefer().verifyLoop(self)
        else
            true
        endif
    endif
```

Notice that the only validation property that is not represented by OCL invariants is the lexical and syntactic property. Since the OCL invariants are applied over an object diagram, this validation should be done before the transformation of a document into an object diagram, as will be presented in Chapter 8.

This section presented a few invariants used for the model validation, Appendix A presents the complete list of invariants. The following section presents the representation of an NCL document as an object diagram.

6.3 Document representation

In order to validate an NCL document, it must be represented as an object diagram. This diagram represents an instance of the NCL LSM. Each element in the document is represented by an object. This object defines attribute values the element has and is associated with the objects that represent the other document elements it is related to (child/parent relation or reference).

Regarding the example presented in Section 3.3.2, the object diagram that represents that document should present objects representing the document elements (see Listings 3.7, 3.8 and 3.9). These objects define attribute values based on the XML element attribute values and associations according to elements hierarchy and reference between elements. Figure 6.4 presents the representation of that document head as an object diagram and Figure 6.5 presents the representation of that document body.

The object diagrams presented in Figures 6.4 and 6.5 show the objects that represent NCL document elements and their attributes. Associations representing elements hierar-

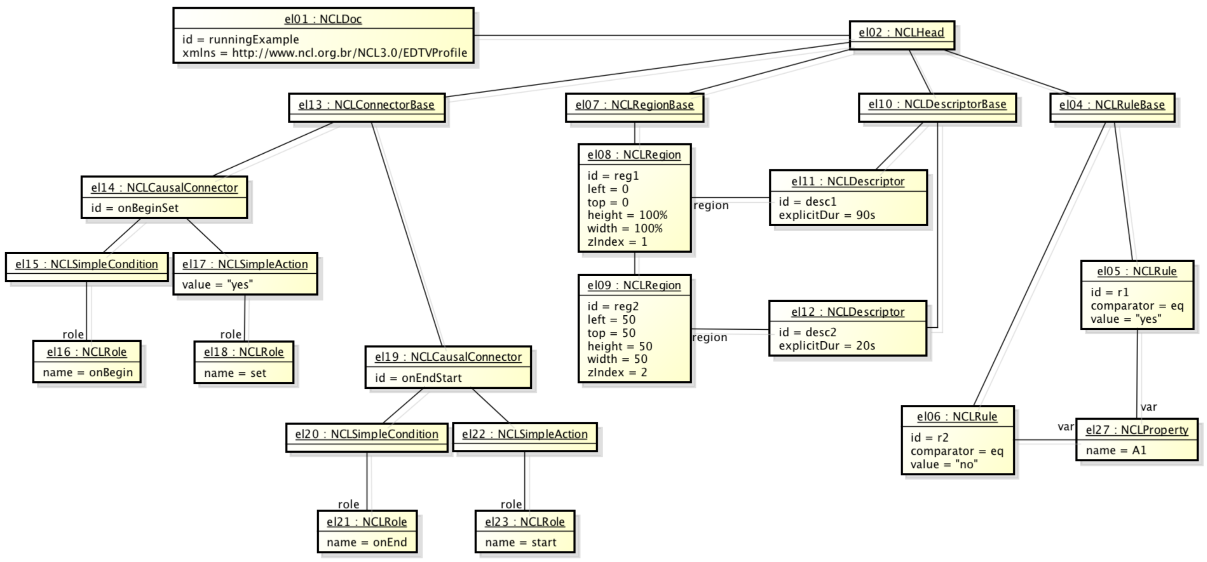


Figure 6.4: Sample document head representation as an object diagram

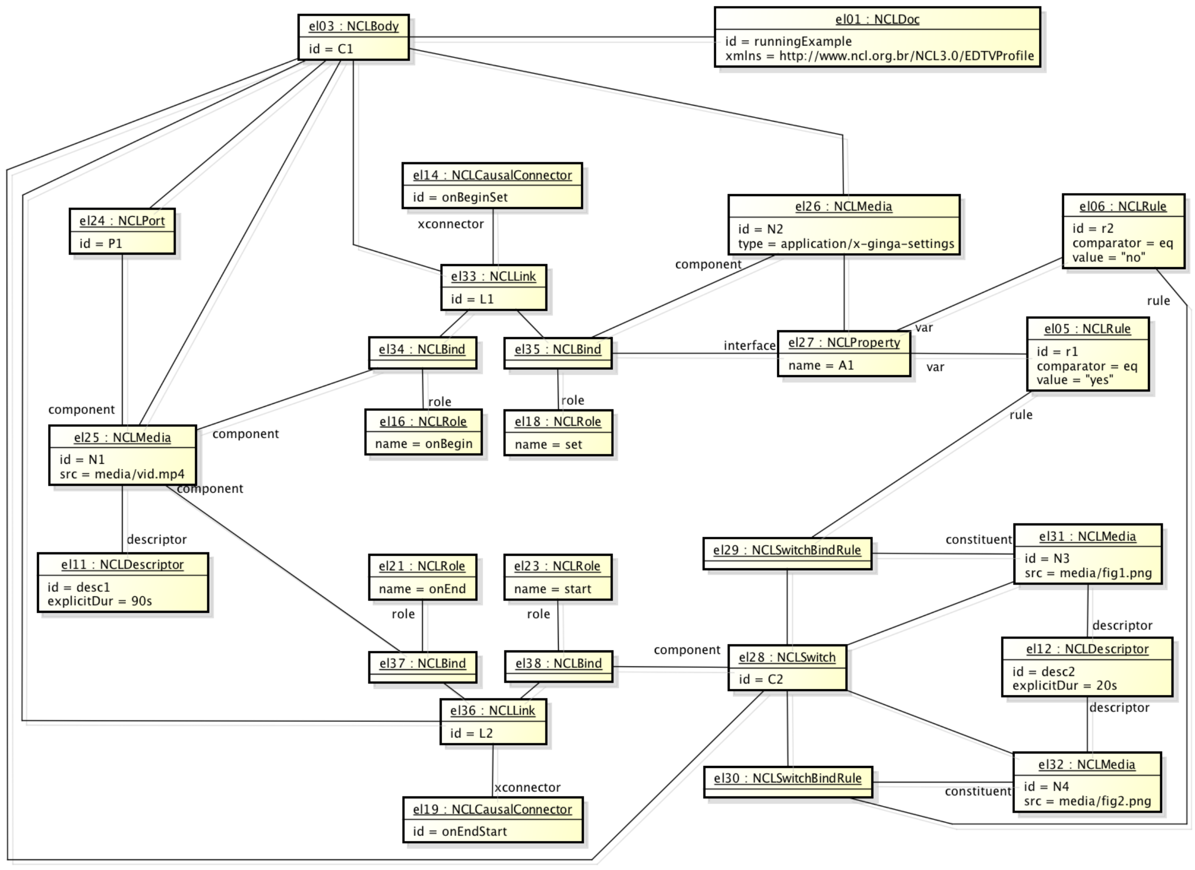


Figure 6.5: Sample document body representation as an object diagram

chy are shown without roles, while associations representing element references show roles representing the element attributes that make references.

The sample document presented satisfies all static validation properties. Taking as example the invariants presented in Section 6.2, the following paragraphs give a brief explanation about the invariants validation.

The invariants shown in Listing 6.2 hold since both links *L1* and *L2* have two or more bind elements and their "xconnector" attributes are not empty.

The invariant shown in Listing 6.3 holds since both *NCLSimpleAction* elements (objects `el17` and `el22`) neither defines the "repeat" nor the "repeatDelay" attributes.

The invariants shown in Listing 6.4 also hold. The first invariant tests if the property "name" attribute is unique inside a media. Property *A1* satisfies it. The second invariant tests if roles inside a connector have different names. This invariant holds since, for both connectors *onBeginSet* and *onEndStart*, their roles are different, *onBegin* and *set* for the first connector and *onEnd* and *start* for the second one. Finally, the third invariant tests if all elements that define an "id" attribute have an unique id. It is possible to observe that the "id" attribute of objects `el01`, `el03`, `el05`, `el06`, `el08`, `el09`, `el11`, `el12`, `el14`, `el19`, `el24`, `el25`, `el26`, `el27`, `el28`, `el31`, `el32`, `el33` and `el36` are different.

The invariant shown in Listing 6.5 holds since for both links, the *NCLBind* elements (objects `el34` and `el35` for link *L1* and `el37` and `el38` for link *L2*) refer to roles defined by the connector referred by the link "xconnector" attribute.

The invariant shown in Listing 6.6 holds since port *P1* refers to element *N1* and both are direct children of the document body.

The invariants shown in Listing 6.7 hold since switch *C2* neither nests itself nor refers to itself creating a nesting loop or a refer loop, respectively.

Suppose we do the following changes to the sample document:

- Object `el24` "id" attribute changes to *r1* and the component association is changed to object `el31`;
- Object `el35` changes its association representing its parent link from link *L1* to link *L2*.

Figure 6.6 presents the sample document body with those changes. Changed objects are highlighted in red.

Regarding the changed document, the first invariant in Listing 6.2 does not hold since link *L1* does not have two or more bind elements. The third invariant in Listing 6.4 does not hold anymore since the "id" attribute of objects `el05` and `el24` are not different.

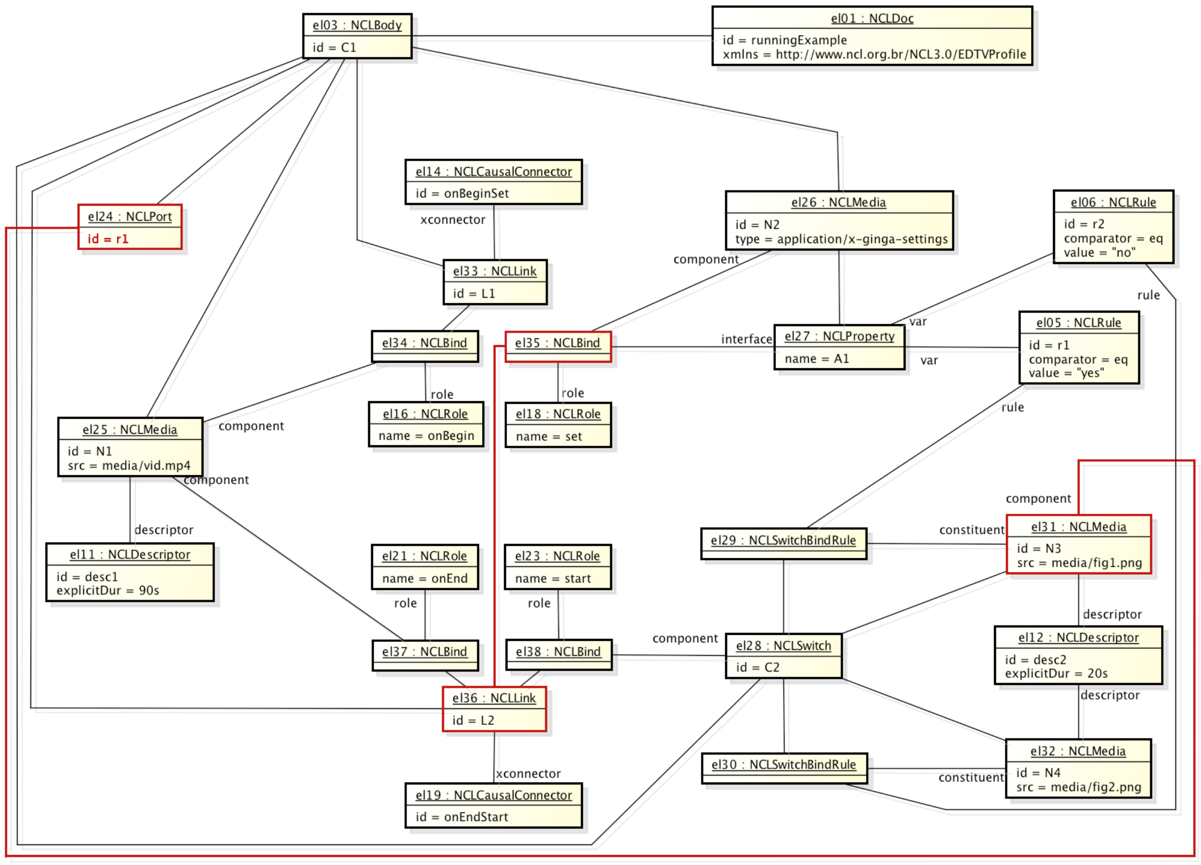


Figure 6.6: Sample document body representation with error

The invariant shown in Listing 6.5 does not hold since, for link *L2*, the *NCLBind* element represented by object `el35` refers to a role not defined by connector *onEndStart*. The invariant shown in Listing 6.6 does not hold since port *P1* (changed to *r1*) refers to element *N3* that is not a direct children of the document body.

6.4 Closing remarks

This chapter presented the metamodel that represents the NCL language syntax. It also presented the representation of the validation properties as OCL invariants and the representation of the document to be validated as an object diagram.

The next chapter proposes SHM (Simple Hypermedia Model), which is used to represent a document behavioral definition, and the representation of the set of dynamic verification properties as a set of LTL formulas that can be verified over the transition system induced by SHM.

API aNaa implements the validation here presented as will be seen in Chapter 8.

Chapter 7

Simple Hypermedia Model dynamic semantics

Several multimedia document models [Buchanan and Zellweger 2005, W3C 2008b, Na and Furuta 2001, Soares et al. 2000, Boll 2001] that present different approaches for specifying a temporal scenario can be found in the literature. From a formal point of view, a temporal scenario represents real time constraints among multimedia document components. Although these models provide different primitives for specifying a temporal scenario, they follow common concepts to multimedia systems. In this chapter, we formally define a generic multimedia model expressive enough to represent the properties we have identified in Chapter 5. It presents a reduced set of entities, giving a simplified representation of a multimedia document. The model proposed here models multimedia documents, whose behavior is not time dependent.

Section 7.1 highlights basic concepts of multimedia systems recalling important definitions presented in Chapter 3. Section 7.2 describes the proposed generic multimedia model named Simple Hypermedia Model (SHM). Section 7.3 discusses the representation of a document as a transition system. Section 7.4 presents how the dynamic verification properties are expressed as LTL [Pnueli 1977] formulas to be verified over a transition system to analyze a multimedia document and Section 7.5 presents some closing remarks.

7.1 Basic concepts

This section recalls important concepts presented on Chapter 3. It presents common characteristics of multimedia document models. Those concepts are based on the NCM model [Soares et al. 2000], presented in Section 3.3, which is event-based and expressive enough to represent those common characteristics.

A multimedia document describes a temporal scenario in terms of *nodes* and *links*. *Nodes* represent media objects and *links* relate nodes, defining an order for their presentation, which is based on event occurrences, like user interaction or other events such as node presentation and attribute value changes.

A node can be a *content node* or a *composite node*. A content node represents a media object, while a composite node represents a set of nodes. Some multimedia authoring models support composite nodes, but not all of them.

A content node has a set of interface points called anchors, which can be *content anchors* or *attribute anchors*. Content anchors represent a subpart of the node content, possibly the whole node content. Attribute anchors represent a node attribute and its value.

Composite nodes can be *context nodes* or *switch nodes*. A context node is used to logically structure the multimedia document. It contains a set of component nodes and a set of links. It has a set of interface points, which can be attribute anchors or ports. A port represents a mapping to a component node or a component node interface point. A switch node contains a set of nodes and also has ports, mapping to a component node or a component node interface point. A switch node, however, defines binds relating a condition to a component node. That way only the component node whose associated condition is evaluated as true is presented when the switch node is presented.

Definition 1 *A node defines* interface points*. Content node interface points are called* anchors*. Composite node interface points can be* ports *and/or* attribute anchors*.*

The presentation of content nodes follows the presentation characteristics defined in the document, like screen position, sound level, etc. Those characteristics are defined by *descriptors*. Descriptors are defined as in NCM and the association between a node and a descriptor follows the conceptual model (see Figure 3.4).

A multimedia link may relate several source and target nodes. It defines a condition related to each source node and how conditions are combined to create a single link activation condition. A condition may represent an event state transition, a test over a node, or anchor, state or a test over an attribute value. Regarding target nodes, a multimedia link defines what will happen with them once the link is activated.

A multimedia event has a state, so its behavior follows a state machine. This behavior follows the generic event state machine defined by NCM, presented in Figure 7.1.

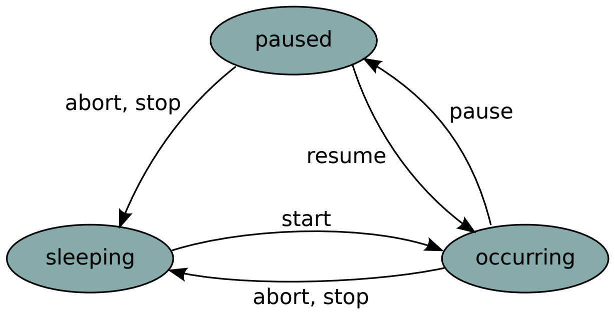


Figure 7.1: NCM event state machine

Definition 2 *The state of an event, or event state, is the current state of an event in its state machine. The possible event states are:* sleeping*,* occurring *and* paused*.*

In the cases where the multimedia authoring model, used by the document to be verified, defines composite nodes, it defines a *node perspective* giving the composition nesting. Since composite nodes can be nested in any depth, a node perspective is the path from the outermost composite node to any node in that path, as defined by NCM semantics.

7.2 Simple Hypermedia Model

The generic multimedia model proposed in this work is based on multimedia events [Pérez-Luque and Little 1996, Blakowski and Steinmetz 1996]. A multimedia event is an occurrence in time with a duration, a state and a type. For example, a user selection is represented as a selection-type event, a media object presentation is represented as a presentation-type event and a attribute value change is represented as an attribution-type event. Other types of events may be defined depending on the multimedia authoring language.

Our model, called Simple Hypermedia Model (SHM) is based on the NCM model [Soares and Rodrigues 2005], supporting the concepts presented in Section 7.1 and is used for the verification of the properties defined in Section 5.2. It represents a multimedia document formalizing the concepts of anchors, links and events. SHM supports three types of events: presentation, selection and attribution.

A multimedia document, using the SHM formalization, is represented as a set of anchors (A) and links (L) and has a set of actions (Ia) that are executed as the document begins.

$$D = (A, L, Ia)$$

Once a document is represented as a set of anchors, the document state is given by the state of all document anchors.

An anchor may be an attribute anchor or a content anchor. An attribute anchor has an identification, a value and the state of the attribution event related to it. A content anchor has an identification and the states of the presentation and selection events related to it.

$$\begin{aligned} A &= Ac \cup Aa \\ Aa &\subseteq Aid \times value \times Es \\ Ac &\subseteq Aid \times Es \times Es \end{aligned} \tag{7.1}$$

In Equations 7.1, Aa represents the set of attribute anchors and Ac the set of content anchors. Aid represents the set of anchor identifications and Es the set of multimedia event states. The anchor identification is formed by the anchor "id" attribute (in the multimedia document), its parent node perspective or the parent node "id" attribute and its parent node descriptor, if defined.

Following the state machine presented in Figure 7.1, the set of multimedia event states Es and the set of multimedia event transitions Et is defined as follows.

$$Es = \{occurring, paused, sleeping\} \tag{7.2}$$

$$Et = \{abort, pause, resume, start, stop, \varepsilon\} \tag{7.3}$$

A multimedia event transition, therefore, is given by Equation (7.4).

$$\rightarrow_{transition} \subseteq Es \times Et \times Es \tag{7.4}$$

SHM multimedia event transitions are presented as follows. Notice that transition ε represents the maintenance of the event state. It will be used when modeling the document state change, which will be presented later in this section.

$$\begin{array}{rcl}
e & \xrightarrow{\varepsilon}_{transition} & e \\
occurring & \xrightarrow{pause}_{transition} & paused \\
occurring & \xrightarrow{stop}_{transition} & sleeping \\
occurring & \xrightarrow{abort}_{transition} & sleeping \\
paused & \xrightarrow{resume}_{transition} & occurring \\
paused & \xrightarrow{stop}_{transition} & sleeping \\
paused & \xrightarrow{abort}_{transition} & sleeping \\
sleeping & \xrightarrow{start}_{transition} & occurring, \\
e & \xrightarrow{et}_{transition} & e \quad , e \in Es, et \in Et \quad otherwise
\end{array}$$

Once the document execution begins, a set of initial actions are executed, changing the state of anchors of the document nodes. An initial action may have associated conditions, so the actions whose conditions hold are executed.

$$Ia = A \times Et \times Bool$$

The general form of elements of Ia is described by (a_n, et_n) if $P_j(a_l) \wedge \ldots \wedge P_k(a_r)$ where $a_i \in A$, $et_i \in Et$ and $P(a_i)$ represent a condition, defining a test over an anchor and returning a boolean value.

A link has conditions and actions. Conditions must be satisfied in order to activate the link and actions are executed as the link is active, modifying the document state. A link condition is triggered by multimedia event transitions related to its source anchors and may define tests over anchor states or values. A link action defines a multimedia event transition that will be induced over a multimedia event state related to its target anchor.

$$L = Set\{A, Et\} \times Set\{A, Et\} \times Bool \tag{7.5}$$

In Equation 7.5, (A, Et) represents the test of the occurrence of a multimedia event transition in the source anchor event state. In the second projection, (A, Et) represents a multimedia event transition in the target anchor event state. Tests over anchor states or values are represented as the link's predicate ($Bool$ in third projection).

Definition 3 *A link is referred as* enabled *over a set of anchors A when its first projection is equal to A. That is, the link conditions are satisfied.*

The general form of elements of L is described by $\{(a_1, et_1), \ldots, (a_k, et_k)\} \rightarrow \{(a_l, et_l), \ldots, (a_n, et_n)\}$ *if* $P_m(a_l) \wedge \ldots \wedge P_s(a_r)$ where $a_i \in A$ and $et_i \in Et$.

Taking as example link L_1, in the sample multimedia document presented in Listings 3.8 and 3.9, its SHM representation is given as follows.

$$(< a_{N_1}, sleeping, e_1 >, start) \rightarrow (< a_{A_1}, \text{``}yes'', sleeping >, start)$$

The $\rightarrow_{transition}$ relation presented in Equation 7.4 represents a transition in the multimedia state machine presented in Figure 7.1. In a link, that transition is applied to an event related to an anchor. A transition applied to an event related to an anchor is called an event occurrence.

Definition 4 *An* event occurrence *is the occurrence of a transition in the state of a multimedia event related to an anchor.*

An event occurrence is formalized by Equations (7.6) and (7.7). In the equations, index P represents presentation, S selection and A attribution related event states or transitions.

$$\frac{e_P \xrightarrow{et_P}_{transition} e'_P \quad e_S \xrightarrow{et_S}_{transition} e'_S}{< a_{id}, e_P, e_S > \xrightarrow{et_P, et_S}_{event} < a_{id}, e'_P, e'_S >}, \quad \begin{array}{c} et_P, et_S \in Et \\ e_P, e'_P, e_S, e'_S \in Es \\ a_{id} \in Aid \end{array} \tag{7.6}$$

$$\frac{e_A \xrightarrow{et_A}_{transition} e'_A}{< a_{id}, value, e_A > \xrightarrow{et_A}_{event} < a_{id}, value, e'_A >}, \quad \begin{array}{c} et_A \in Et \\ e_A, e'_A \in Es \\ a_{id} \in Aid \end{array} \tag{7.7}$$

An event occurrence is considered natural when it occurs independently of the links specified in the multimedia document. Examples of natural event occurrences are user interaction and the anchor presentation end when its duration is reached. A natural event occurrence follows the same formalization as $\rightarrow_{event}$, where $t_P \in \{stop, \varepsilon\}$, $t_S \in Et$ and $t_A \in \{stop, \varepsilon\}$.

Definition 5 *A* natural event occurrence *is an event occurrence that happens independently of the links specified in the multimedia document. A* natural event occurrence *is represented by* $\xrightarrow{et_P, et_S, et_A}_{natural}$ *where* $et_P \in \{stop, \varepsilon\}$*,* $et_S \in Et$ *and* $et_A \in \{stop, \varepsilon\}$*.*

A relation of type $\rightarrow_{natural}$ represents a modification of an anchor state, which is given by the occurrence of a multimedia event transition in the anchor event states.

Once a natural event occurrence happens, all links enabled by that event occurrence are triggered, changing the state of document anchors, possibly the same source anchor. That behavior is formalized by Equation (7.8).

$$\frac{\begin{array}{cc} et_P \neq \varepsilon \vee et_S \neq \varepsilon \vee et_A \neq \varepsilon & S'_1 \subseteq S_1,\, a_{l_x} \in S'_1,\, a_{r_y} \in S_1 \cup S_2 \\ a_i \in S_1 & m \leq |S'_1|,\, k \leq |S_1 \cup S_2| \\ & 1 \leq x \leq m,\, 1 \leq y \leq k \\ & \\ a_i \xrightarrow{et_P, et_S, et_A}_{natural} a'_i & \{(a_{l_1}, et_{l_1}), \ldots, (a_{l_m}, et_{l_m})\} \rightarrow^*_L \\ & \{(a_{r_1}, et_{r_1}), \ldots, (a_{r_k}, et_{r_k})\}\; if\, et_{l_1} \neq \varepsilon \wedge \ldots \wedge et_{l_m} \neq \varepsilon \end{array}}{S_1 \cup S_2 \quad \rightarrow_{SHM} \quad (\{a'_1, \ldots, a'_n\} \cup S_2)/\{a_{r_1}, \ldots, a_{r_k}\}} \tag{7.8}$$

where / is the overriding operation on anchor sets defined as follows.

$$\begin{array}{rcl} \emptyset/S_1 & = & S_1 \\ (\{a_i\} \cup S_1)/S_2 & = & \{a_i\} \cup (S_1/S_2)\, if a_i \notin S_2 \\ (\{a_i\} \cup S_1)/(\{a'_i\} \cup S_2) & = & \{a'_i\} \cup (S_1/S_2) \end{array} \tag{7.9}$$

On the formalization above, S_1 represents the anchors that will suffer a natural event occurrence and S_2 represents the remaining anchors of the document. The anchors $a_i \in S_1$ will have at least one of its multimedia event state changed, since at least one of the multimedia event transitions et_P, et_S and et_A should not be equal to ε.

Once the natural event occurrence happens, links enabled over a subset of the anchors that suffered the natural event occurrence (S'_1) will be triggered, changing the state of one or more anchors in the document. It is worth to notice that once a link changes the state of document anchors, it may enable other document links, which will be triggered. The document link evaluation continues until no other document link is enabled.

After a $\rightarrow_{SHM}$ relation, the document state will be given by the state of the document anchors after the natural event occurrence, taking into consideration the anchors affected by document links.

Regarding the SHM formalization, the sample multimedia document presented in Listings 3.8 and 3.9 is described by Equation 7.10 where $e_1, e_2 \in Es$.

$$\begin{aligned}
doc &= (A_{doc}, L_{doc}) \\
A_{doc} &= \{a_{N_1}, a_{A_1}, a_{N_3}, a_{N_4}\} \\
L_{doc} &= \{((< a_{N_1}, sleeping, e_1 >, start) \rightarrow (< a_{A_1}, “yes'', sleeping >, start)), \\
&\quad ((< a_{N_1}, occurring, e_1 >, stop) \rightarrow (< a_{N_3}, sleeping, e_2 >, start)\ \mathit{if}\ P_{r_1}(a_{A_1})), \\
&\quad ((< a_{N_1}, occurring, e_1 >, stop) \rightarrow (< a_{N_4}, sleeping, e_2 >, start)\ \mathit{if}\ P_{r_2}(a_{A_1}))\}
\end{aligned} \tag{7.10}$$

The anchor identifications are described by Equation 7.11.

$$\begin{aligned}
a_{N_1} &= N_1 : C_1, N_1 : desc1 \\
a_{A_1} &= A_1 : C_1, N_2 \\
a_{N_3} &= N_3 : C_1, C_2, N_3 : desc2 \\
a_{N_4} &= N_4 : C_1, C_2, N_4 : desc2
\end{aligned} \tag{7.11}$$

The link conditions are described by Equation 7.12.

$$\begin{aligned}
P_{r_1}(< a_{A_1}, v, e_1 >) &= true \qquad \mathit{if}\ v = “yes'' \\
P_{r_2}(< a_{A_1}, v, e_1 >) &= true \qquad \mathit{if}\ v = “no''
\end{aligned} \tag{7.12}$$

Notice, from equation 7.11 that the *all content anchor* of nodes N_1, N_3 and N_4 have the same id as the parent node. Also, from Equation 7.12, link L_2 was converted to two links having as target each one of the switch C_2 component anchors and the switch rules were represented as link additional conditions.

7.3 SHM transition system

The behavior of an SHM document is given by the transition system associated to it. This section presents that representation of an SHM document as a transition system. The configuration of the transition system $\mathcal{M}_{SHM}$ associated to an SHM document is comprised by a set of anchors $Set\{A\}$.

$$\begin{aligned}
S_{SHM} &= Set\{A\} \\
\rightarrow_{SHM} &\subseteq S_{SHM} \times S_{SHM}
\end{aligned}$$

Every state of the transition system (S_{SHM}) represents the state of the whole multimedia document, which is given by the state of the anchors that compose the document.

Every transition represents a modification in that document state. A transition in the document state is given by $\rightarrow_{SHM}$ relations.

Regarding the SHM formalization of the sample multimedia document presented in Equation 7.10, the transition system induced by the document is presented in Figure 7.2. In the figure, rectangles represent the transition system states (s_0 until s_4). Dark rectangles s_0 and s_4 represent the initial and final state of the transition system, respectively.

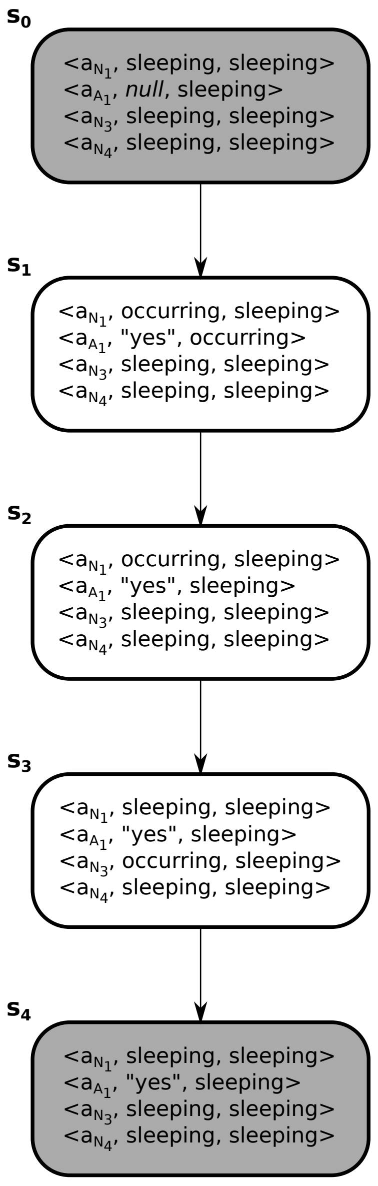

s_0
<a_{N_1}, sleeping, sleeping>
<a_{A_1}, *null*, sleeping>
<a_{N_3}, sleeping, sleeping>
<a_{N_4}, sleeping, sleeping>
s_1
<a_{N_1}, occurring, sleeping>
<a_{A_1}, "yes", occurring>
<a_{N_3}, sleeping, sleeping>
<a_{N_4}, sleeping, sleeping>
s_2
<a_{N_1}, occurring, sleeping>
<a_{A_1}, "yes", sleeping>
<a_{N_3}, sleeping, sleeping>
<a_{N_4}, sleeping, sleeping>
s_3
<a_{N_1}, sleeping, sleeping>
<a_{A_1}, "yes", sleeping>
<a_{N_3}, occurring, sleeping>
<a_{N_4}, sleeping, sleeping>
s_4
<a_{N_1}, sleeping, sleeping>
<a_{A_1}, "yes", sleeping>
<a_{N_3}, sleeping, sleeping>
<a_{N_4}, sleeping, sleeping>

Figure 7.2: Sample document induced transition system

State s_0 represents the document initial state. At that moment all document anchors have their associated event states equal to "sleeping". Attribute anchor a_{A_1} does not have a value at this moment. When the document execution begins anchor a_{N_1} starts its presentation and, as defined by link L_1, anchor a_{A_1} have its value set to "yes" (state s_1).

After anchor a_{A_1} value is set, its attribution event state changes to “sleeping” (state s_2). Once the anchor a_{N_1} duration is reached, it stops its presentation and, as defined by link L_2, anchor a_{N_3} presentation starts (state s_3). Once the anchor a_{N_3} duration is reached, it stops its presentation and the document execution ends (state s_4).

The following section presents how the dynamic properties defined in Chapter 5 are represented as LTL formulas and how they are applied over the sample document induced transition system.

7.4 Verification of SHM properties

The dynamic verification is done over the generic multimedia model presented in Section 7.2 (SHM). The verification is achieved by transforming SHM into a transition system. The set of dynamic verification properties is represented as a set of LTL formulas that can be verified over the transition system. The automatic investigation of transition system properties is done by a model checker [Clarke et al. 2000].

Let $a_1 \in A$ be the anchor that we want to reason about. The reachability verification property is represented as an LTL formula that investigates if there is a path π and a state $s \in \pi$, such that, a_1 in s presents the state of at least one event related to it equals to *occurring*. Then it is possible to determine if an anchor event will be executed during the document execution. An anchor event is considered to be executed, when its state changes to *occurring*. The LTL formula representing the reachability property is presented by Equation (7.13).

$$F\,\text{pre-occurring}(A_{id}) \;\vee\; F\,\text{att-occurring}(A_{id}) \tag{7.13}$$

In Equations 7.13, 7.14, 7.15 and 7.16, F and G represents the *future* and *global* operators, respectively (see Section 4.3.1). The functions that test the anchor event state are formed by a prefix representing the event type (presentation, attribution and selection) and the state to be tested. For example, function pre-occurring(A_{id}) tests if the *presentation* state of anchor A_{id} is equal to *occurring*.

The anchor termination verification property is represented as an LTL formula that investigates if there is a path π and states $s_i, s_j \in \pi, i < j$, such that, a_1 in s_i has an event state equal to *occurring* and in s_j has the state of the same event equal to *sleeping*. In this way it is possible to determine that once an anchor event is executed, it will stop sometime afterwards. The LTL formula representing the anchor termination property is presented by Equation (7.14).

$$\begin{aligned}(GF\,\text{pre-occurring}(A_{id}) &\rightarrow GF\,\text{pre-sleeping}(A_{id})) \wedge \\ (GF\,\text{att-occurring}(A_{id}) &\rightarrow GF\,\text{att-sleeping}(A_{id})) \wedge \\ (GF\,\text{sel-occurring}(A_{id}) &\rightarrow GF\,\text{sel-sleeping}(A_{id}))\end{aligned} \tag{7.14}$$

The document termination verification property is represented as an LTL formula that investigates if there is a path π and a state $s \in \pi$, such that, $\forall\ a_i \in A$, the state of the events in a_i is *sleeping* and remains that way for every state in π after s. The LTL formula representing the document termination property is given by Equation (7.15), where function doc-sleeping tests if the document reached its end.

$$GF\,\text{doc-sleeping} \tag{7.15}$$

It is worth noticing that a multimedia document will present a loop without an exit condition, for example a user interaction, if the final state, where every anchor has its state equal to *sleeping*, is unreachable. Since the termination verification is undecidable, an ending condition for $\mathcal{M}$ with a document maximum duration may be defined.

Finally the resource verification property is represented as an LTL formula that investigates if there is a path π and a state $s \in \pi$, such that, a_i and a_j in s have the state of events related to a_i and a_j equal to *occurring*. Where a_i and a_j represent the anchors that use a same resource, for example, a same screen space at a specific presentation device. The LTL formula representing the resource property is presented by Equation (7.16).

$$G\,(\neg\,\text{pre-occurring}(A_{id_1})\ \wedge\ \neg\,\text{pre-occurring}(A_{id_2})) \tag{7.16}$$

Regarding the sample multimedia document induced transition system presented in Figure 7.2, the following paragraphs explain the application of the LTL formulas previously presented.

The transition system example of Figure 7.2 defines a path $\pi = \{s_0, s_1, s_2, s_3, s_4\}$ over which the properties are verified.

The reachability verification property holds for anchors a_{N_1}, a_{A_1} and a_{N_3}, since there is a state for a_{N_1} (s_1), a_{A_1} (s_1) and a_{N_3} (s_3), where the anchor presentation state (for a_{N_1} and a_{N_3}) and attribution state (for a_{A_1}) is equal to *occurring*. The property does not hold for anchor a_{N_4}, since there is no state where the anchor presentation state is equal to *occurring*.

The anchor termination property holds for all anchors, since in the states s_2, s_3 and s_4, a_{A_1}, a_{N_1} and a_{N_3} have their event states equal to *sleeping*, respectively. Since anchor a_{N_4} does not change its state in path π, the property also holds.

The document termination property holds, since in s_4 every anchor has its event state equal to *sleeping* and remains that way.

The resource property also holds since both anchors that use a same resource (a_{N_3} and a_{N_4}) are not executed at the same time.

7.5 Closing remarks

This chapter presented the SHM model used for representing the verification properties presented in Chapter 5. That model is based in common characteristics of multimedia document models and represents a multimedia document as anchors, events and links. SHM definition induces a transition system. The set of dynamic verification properties is represented as a set of LTL formulas that can be verified over the transition system. The automatic investigation of transition system properties is done by a model checker.

Although SHM was based on the NCM model, some characteristics of NCM were not comprised in this version of SHM, such as the representation of link delays and the complete representation of an event, representing the occurrences and repetitions concepts [Soares and Rodrigues 2005]. Those characteristics are important and will be addressed as future work.

The next chapter presents the implementation of the verification here presented in the API aNaa.

Chapter 8

API for NCL Authoring and Analysis

This chapter presents aNaa (API for NCL Authoring and Analysis), the test case tool that implements the analysis proposed in this work for multimedia documents defined by the NCM model [Soares et al. 2000], using the NCL language [ITU 2009]. aNaa can be used as an analysis tool, receiving NCL documents as input and returning error messages, or as the basis for developing NCL authoring tools with analysis capabilities.

To achieve the analysis here presented, aNaa uses some supporting tools. Those tools are used in the validation of NCL document structure and the verification of NCL document behavior. Those supporting tools will be presented in Section 8.1.

aNaa uses the aNa API as it basis, which represents NCL 3.0 documents. The aNa API implementation is shown in Section 8.2. Finally the aNaa implementation is presented in Section 8.3. Section 8.4 concludes this chapter with some remarks.

8.1 aNaa supporting tools

aNaa implements both methods presented in this work for an NCL document analysis, which are: the validation of an NCL document structure and the verification of an NCL document behavior. This section presents the supporting tools used to achieve that analysis. Figure 8.1 presents an overview of aNaa, regarding the use of external tools.

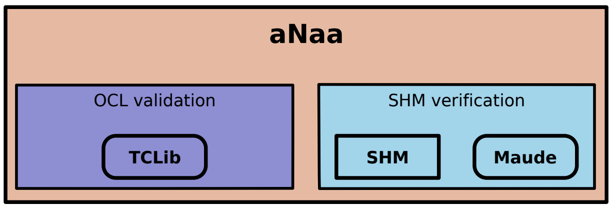


Figure 8.1: aNaa external tools

To validate an NCL document structure, aNaa uses the TCLib tool [Braga et al. 2011]. To validate an NCL document behavior, aNaa uses both the SHM model and the Maude tool. It is worth noticing that the validation and the verification implementations are independent from each other. Further details of the tools will be presented in the following sections. Section 8.3 will present the API architecture in more details.

8.1.1 Supporting tool for OCL validation

aNaa validates NCL document structure. This validation is achieved using the TCLib tool [Braga et al. 2011]. This tool receives a metamodel and a set of invariants defining properties that every instance of that metamodel should satisfy. When the tool receives an instance of that metamodel, it investigates, for that instance, if the invariants hold. TCLib represents the metamodel and its instance as class and object diagrams respectively. Invariants are described with OCL (Object Constraint Language). Figure 8.2 presents an overview of TCLib.

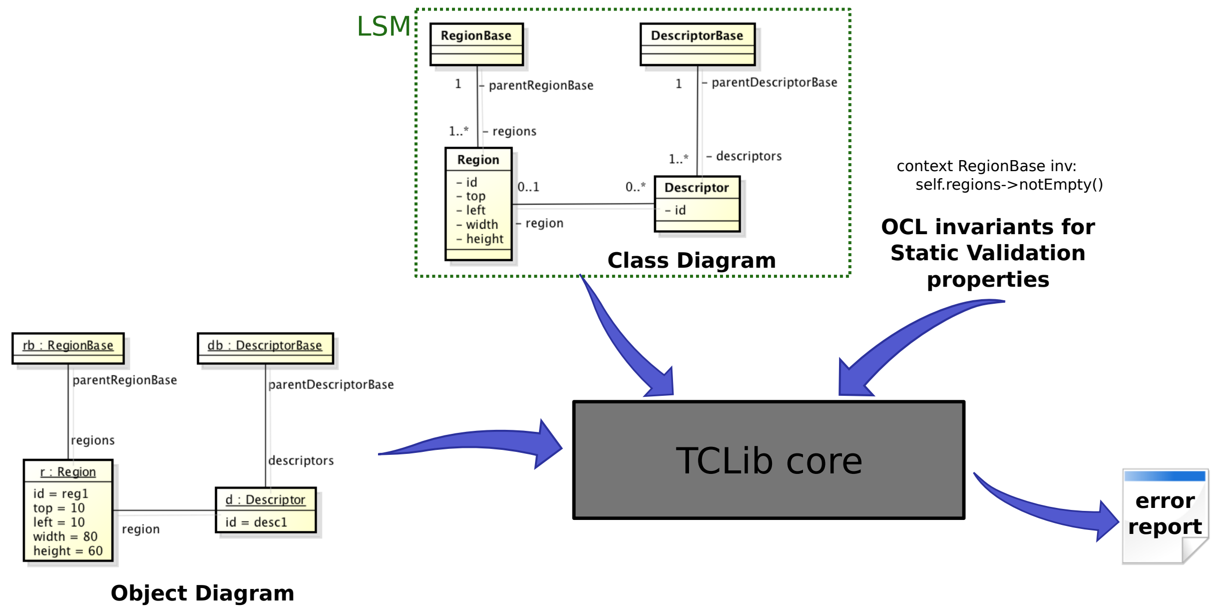


Figure 8.2: TCLib tool overview

TCLib provides operations to create a class diagram, that is, create classes, attributes, associations, generalizations, etc; to create an object diagram, that is, create objects, object attribute values, object associations, etc; to create a set of desired properties to be validated and the validation of those properties. TCLib is implemented based on a Transformation Contract project pattern [Braga et al. 2011]. This pattern is used in Model-driven Development to guarantee the consistency of a model before and after its transformation among different domains. During the transformation, consistency checking is done using OCL invariants.

Although TCLib was designed for consistency checking in a model transformation, it architecture allows its use just for model validation. aNaa only uses the tool validation capabilities.

aNaa creates inside TCLib the NCL Language Structure Metamodel (LSM), presented in Section 6.1, and OCL invariants that represent the static validation properties, presented in Section 5.1. It also represents the document being authored as an object diagram, which is used as input for TCLib to test the invariants. Once one invariant fails, TCLib returns the invariant that was not respected and the object that did not respect it. That way aNaa can return to the author error messages that clearly indicate the NCL element and the specification problem related to it.

8.1.2 Supporting tool for SHM verification

aNaa verifies the behavior of NCL documents. The verification is done over the SHM model as presented in Chapter 7. An NCL document is represented as an SHM document. The automatic investigation of transition system properties is done by a model checker [Clarke et al. 2000]. The properties to be verified, as well as the SHM representation of the multimedia document to be verified and the investigation of the temporal properties uses Maude [Clavel et al. 2007].

As presented in Chapter 7, an SHM document has an associated transition system TS_{SHM}. Maude is used to represent an SHM document as a rewrite theory $\mathcal{R}_{SHM}$ and, therefore, as a transition system induced by $\mathcal{R}_{SHM}$. This section presents the Maude representation of SHM documents.

SHM represents a document D in terms of anchors, links and initial actions $D = (A, L, Ia)$.

Maude represents the sort of anchors A as the *anchor* sort. An element of sort *anchor* is formed by the anchor identification - which is the anchor *id*, *perspective* and *descriptor* - and a set of attributes that represent information related to it. Listing 8.1 presents the representation of anchor identification (sort $AnchorIdentInfo$), Listing 8.2 presents the definition of an attribute set and Listing 8.3 presents the representation of an anchor.

Listing 8.1: Maude anchor identification representation

```
sort AnchorIdentInfo .
sorts AnchorId AnchorPerspective AnchorDescriptor .
```

```

subsort String < AnchorId AnchorPerspective AnchorDescriptor .

op (_:_) : AnchorId AnchorPerspective -> AnchorIdentInfo .
op (_:_:_) : AnchorId AnchorPerspective AnchorDescriptor -> AnchorIdentInfo .
```

Listing 8.2: Maude attribute set

```
sorts Attribute AttributeSet .

subsort Attribute < AttributeSet .

op none : -> AttributeSet [ctor] .
op _,_ : AttributeSet AttributeSet -> AttributeSet [ctor assoc comm id: none] .
```

Listing 8.3: Maude anchor representation

```
sort Anchor .

ops defContent defAttribute : -> AttributeSet .

op <_|_> : AnchorIdentInfo AttributeSet -> Anchor [ctor] .
```

Operations *defContent* and *defAttribute* in Listing 8.3 represent a syntactic sugar for the construction of content anchors and attribute anchors, respectively. Listing 8.4 presents the anchor attributes defined by Maude and the *defContent* and *defAttribute* values. The list is divided by the anchor type.

Listing 8.4: Maude anchor attribute list

```
*** attribute anchor attributes
op att-value =_ : EventValue -> Attribute [ctor] .
op att-state =_ : EventState -> Attribute [ctor] .

*** content anchor attributes
op pre-state =_ : EventState -> Attribute [ctor] .
op pre-duration =_ : EventDuration -> Attribute [ctor] .
op sel-state =_ : EventState -> Attribute [ctor] .
op sel-pressedKey =_ : EventKey -> Attribute [ctor] .

*** default attribute values
eq defContent = pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
    sel-pressedKey = noKey .
eq defAttribute = att-value = "noVal", att-state = sleeping .
```

An anchor has the state of the events associated to it (see Equation 7.1). The sort *EventState* represents the set of multimedia event states Es in Maude. Listing 8.5 presents the definition of sort *EventState*.

Listing 8.5: Maude event state representation

```
sorts StandingState TransientState EventState .
sorts InitStandingState EndStandingState InitTransientState EndTransientState
    InitEventState EndEventState .
sort EventType EventDuration EventValue EventKey .

subsort InitStandingState EndStandingState < StandingState .
subsort InitTransientState EndTransientState < TransientState .
subsort StandingState TransientState < EventState .
subsort InitStandingState InitTransientState < InitEventState .
subsort EndStandingState EndTransientState < EndEventState .
subsort Nat < EventDuration .
subsort String < EventValue .

ops presentation selection attribution : -> EventType [ctor] .

op sleeping : -> EndStandingState [ctor] .
ops occurring paused : -> InitStandingState [ctor] .
ops stopping aborting : -> EndTransientState [ctor] .
ops starting pausing resuming : -> InitTransientState [ctor] .

ops noKey RED GREEN BLUE YELLOW OK : -> EventKey .
```

Since it is necessary to be able to model and reason over the occurrence of event transitions (see Figure 7.1), they were represented as event states in the Maude specification. The set Es of event states was extended with new states representing those transitions. Figure 8.3 presents the extended generic event state machine. SHM event states are the green ones and Maude transient states are the yellow ones.

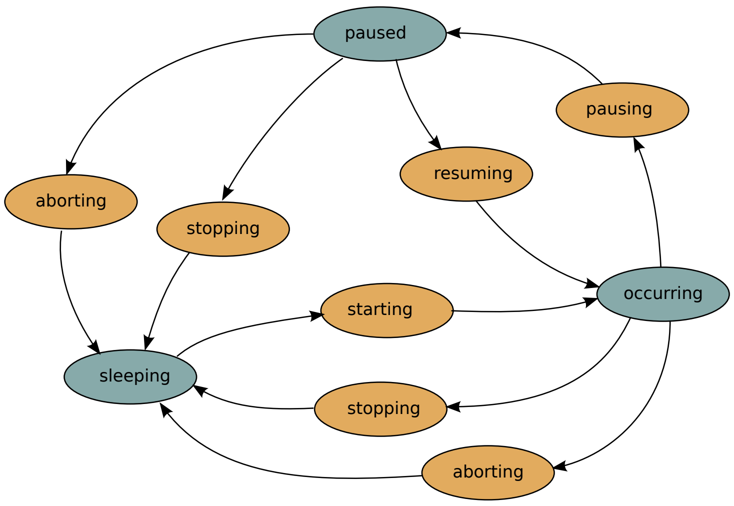


Figure 8.3: Maude generic event state machine

Maude defines sort *EventState* as the sort formed by the union of the event states defined by SHM - sort *StandingState* - and the states that represent transitions - sort *TransientState*. Maude also defines sorts *EventType* and *EventKey* to represent the types of events that may be related to anchors and a set of keys related to user selection, respectively. Notice that sorts *EventState*, *StandingState* and *TransientState* have subsorts to help identifying the state. That characteristic makes easy the definition of equations that change the document state.

SHM transitions model a modification of an event state. Sort *StateTransition* represents the sort of transitions *Et* in Maude. Listing 8.6 presents the definition of sort *StateTransition*.

Listing 8.6: Maude event state transition representation

```
sort StateTransition .

ops start stop pause resume abort : -> StateTransition [ctor] .
```

The event relation (see Equations 7.6 and 7.7) represents the occurrence of a transition in the state of a specific event of an anchor. Maude represents an event occurrence by sort *EventTransition* presented in Listing 8.7.

Listing 8.7: Maude event occurrence representation

```
sort EventTransition .
sort TransitionType .

op __ : StateTransition EventType -> TransitionType [ctor] .
op <_|_|_> : TransitionType AnchorIdentInfo AttributeSet -> EventTransition [ctor] .
op <_|_> : TransitionType AnchorIdentInfo -> EventTransition .

var I : TransitionTarget .
var T : TransitionType .

eq < T | I > = < T | I | none > [owise] .
```

An element of sort *EventTransition* may have attributes. Listing 8.8 presents those possible attributes.

Listing 8.8: Maude transition attributes

```
sort EventValue .
subsort String < EventValue .

op value =_ : EventValue -> Attribute [ctor format (s s b o)] .
op key =_ : EventKey -> Attribute [ctor format (s s b o)] .
```

The *value* attribute represents the value to be set in a property anchor in an attribution event. The *key* attribute represents the key pressed in a user selection. The application of a transition over an anchor is represented by the equations presented in Listing 8.9.

Listing 8.9: Maude transition occurrence equations

```
var D : DocContent .
var Ac : Anchor .
var I : AnchorIdentInfo .
var A : AttributeSet .
var T : EventTransition .
var Ies : InitEventState .
var Ees : EndEventState .
vars Ev Ev' : EventValue .
vars Ek Ek' : EventKey .
var Et : EventDuration .

*** start transitions
eq < start presentation | I | none > < I | pre-state = Ees, A > D = < I | pre-state =
    starting , A > D .
eq < start attribution | I | value = Ev > < I | att-value = Ev', att-state = Ees, A > D
    = < I | att-value = Ev, att-state = starting , A > D .
eq < start selection | I | key = Ek > < I | sel-state = Ees, sel-pressedKey = Ek', A > D
    = < I | sel-state = starting , sel-pressedKey = Ek, A > D .

*** stop transitions
eq < stop presentation | I | none > < I | pre-state = Ies , pre-duration = Et, A > D = <
    I | pre-state = stopping , pre-duration = 0, A > D .
eq < stop attribution | I | none > < I | att-state = Ies , A > D = < I | att-state =
    stopping , A > D .
eq < stop selection | I | none > < I | sel-state = Ies , A > D = < I | sel-state =
    stopping , A > D .

*** abort transitions
eq < abort presentation | I | none > < I | pre-state = Ies , pre-duration = Et, A > D = <
    I | pre-state = aborting , pre-duration = 0, A > D .
eq < abort attribution | I | none > < I | att-state = Ies , A > D = < I | att-state =
    aborting , A > D .
eq < abort selection | I | none > < I | sel-state = Ies , A > D = < I | sel-state =
    aborting , A > D .

*** pause transitions
eq < pause presentation | I | none > < I | pre-state = occurring , A > D = < I |
    pre-state = pausing , A > D .
eq < pause attribution | I | none > < I | att-state = occurring , A > D = < I | att-state
    = pausing , A > D .
eq < pause selection | I | none > < I | sel-state = occurring , A > D = < I | sel-state =
    pausing , A > D .

*** resume transitions
eq < resume presentation | I | none > < I | pre-state = paused , A > D = < I | pre-state
    = resuming , A > D .
eq < resume attribution | I | none > < I | att-state = paused , A > D = < I | att-state =
    resuming , A > D .
eq < resume selection | I | none > < I | sel-state = paused , A > D = < I | sel-state =
    resuming , A > D .

eq T Ac = Ac [owise] .
```

A document, in Maude, is represented by the set of anchors that compose it, which is represented by sort *DocContent*. Listing 8.10 presents the definition of that sort.

Listing 8.10: Maude SHM document representation

```
sorts DocContent .

subsort Anchor EventTransition < DocContent .

op none : -> DocContent [ctor] .
op __ : DocContent DocContent -> DocContent [ctor assoc comm id: none] .
```

SHM relations model the document state modification by natural event occurrences and the application of document links enabled by them. A natural event occurrence is given by the end of an anchor presentation, regarding its duration, the end of an attribution or the occurrence of user selection (see Definition 5 in Section 7.2).

Maude maps SHM transitions to evolution steps, where four steps are necessary to model an SHM transition. They are: (1) the increment of anchors duration, (2) the evaluation of natural event occurrences, (3) the evaluation of links and (4) the evolving of states that represent transitions. The application of those four steps is equivalent to the application of one SHM transition. Those steps application continues until the document ends.

$$\frac{\overset{a_i \in D}{a_i \xrightarrow{increment}_1 a_i'} \quad \overset{a_i' \in D'}{a_i' \xrightarrow{natural}_1 a_i''} \quad \overset{a_i'' \in D''}{a_i'' \xrightarrow{links}_1 a_i'''} \quad \overset{a_i''' \in D'''}{a_i''' \xrightarrow{evolving}_1 a_i''''}}{D \xrightarrow{SHM}_1 D''''} \quad \neg ended(D) \tag{8.1}$$

The test of the document end is presented in Listing 8.11.

Listing 8.11: Document end test

```
op ended : DocContent -> Bool .
eq ended(< I | att-state = sleeping , A > D) = ended(D) .
eq ended(< I | pre-state = sleeping , sel-state = sleeping , A > D) = ended(D) .
eq ended(< I | att-state = sleeping , A > none) = true .
eq ended(< I | pre-state = sleeping , sel-state = sleeping , A > none) = true .
eq ended(none) = true .
eq ended(D) = false [owise] .
```

The system configuration, represented by sort *DocState*, is given by *DocContent* plus a token to indicate the current document evolution step. Listing 8.12 presents the sort *DocState* definition.

Listing 8.12: Maude system configuration

```
sorts DocContent DocState .

subsort EvolvingToken < DocContent .

op _[_] : EvolvingInfo DocContent -> DocState .
```

The *EvolvingInfo* sort definition is presented in Listing 8.13.

Listing 8.13: Maude evolving info

```
sort EvolvingInfo EvolvingToken .
sort EvolvingStep EvolvingTime EvolvingMessage .

subsort Nat < EvolvingTime .

ops formatter implicit explicit : -> EvolvingStep [ctor] .

op END-TIME : -> EvolvingMessage [ctor] .
op DOC-END : -> EvolvingMessage [ctor] .

op [_:_] : EvolvingStep EvolvingTime -> EvolvingInfo [ctor] .
op [_:_|_] : EvolvingStep EvolvingTime EvolvingMessage -> EvolvingInfo [ctor] .
op [_|_] : EvolvingMessage EvolvingTime -> EvolvingInfo [ctor] .
```

Notice that Maude just defines three evolving steps: *formatter*, *implicit* and *explicit*. During the *formatter* step, Maude implements the evolving of states that represent transitions and the increment of anchors duration (steps *evolving* and *increment* in Equation 8.1). During the *implicit* step, Maude evaluates natural event occurrences (step *natural* in Equation 8.1). And during the *explicit* step, Maude evaluates links (step *links* in Equation 8.1). Listing 8.14 presents the change from one step to another.

Listing 8.14: Maude evolving steps

```
var D : DocContent .
var I : AnchorIdentInfo .
var A : AttributeSet .
var S : EvolvingStep .
var T : EvolvingTime .
var ET : EvolvingToken .

--- operation to test if the document is still evolving
op docIsEvolving : DocContent -> Bool .
eq docIsEvolving(ET D) = true .
eq docIsEvolving(D) = false [owise] .

--- start the document simulation
eq run = [ explicit : 0 ] [(InitActions Document) links] .

--- document evolving steps
```

```
crl [ explicit : T ]  [D] => [ formatter : inc(T) ] [formatter-evolve-doc(D)] if
    not(ended(D)) and docWillFormatterEvolve(D) .
ceq [ explicit : T ]  [D] = [ formatter : inc(T) ] [formatter-evolve-doc(D)] if
    not(docIsEvolving(D)) and not(ended(D)) and not(docWillFormatterEvolve(D)) .

crl [ formatter : T ] [D] => [ implicit : T ] [mountEvolvingStep(T, D,
    EvolvingOccurSelection) D] if not(ended(D)) and docWillImplicitEvolve(selectKey(T),
    D) .
ceq [ formatter : T ] [D] = [ implicit : T ] [mountEvolvingStep(T, D,
    EvolvingOccurSelection) D] if not(docIsEvolving(D)) and not(ended(D)) and
    not(docWillImplicitEvolve(selectKey(T), D)) .

crl [ implicit : T ]  [D] => [ explicit : T ]  [links D] if not(ended(D)) and
    docWillExplicitEvolve(D) .
ceq [ implicit : T ]  [D] = [ explicit : T ]  [links D] if not(docIsEvolving(D)) and
    not(ended(D)) and not(docWillExplicitEvolve(D)) .

--- end the document if its ended
crl [ S : T ] [D] => [ DOC-END | T ] [D] if ended(D) .
```

At each step, Maude verifies if the document is still executing. If so, it applies the equations that describe the document behavior inside a step.

After all equations are applied, Maude tests if any modification in the document state will occur in the next step. If the document state will change, a rule is used to change the evolving step, and consequently, creating a new state in the transition system. If the document state will not change, an equation is used, so no new state is created.

The natural event occurrences are evaluated at the same time, by equations that end an anchor presentation if the anchor duration was reached, end an anchor attribution once it is occurring and start an anchor selection once it is sleeping. Listing 8.15 shows the definition regarding the *implicit* step.

Listing 8.15: Maude implicit evolve equations

```
op implicit-evolve-doc : DocContent -> DocContent .
op implicit-evolve-doc : DocContent EventKey -> DocContent .
op implicit-evolve : AnchorIdentInfo -> EvolvingToken .
op implicit-evolve : AnchorIdentInfo EventKey -> EvolvingToken .

*** Implicit evolve behavior
var I : AnchorIdentInfo .
var A : AttributeSet .
var T : EventTransition .
var D : DocContent .
var K : EventKey .
var S : EventState .

eq implicit-evolve-doc(< I | A > D) = implicit-evolve(I) implicit-evolve-doc(D) .
eq implicit-evolve-doc(< I | A >) = implicit-evolve(I) .
eq implicit-evolve-doc(T D) = implicit-evolve-doc(D) .
eq implicit-evolve-doc(none) = none .
eq implicit-evolve-doc(D, noKey) = implicit-evolve-doc(D) .
eq implicit-evolve-doc(< I | sel-state = S, A > D, K) = implicit-evolve(I)
    implicit-evolve(I, K) implicit-evolve-doc(D, K) .
eq implicit-evolve-doc(< I | A > D, K) = implicit-evolve(I) implicit-evolve-doc(D, K) .
eq implicit-evolve-doc(< I | sel-state = S, A >, K) = implicit-evolve(I)
    implicit-evolve(I, K) .
eq implicit-evolve-doc(< I | A >, K) = implicit-evolve(I) .
eq implicit-evolve-doc(T D, K) = implicit-evolve-doc(D, K) .
eq implicit-evolve-doc(none, K) = none .

var Ev : EventValue .
var Ies : InitEventState .
var Ees : EndEventState .
var Et : EventDuration .

eq implicit-evolve(I) < I | att-state = occurring, A > D = < I | att-state = occurring,
   A > < stop attribution | I | none > D .
eq implicit-evolve(I, Ek) < I | pre-state = occurring, sel-state = Ees, A > D = < I |
    pre-state = occurring, sel-state = Ees, A > < start selection | I | key = Ek > D .
eq implicit-evolve(I, Ek) < I | pre-state = occurring, sel-state = Ies, A > D = < I |
    pre-state = occurring, sel-state = Ies, A > D .
eq implicit-evolve(I) < I | sel-state = occurring, A > D = < I | sel-state = occurring,
   A > < stop selection | I | none > D .
eq implicit-evolve(I) An = An [owise] .
eq implicit-evolve(I, Ek) An = An [owise] .
```

Also, in order to reason about anchor duration, equations that increment the anchor time information are applied. Listing 8.15 shows the definition regarding the $formatting$ step.

Listing 8.16: Maude formatter evolve equations

```
op formatter-evolve-doc : DocContent -> DocContent .
op formatter-evolve-anchor : Anchor -> EvolvingToken .

*** Formatter evolve behavior
var An : Anchor .

eq formatter-evolve-doc(An D) = formatter-evolve-anchor(An) formatter-evolve-doc(D) .
eq formatter-evolve-doc(An) = formatter-evolve-anchor(An) .
eq formatter-evolve-doc(D) = D [owise] .

var Ts : TransientState .
vars Its Its' : InitTransientState .
vars Ets Ets' : EndTransientState .
vars Ss Ss' : StandingState .
vars Ed Ed' : EventDuration .
var Tt : TransitionType .
var Td : Nat .

op evolve-state : TransientState -> StandingState .
```

```
eq evolve-state(starting) = occurring .
eq evolve-state(stopping) = sleeping .
eq evolve-state(aborting) = sleeping .
eq evolve-state(pausing) = paused .
eq evolve-state(resuming) = occurring .

eq formatter-evolve-anchor(< I | att-state = Its , A >) = < I | att-state =
    evolve-state(Its), A > .
eq formatter-evolve-anchor(< I | att-state = Ets , A >) = < I | att-state =
    evolve-state(Ets), A > .
eq formatter-evolve-anchor(< I | pre-state = Its , sel-state = Et, pre-duration = Ed, A
    >) = < I | pre-state = evolve-state(Its), sel-state = evolve-state(Et), pre-duration
    = inc(Ed), A > .
eq formatter-evolve-anchor(< I | pre-state = Ets, sel-state = Et, A >) = < I | pre-state
    = evolve-state(Ets), sel-state = evolve-state(Et), A > .
eq formatter-evolve-anchor(< I | pre-state = Its , sel-state = Ss, pre-duration = Ed, A
    >) = < I | pre-state = evolve-state(Its), sel-state = Ss, pre-duration = inc(Ed), A
    > .
eq formatter-evolve-anchor(< I | pre-state = Ets, sel-state = Ss, A >) = < I | pre-state
    = evolve-state(Ets), sel-state = Ss, A > .
eq formatter-evolve-anchor(< I | pre-state = occurring, sel-state = Et, pre-duration =
    Ed, A >) = < I | pre-state = occurring, sel-state = evolve-state(Et), pre-duration =
    inc(Ed), A > .
eq formatter-evolve-anchor(< I | pre-state = occurring, sel-state = Ss, pre-duration =
    Ed, A >) = < I | pre-state = occurring, sel-state = Ss, pre-duration = inc(Ed), A > .
eq formatter-evolve-anchor(< I | pre-state = Ss, sel-state = Ts, A >) = < I | pre-state
    = Ss, sel-state = evolve-state(Ts), A > .
eq formatter-evolve-anchor(< I | A >) = < I | A > [owise] .
```

A user selection defines the key that was pressed. This information is important since links may define a different document behavior depending on the key pressed. When a selection may occur, Maude chooses one element of set *EventKey* to represent the key pressed. The element *noKey* represents that the selection will not occur at that moment. Listing 8.15 presents the behavior associated to user selection.

Listing 8.17: Maude selection equations

```
op formatter-evolve-doc : DocContent -> DocContent .
op formatter-evolve-anchor : Anchor -> EvolvingToken .

*** Formatter evolve behavior
var An : Anchor .

eq formatter-evolve-doc(An D) = formatter-evolve-anchor(An) formatter-evolve-doc(D) .
eq formatter-evolve-doc(An) = formatter-evolve-anchor(An) .
eq formatter-evolve-doc(D) = D [owise] .

var Ts : TransientState .
vars Its Its' : InitTransientState .
vars Ets Ets' : EndTransientState .
vars Ss Ss' : StandingState .
vars Ed Ed' : EventDuration .
```

```
var Tt : TransitionType .
var Td : Nat .

op evolve-state : TransientState -> StandingState .
eq evolve-state(starting) = occurring .
eq evolve-state(stopping) = sleeping .
eq evolve-state(aborting) = sleeping .
eq evolve-state(pausing) = paused .
eq evolve-state(resuming) = occurring .

eq formatter-evolve-anchor(< I | att-state = Its, A >) = < I | att-state =
    evolve-state(Its), A > .
eq formatter-evolve-anchor(< I | att-state = Ets, A >) = < I | att-state =
    evolve-state(Ets), A > .
eq formatter-evolve-anchor(< I | pre-state = Its, sel-state = Et, pre-duration = Ed, A
    >) = < I | pre-state = evolve-state(Its), sel-state = evolve-state(Et), pre-duration
    = inc(Ed), A > .
eq formatter-evolve-anchor(< I | pre-state = Ets, sel-state = Et, A >) = < I | pre-state
    = evolve-state(Ets), sel-state = evolve-state(Et), A > .
eq formatter-evolve-anchor(< I | pre-state = Its, sel-state = Ss, pre-duration = Ed, A
    >) = < I | pre-state = evolve-state(Its), sel-state = Ss, pre-duration = inc(Ed), A
    > .
eq formatter-evolve-anchor(< I | pre-state = Ets, sel-state = Ss, A >) = < I | pre-state
    = evolve-state(Ets), sel-state = Ss, A > .
eq formatter-evolve-anchor(< I | pre-state = occurring, sel-state = Et, pre-duration =
    Ed, A >) = < I | pre-state = occurring, sel-state = evolve-state(Et), pre-duration =
    inc(Ed), A > .
eq formatter-evolve-anchor(< I | pre-state = occurring, sel-state = Ss, pre-duration =
    Ed, A >) = < I | pre-state = occurring, sel-state = Ss, pre-duration = inc(Ed), A > .
eq formatter-evolve-anchor(< I | pre-state = Ss, sel-state = Ts, A >) = < I | pre-state
    = Ss, sel-state = evolve-state(Ts), A > .
eq formatter-evolve-anchor(< I | A >) = < I | A > [owise] .
```

Links, in the Maude specification, are represented by equations that are applied over anchors whose state represents a transition, inducing the modification of the state of other anchors. Since equations are also used in a transition system state definition, a modification of the document state will be given by the application of all enabled links, as it occurs in the SHM transition system. Once no other link can be applied, the states that represent transitions are evolved to states that represent SHM states.

Once the equations that represent document links depend on the document that will be analyzed, Maude does not define any behavior for the *explicit* step a priori. Listing 8.18 shows the definition regarding the *explicit* step.

Listing 8.18: Maude explicit evolve operations

```
op links : -> DocContent .
op explicit-evolve : String -> EvolvingToken .
```

The model checker tool is used, in this work, for the verification of termination properties. Since termination properties are not decidable, the document execution can be ended if a maximum time is achieved[1]. Listing 8.19 presents the implementation of that option.

Listing 8.19: Ending document execution by maximum time

```
*** raise an error message if the maximum time is reached
ceq [ explicit : T ] = [ explicit : T | END-TIME ] if T == EvolvingMaxDuration .

*** end the document if an error message appear
rl [ S : T | M ]  [D] => [ M | T ] [D] .
```

The Maude specification, as presented, represents the signature of SHM in Maude, besides the generic behavior of SHM documents.

A specific document uses that specification to define its content (anchors) and defines its specific behavior. Regarding the sample multimedia document presented in Listings 3.8 and 3.9, Listing 8.20 shows the representation of its SHM document in Maude.

Listing 8.20: Sample document Maude representation

```
mod NCL-DOC is
protecting NCM-MODEL .

eq Document = < "N1" : "C1.N1" : "desc1" | defContent > < "A1" : "C1.N2" | defAttribute
    > < "N3" : "C1.C2.N3" : "desc2" | defContent > < "N4" : "C1.C2.N4" : "desc2" |
    defContent > .

eq links = explicit-evolve("L1")  explicit-evolve("L2")  .

var V0 : EventValue .
vars A0 A1 : AttributeSet .
var D : DocContent .
var EI : EvolvingInfo .

eq InitActions = < start presentation | "N1" : "C1.N1" : "desc1" > .

*** explicit evolve links
eq EI [ explicit-evolve("L1") < "N1" : "C1.N1" : "desc1" | pre-state = starting , A0 > D
    ] = EI [ < start attribution | "A1" : "C1.N2" | value = "yes" > < "N1" : "C1.N1" :
    "desc1" | pre-state = starting , A0 > D ] .

ceq EI [ explicit-evolve("L2") < "A1" : "C1.N2" | att-value = V0, A0 > < "N1" : "C1.N1"
    : "desc1" | pre-state = stopping , A1 > D ] = EI [ < start presentation | "N3" :
    "C1.C2.N3" : "desc2" > < "A1" : "C1.N2" | att-value = V0, A0 > < "N1" : "C1.N1" :
    "desc1" | pre-state = stopping , A1 > D ] if V0 == "yes" .

```

[1]The maximum execution time is defined by the user, by determining the value of variable $EvolvingMaxDuration$. By default that variable value is one hour

```
ceq EI [ explicit-evolve("L2") < "A1" : "C1.N2" | att-value = V0, A0 > < "N1" : "C1.N1"
    : "desc1" | pre-state = stopping , A1 > D ] = EI [ < start presentation | "N4" :
    "C1.C2.N4" : "desc2" > < "A1" : "C1.N2" | att-value = V0, A0 > < "N1" : "C1.N1" :
    "desc1" | pre-state = stopping , A1 > D ] if V0 == "no" .

*** implicit evolve links
eq implicit-evolve("N1" : "C1.N1" : "desc1") < "N1" : "C1.N1" : "desc1" | pre-state =
    occurring , pre-duration = 900, A0 > D = < stop presentation | "N1" : "C1.N1" :
    "desc1" > < "N1" : "C1.N1" : "desc1" | pre-state = occurring , pre-duration = 900, A0
    > D .

eq implicit-evolve("N3" : "C1.C2.N3" : "desc2") < "N3" : "C1.C2.N3" : "desc2" |
    pre-state = occurring , pre-duration = 200, A0 > D = < stop presentation | "N3" :
    "C1.C2.N3" : "desc2" > < "N3" : "C1.C2.N3" : "desc2" | pre-state = occurring ,
    pre-duration = 200, A0 > D .

eq implicit-evolve("N4" : "C1.C2.N4" : "desc2") < "N4" : "C1.C2.N4" : "desc2" |
    pre-state = occurring , pre-duration = 200, A0 > D = < stop presentation | "N4" :
    "C1.C2.N4" : "desc2" > < "N4" : "C1.C2.N4" : "desc2" | pre-state = occurring ,
    pre-duration = 200, A0 > D .

*** evolution testing
eq docWillExplicitEvolve(< "N1" : "C1.N1" : "desc1" | pre-state = starting , A0 > D) =
    true .

ceq docWillExplicitEvolve(< "A1" : "C1.N2" | att-value = V0, A0 > < "N1" : "C1.N1" :
    "desc1" | pre-state = stopping , A1 > D) = true if V0 == "yes" .

ceq docWillExplicitEvolve(< "A1" : "C1.N2" | att-value = V0, A0 > < "N1" : "C1.N1" :
    "desc1" | pre-state = stopping , A1 > D) = true if V0 == "no" .

eq anchorWillImplicitEvolve(< "N1" : "C1.N1" : "desc1" | pre-state = occurring ,
    pre-duration = 900, A0 > ) = true .

eq anchorWillImplicitEvolve(< "N3" : "C1.C2.N3" : "desc2" | pre-state = occurring ,
    pre-duration = 200, A0 > ) = true .

eq anchorWillImplicitEvolve(< "N4" : "C1.C2.N4" : "desc2" | pre-state = occurring ,
    pre-duration = 200, A0 > ) = true .

endm
```

The Maude model checker tool receives the initial document state and the temporal property to be verified. If that property holds, the model checker returns the boolean value *true*. If not, it returns a counterexample. Listing 8.21 presents the model checker result for testing the reachability and document termination properties using the document in Listing 8.20.

Listing 8.21: Maude model checker result example

```
==========================================
reduce in NCL-DOC : modelCheck(run, reachability("N1" : "C1.N1" : "desc1")) .
rewrites: 214 in 0ms cpu (0ms real) (254458 rewrites/second)
result Bool: true
==========================================
reduce in NCL-DOC : modelCheck(run, reachability("A1" : "C1.N2")) .
rewrites: 213 in 0ms cpu (0ms real) (569518 rewrites/second)
result Bool: true
==========================================
reduce in NCL-DOC : modelCheck(run, reachability("N3" : "C1.C2.N3" : "desc2")) .
rewrites: 241881 in 428ms cpu (429ms real) (564890 rewrites/second)
result Bool: true
==========================================
reduce in NCL-DOC : modelCheck(run, reachability("N4" : "C1.C2.N4" : "desc2")) .
rewrites: 242247 in 429ms cpu (429ms real) (563750 rewrites/second)
result ModelCheckResult: counterexample({[explicit : 0][< "A1" : "C1.N2" | att-value =
    "yes", att-state = starting > < "N1" : "C1.N1" : "desc1" | pre-state = starting,
    pre-duration = 0, sel-state = sleeping, sel-pressedKey = noKey > < "N3" : "C1.C2.N3"
    : "desc2" | pre-state = sleeping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey > < "N4" : "C1.C2.N4" : "desc2" | pre-state = sleeping,
    pre-duration = 0, sel-state = sleeping, sel-pressedKey = noKey >], unlabeled}

{[formatter : 1][< "A1" : "C1.N2" | att-value = "yes", att-state = occurring > < "N1" :
    "C1.N1" : "desc1" | pre-state = occurring, pre-duration = 1, sel-state = sleeping,
    sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = sleeping,
    pre-duration = 0, sel-state = sleeping, sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
    : "desc2" | pre-state = sleeping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey >], unlabeled}

{[explicit : 1][< "A1" : "C1.N2" | att-value = "yes", att-state = stopping > < "N1" :
    "C1.N1" : "desc1" | pre-state = occurring, pre-duration = 1, sel-state = sleeping,
    sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = sleeping,
    pre-duration = 0, sel-state = sleeping, sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
    : "desc2" | pre-state = sleeping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey >], unlabeled}

{[formatter : 900][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1" :
    "C1.N1" : "desc1" | pre-state = occurring, pre-duration = 900, sel-state = sleeping,
    sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = sleeping,
    pre-duration = 0, sel-state = sleeping, sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
    : "desc2" | pre-state = sleeping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey >], unlabeled}

{[implicit : 900][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1" :
    "C1.N1" : "desc1" | pre-state = stopping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = sleeping,
    pre-duration = 0, sel-state = sleeping, sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
    : "desc2" | pre-state = sleeping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey >], unlabeled}

{[explicit : 900][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1" :
    "C1.N1" : "desc1" | pre-state = stopping, pre-duration = 0, sel-state = sleeping,
    sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = starting,
      pre-duration = 0, sel-state = sleeping , sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
      : "desc2" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey >], unlabeled}
27
28 {[formatter : 1100][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1"
      : "C1.N1" : "desc1" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = occurring ,
      pre-duration = 200, sel-state = sleeping , sel-pressedKey = noKey > < "N4" :
      "C1.C2.N4" : "desc2" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey >], unlabeled}
29
30 {[explicit : 1100][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1" :
      "C1.N1" : "desc1" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = stopping ,
      pre-duration = 0, sel-state = sleeping , sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
      : "desc2" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey >], unlabeled}
31
32 {[formatter : 1101][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1"
      : "C1.N1" : "desc1" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = sleeping ,
      pre-duration = 0, sel-state = sleeping , sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
      : "desc2" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey >], unlabeled},
33
34 {[DOC-END | 1101][< "A1" : "C1.N2" | att-value = "yes", att-state = sleeping > < "N1" :
      "C1.N1" : "desc1" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey > < "N3" : "C1.C2.N3" : "desc2" | pre-state = sleeping ,
      pre-duration = 0, sel-state = sleeping , sel-pressedKey = noKey > < "N4" : "C1.C2.N4"
      : "desc2" | pre-state = sleeping , pre-duration = 0, sel-state = sleeping ,
      sel-pressedKey = noKey >], deadlock})
35 ==========================================
36 reduce in NCL-DOC : modelCheck(run, doc-end) .
37 rewrites: 242264 in 419ms cpu (426ms real) (577583 rewrites/second)
38 result Bool: true
```

Notice that the reachability property holds for *N1* (lines 2 to 4), *A1* (lines 6 to 8) and *N3* (lines 10 to 12), but not for *N4*, where a counterexample was presented (lines 14 to 34). The counterexample shows a document execution path where that anchor was not reached. The document termination property holds for that document (lines 36 to 38).

8.2 aNa

aNa (API for NCL Authoring) was created to represent an NCL document. Its structure is optimized so the author of NCL tools that manipulate NCL XML code does not need to worry about the language representation. aNa defines classes that represent NCL elements with the same characteristics presented in Section 6.1 for the NCL LSM.

Every NCL element is represented as a class, which will be called element class. An element class contains the same attributes of the NCL element it represents. Every element class in aNa inherits from basic type *NCLElement*. Additionally, every element that has an *id* attribute inherits from basic type *NCLIdentifiableElement*. Figure 8.4 presents that representation in aNa.

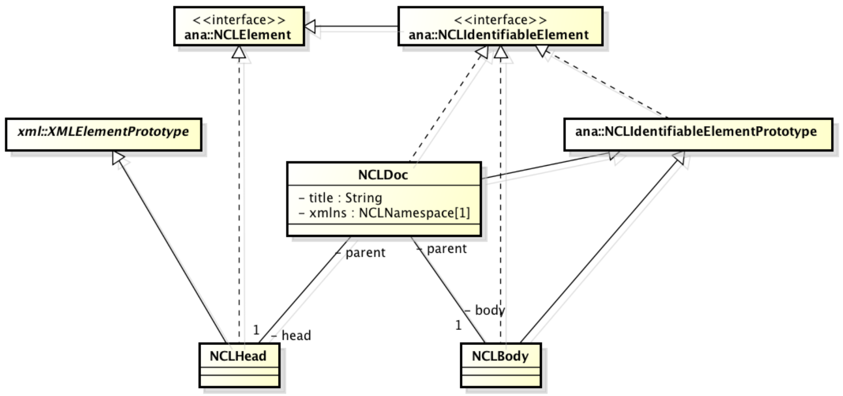


Figure 8.4: NCLDoc class representation

Some element attributes may have a value from a specific value set, like the *xmlns* attribute. In those cases, aNa defines the attribute type as an Enumeration with all the possible values for that attribute (see *xmlns* in Figure 8.4). Sometimes an attribute value can be of more that one type. For example, consider the example presented in Listing 8.22.

Listing 8.22: Attribute value examples

```
<region id="reg1" top="10" left="10"/>
<region id="reg2" top="10.5" left="10"/>
<region id="reg3" top="10.5%" left="10%"/>
```

Notice that element *region* has attributes that may be an integer, a double or a number with a percent sign (%). For those elements, aNa defines basic types that can have any of the possible values that the attribute receives. For the example presented in Listing 8.22, aNa defined the attributes as a PercentType value. In order to represent those values, aNa defines NCL basic types, which will be presented in the following paragraphs.

Figure 8.5 presents the *ArrayType* class. That class represents an attribute value represented by several double numbers separated by commas. It may be created from an array of double numbers or from a string where each value is separated by a comma.

Figure 8.6 presents three classes: *MaxType*, *PercentageType*, *RelativeType*. The *MaxType* class represents an attribute value represented by a positive integer or the string

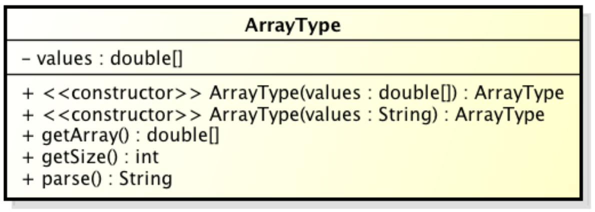

ArrayType
– values : double[]
+ <<constructor>> ArrayType(values : double[]) : ArrayType
+ <<constructor>> ArrayType(values : String) : ArrayType
+ getArray() : double[]
+ getSize() : int
+ parse() : String

Figure 8.5: Array type class

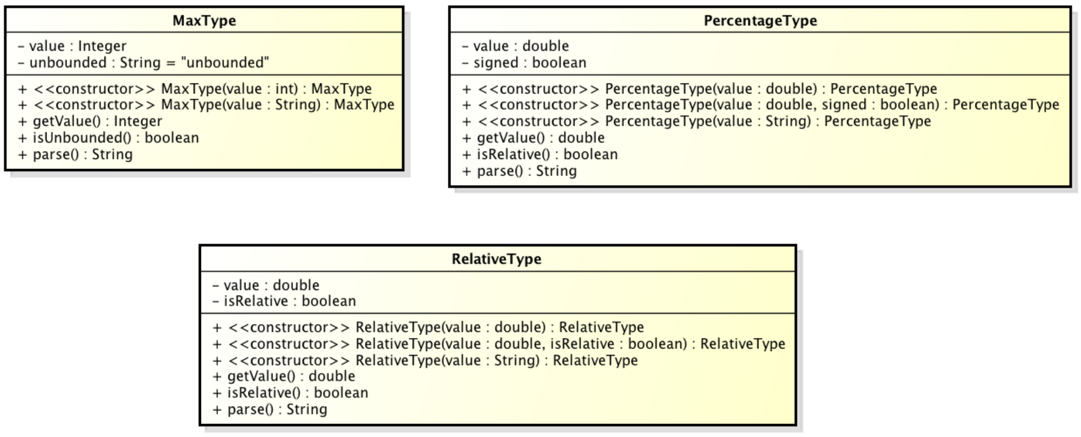

MaxType
– value : Integer
– unbounded : String = "unbounded"
+ <<constructor>> MaxType(value : int) : MaxType
+ <<constructor>> MaxType(value : String) : MaxType
+ getValue() : Integer
+ isUnbounded() : boolean
+ parse() : String
PercentageType
– value : double
– signed : boolean
+ <<constructor>> PercentageType(value : double) : PercentageType
+ <<constructor>> PercentageType(value : double, signed : boolean) : PercentageType
+ <<constructor>> PercentageType(value : String) : PercentageType
+ getValue() : double
+ isRelative() : boolean
+ parse() : String
RelativeType
– value : double
– isRelative : boolean
+ <<constructor>> RelativeType(value : double) : RelativeType
+ <<constructor>> RelativeType(value : double, isRelative : boolean) : RelativeType
+ <<constructor>> RelativeType(value : String) : RelativeType
+ getValue() : double
+ isRelative() : boolean
+ parse() : String

Figure 8.6: Number type classes

“unbounded”. The *PercentageType* class represents an attribute value represented by a double number between 0 and 1 without a percent sign, or between 0 and 100 with a percent sign. The *RelativeType* class represents an attribute value represented by a double number with or without the percent sign. It is used for representing an absolute value or a relative value.

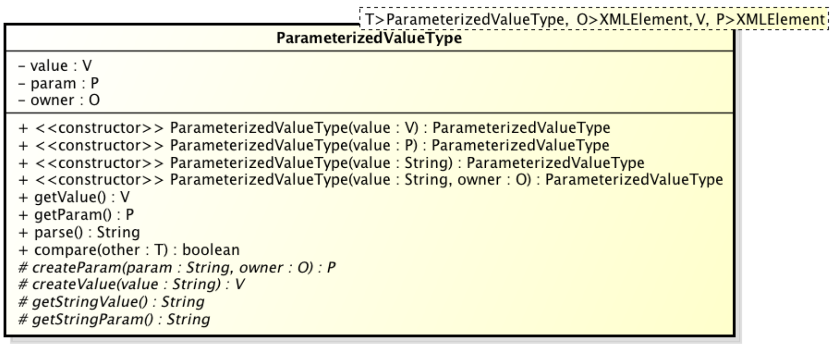

T>ParameterizedValueType, O>XMLElement, V, P>XMLElement
ParameterizedValueType
– value : V
– param : P
– owner : O
+ <<constructor>> ParameterizedValueType(value : V) : ParameterizedValueType
+ <<constructor>> ParameterizedValueType(value : P) : ParameterizedValueType
+ <<constructor>> ParameterizedValueType(value : String) : ParameterizedValueType
+ <<constructor>> ParameterizedValueType(value : String, owner : O) : ParameterizedValueType
+ getValue() : V
+ getParam() : P
+ parse() : String
+ compare(other : T) : boolean
# createParam(param : String, owner : O) : P
# createValue(value : String) : V
# getStringValue() : String
# getStringParam() : String

Figure 8.7: Parameterized type class

Figure 8.7 presents the *ParameterizedValueType* class. This class represents an attribute value represented by a value (like a string, a number, etc) or an element that represents a parameter. This class is specialized as presented in Figure 8.8 in order to create other classes that represent attributes of connector conditions or actions. Those values may have a type (*AssValueType*, *ByType*, *double*, *integer*, *string*, *NCLKey*) or refer to a connector parameter.

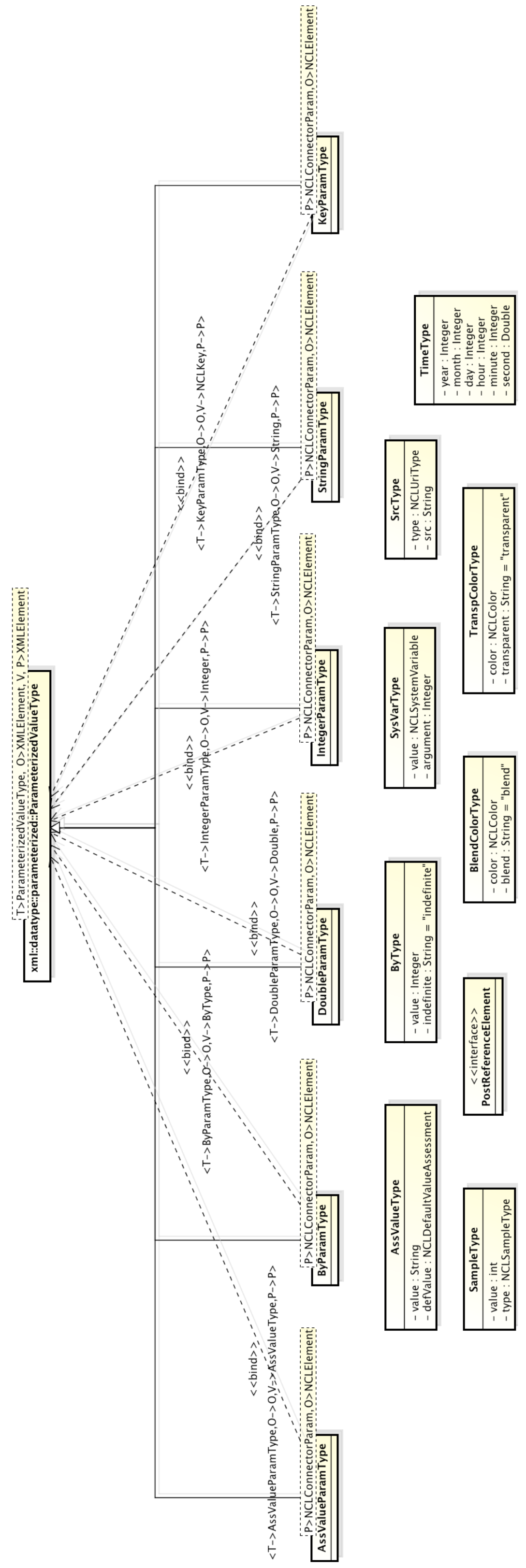

T>ParameterizedValueType, O>XMLElement, V, P>XMLElement
xml::datatype::parameterized::ParameterizedValueType
P>NCLConnectorParam,O>NCLElement
KeyParamType
StringParamType
IntegerParamType
DoubleParamType
ByParamType
AssValueParamType
<<bind>>
<T->KeyParamType,O->O,V->NCLKey,P->P>
<T->StringParamType,O->O,V->String,P->P>
<T->IntegerParamType,O->O,V->Integer,P->P>
<T->DoubleParamType,O->O,V->Double,P->P>
<T->ByParamType,O->O,V->ByType,P->P>
<T->AssValueParamType,O->O,V->AssValueType,P->P>
TimeType
- year : Integer
- month : Integer
- day : Integer
- hour : Integer
- minute : Integer
- second : Double
SrcType
- type : NCLUriType
- src : String
TranspColorType
- color : NCLColor
- transparent : String = "transparent"
SysVarType
- value : NCLSystemVariable
- argument : Integer
BlendColorType
- color : NCLColor
- blend : String = "blend"
ByType
- value : Integer
- indefinite : String = "indefinite"
<<interface>>
PostReferenceElement
AssValueType
- value : String
- defValue : NCLDefaultValueAssessment
SampleType
- value : int
- type : NCLSampleType

Figure 8.8: aNa auxiliary type classes

Figure 8.8 also presents classes that define basic types. *AssValueType* represents the value attribute of a *valueAssessment* element, which can be a string or a default value (listed in an Enumeration). *BlendColorType* represents an attribute value represented by a color or the string "blend", the same for the *TranspColorType* where it could be a color or the string "transparent". *ByType* represents an attribute value represented by an integer or the string "indefinite". *SrcType* represents an attribute value that is a resource location or a time. *TimeType* represents a time value in the format "seconds.fraction", "hour:minute:seconds.fraction" or "year:month:day:hour:minute:seconds.fraction". *SampleType* represents an attribute value that is a number followed by S (for seconds), F (for frames) or NPT (for Normal Play Time).

All classes that represent basic types have methods to create the value from a string and to parse the value in the format used by NCL.

aNa is implemented in Java. The document parsing is done using DOM (Document Object Model) [W3C 2000]. aNa walks through the DOM tree gathering information about the NCL elements and creating Java objects that represent them. During that object creation, aNa already creates references between objects. For example, if aNa finds the value "reg1" in the attribute *region* of a *descriptor* element, it will search for a *region* element in the region base with that *id* in order to create this reference and it will raise an error if no *region* with that *id* is found.

The search for a referred element is done only in the attribute scope. That is, if an attribute indicates the *id* of an element inside the same context, aNa will search for that element only inside the context. For example, suppose a *port* element that defines *component* and *interface* attributes. aNa will search for an NCL node with the *id* defined in the *component* attribute inside the port parent *context* element. Once that element is found, aNa will search for an interface with the *id* defined in the *interface* attribute inside that node.

During parsing, aNa gathers from the DOM representation of an NCL element only the information that makes sense to it, that is, the attributes and child elements defined in the language specification. Also, when reading an NCL document, the Java DOM API already validates if the XML document is well written, that is, if all the XML tags are opened and closed correctly. So, after document parsing, aNa will have a consistent document representation.

aNa raises errors when a wrong definition is found in the NCL document. Listing 8.23 presents an error example. Errors always present the whole path from the root document element to the element where the error occurred. It also shows a message that informs the error found to the author.

Listing 8.23: Parsing error example

```
Error parsing Head > ConnectorBase > CausalConnector(onKeySelectionStop) >
    SimpleCondition
Could not find a param in connector with name: tecla
```

aNa will not parse a document where an element refers to (through an attribute) either an element different from the one required (for that attribute) or a non-existing one. Also, during parsing, aNa gathers from the DOM representation of an NCL element only the information about attributes and child elements defined in the NCL language specification. The parsing step, by itself, already implements the lexical and syntactic, reference and compositionality validation properties and partially implements the hierarchy and attribute validation properties.

The opposite way, that is, creating an NCL XML document from the aNa representation is done by getting, from each Java object, its XML representation. The method that implements it returns a String with the XML element representation. It is worth to highlight that the code returned is indented, making the document reading easier.

Once aNa is developed to be used by tools that manipulate NCL code, it is able to notify the tool that uses it about a modification on an element class. This notification can help the tool to maintain a consistent document representation. For example, suppose a tool that is built to graphically show all document regions. Every time a modification occurs in the position of any region, the tool is notified, so it is able to apply the necessary changes in the graphical position of the modified region. Every element class may have a *ModificationListener*, which will receive notifications when the value of an attribute is set and a child element is added or removed. As this feature may not be necessary for all kinds of tools, the tool is not obliged to implement the *ModificationListener* interface.

Another characteristic of aNa is that it is implemented using parameterized classes, which is done using the Generics Java language feature[2]. Using that feature, aNa element class extensions are simpler, requiring less coding effort.

[2]more information available at http://docs.oracle.com/javase/tutorial/java/generics

8.3 aNaa implementation

To implement the analysis proposed in this work, aNaa (API for NCL Authoring and Analysis) has to:

- create an NCL LSM;
- create the OCL invariants;
- transform an NCL document into an object diagram;
- call TCLib to validate the document structure;
- transform an NCL document into the Maude representation of an SHM document;
- call Maude model checker to verify the document behavior.

In order to be able to do the necessary transformations of an NCL document, aNaa extends aNa. Every aNa class is extended with new methods to perform those transformations. The most important ones are `createObject()` and `createModel()`. The methods signature may vary depending on the class where they are defined, but their behavior is the same in all classes. Method `createObject()` is implemented in all classes that represent NCL elements, it creates and returns the representation of that element as a TCLib object. Method `createModel()` is implemented in the classes that represent elements related to the document behavior, they are the elements of the document body, connectors and rules. This method is used to create the SHM representation of those elements transforming an NCL document into SHM.

Besides the methods, aNaa also defines new packages, which are the *structure*, *shm* and *maude* packages. Figure 8.9 presents an overview of the analysis tool architecture, where rounded rectangles represent external tools.

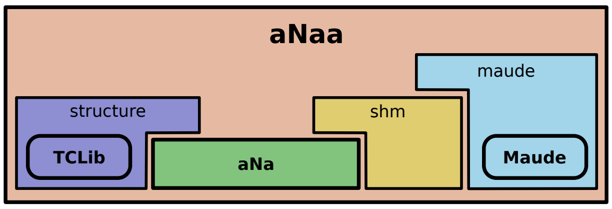


Figure 8.9: aNaa architecture

Package *structure* encapsulates TCLib and is responsible for managing it, creating the NCL LSM and the OCL invariants. It also creates an abstraction layer for helping the creation of TCLib objects, which is used by method `createObject()` and run TCLib to perform the validation. Package *shm* is a representation of SHM in Java. It is used by aNaa during the transformation of an NCL document into SHM, to keep the SHM representation. Package *maude* parses the Java representation of SHM into Maude and is responsible for running Maude to perform the verification.

8.3.1 Static validation implementation

The implementation of the static validation is done as follows. aNaa calls package *structure* to create the NCL LSM inside TCLib (1). The creation of the LSM is always the same, since it does not depend on the document being validated. After the NCL LSM is created, package *structure* creates the OCL invariants for the NCL language inside TCLib (2). Once both the LSM and the invariants are created, aNaa, starting from the document root element (Class *NCLDoc*), parses the document elements, creating TCLib objects (3). During the parsing step, each document element calls the `createObject()` method of its related elements, parsing attribute values and TCLib objects that are related to it. That way, aNaa creates the TCLib objects and associations among them. Listing 8.24 presents the code for transforming an *NCLCompositeRule* element.

Listing 8.24: `createObject()` method implementation for class *NCLCompositeRule*

```
if(thisObject != null)
    return thisObject;

Object aux;

thisObject = new ModelNCLObject(CLASS_NCLCOMPOSITERULE);

if((aux = getId()) != null)
    thisObject.addAttribute(TYPE_STRING, ATT_ID, (String) aux);

if((aux = getOperator()) != null)
    thisObject.addAttribute(ENUM_NCLOPERATOR, ATT_OPERATOR, ((NCLOperator) aux).name());
else
    thisObject.addAttribute(ENUM_NCLOPERATOR, ATT_OPERATOR, ENUM_NULL_LITERAL);

for(NCLTestRule e : getRules())
    thisObject.addAggregation(ATT_RULES, e.createObject());

return thisObject;
```

Once the transformation is finished, aNaa calls TCLib to validate the document (4). If some invariant does not hold, TCLib returns the invariant and a list of TCLib objects that failed on that invariant. aNaa, based on that information, presents to the user the invariant name and body and a description of the NCL element that failed on it. Listing 8.25 shows an aNaa message example, considering the sample document with error presented in Figure 6.6. The element description has the element hierarchy from the document body or head. If the element does not have an id attribute its position and/or other relevant attribute is presented to facilitate the element identification in the document.

Listing 8.25: aNaa validation return message

```
==================================================
ERROR: NCLLink: 'C1 > L1',

invariant106
NCLLink.allInstances()->forAll(i:NCLLink | i.binds->size() >= 2)
==================================================
```

Since both aNaa and the NCL LSM have the same architecture, calling the methods that create TCLib objects of all classes in aNaa does the transformation of the document into an object diagram. Otherwise it would be necessary to process the NCL document in order to transform it into the LSM model.

8.3.2 Behavioral verification implementation

The implementation of the behavioral verification is done as follows. aNaa uses package *shm* to create the representation of the NCL document in SHM (1). aNaa calls package *maude* to parse the SHM document into the Maude representation (2). Once the transformation is finished, aNaa creates the commands to run model checker for each property and for each document anchor (3). Then it calls Maude to verify the document (4). If some property does not hold, Maude presents a counterexample (see Listing 8.20). Based on that information returned by Maude, aNaa presents to the user the anchor and the failed property. Listing 8.26 shows an aNaa message example, regarding the sample document presented in Section 8.1.2.

Listing 8.26: aNaa verification return message

```
==================================================
ERROR: "N4" : "C1.C2.N4" : "desc2"

Anchor is unreachable
==================================================
```

Since a transformation of a multimedia document to SHM is required, NCM (and NCL) was chosen in the implementation once it is also based on events. That way, a simple transformation was required. However, different multimedia document models can be used, by providing a transformation of a document from the model to SHM. Figure 8.10 presents an overview of the verification process done by aNaa.

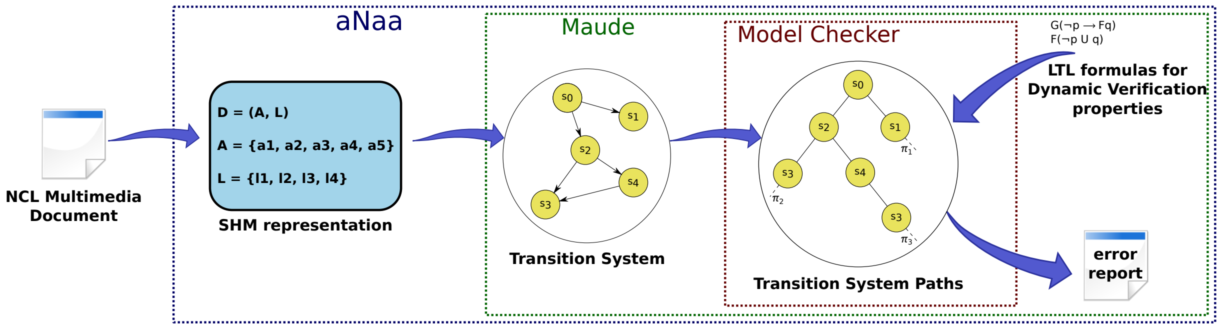


Figure 8.10: aNaa multimedia verification process

The following sections present the architecture of package *shm* and the transformation of an NCL document into SHM.

8.3.2.1 SHM implementation architecture

To help the transformation of an NCL document into SHM, aNaa implements SHM as Java classes. The SHM implementation follows the architecture presented in Figures 8.11, 8.12 and 8.13.

The *Document* class in Figure 8.11 represents the SHM document. It is composed by the document initial actions, the document anchors and the document links.

The initial actions represent the actions that will be performed as the document begins its presentation. An initial action may be an action with an execution condition associated to it, which is represented by the *ConditionalAction* class.

Figure 8.12 presents the *anchor* package. That package defines the structure of an anchor and its type. Every anchor in SHM has an id attribute, a perspective and may also have a layout associated to it. The layout, represented by the *AnchorLayout* class indicates the descriptor used by that anchor. The two types of anchor, attribute anchor and content anchor, are represented by classes *AttributeAnchor* and *ContentAnchor*, respectively.

An attribute anchor represents an attribute of a node and has a value and an attribution event, represented by the *AttributionEventState* class. A content anchor represents a subpart of a node content and has the presentation and selection events, represented by the *PresentationEventState* and *SelectionEventState* classes, respectively.

EventState is a class that represents an event, specifying its type and state. The event

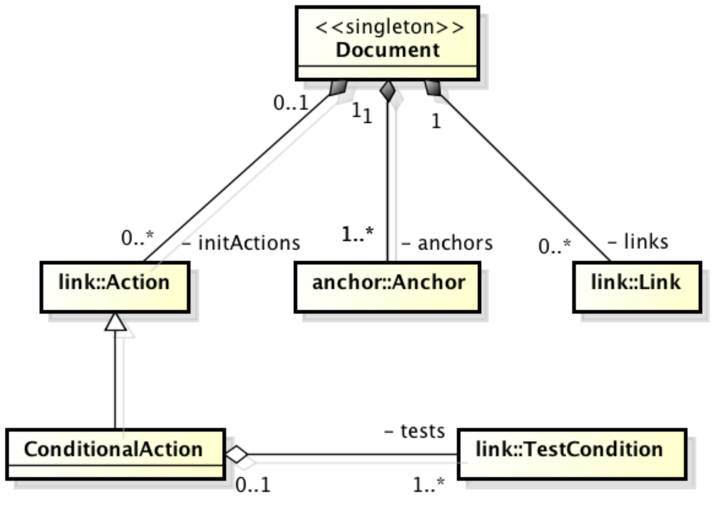


Figure 8.11: SHM implementation

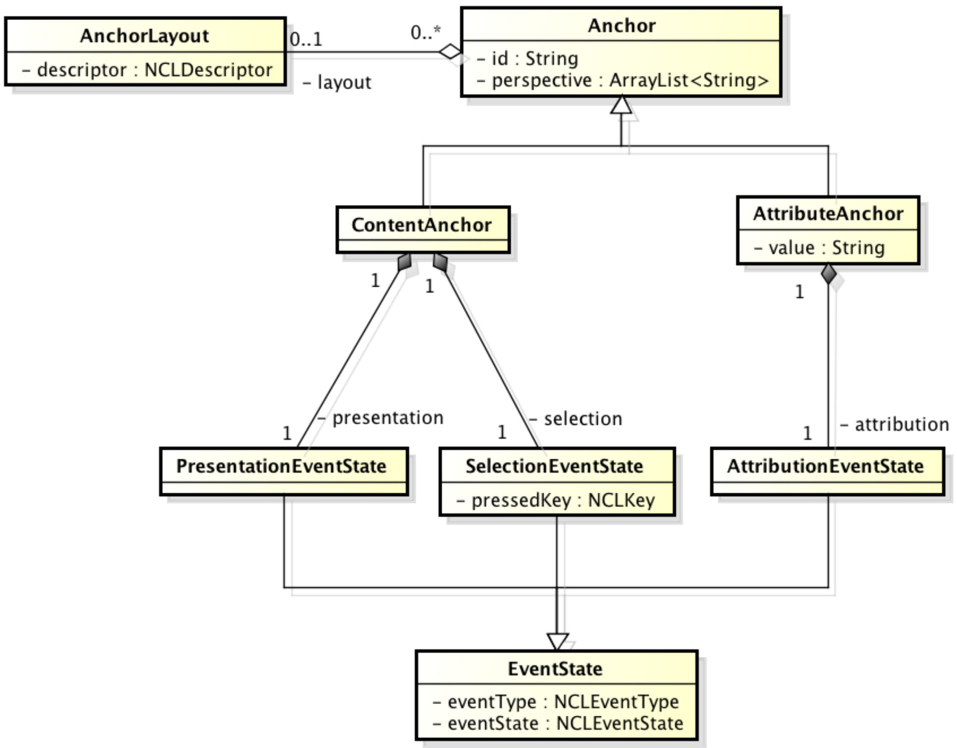


Figure 8.12: SHM implementation anchor package

type defines if it is an attribution, presentation or selection event. The event state defines its state, which can be: *sleeping*, *occurring*, *paused*, *starting*, *stopping*, *aborting*, *pausing* and *resuming*, according to the event state machine presented in Figure 8.3. Classes *PresentationEventState* and *AttributionEventState* extend the *EventState* class specializing it for the presentation and attribution event types, respectively. The *SelectionEventState* class extends *EventState* class specializing it for the selection event type and adds a pressedKey attribute. That attribute represents the key pressed by the user.

Figure 8.13 presents the *link* package. That package defines the structure of an SHM link. An SHM link has two types: an explicit relationship link and an implicit relationship link, represented by the *ExplicitRelationship* and *ImplicitRelationship* classes, respectively.

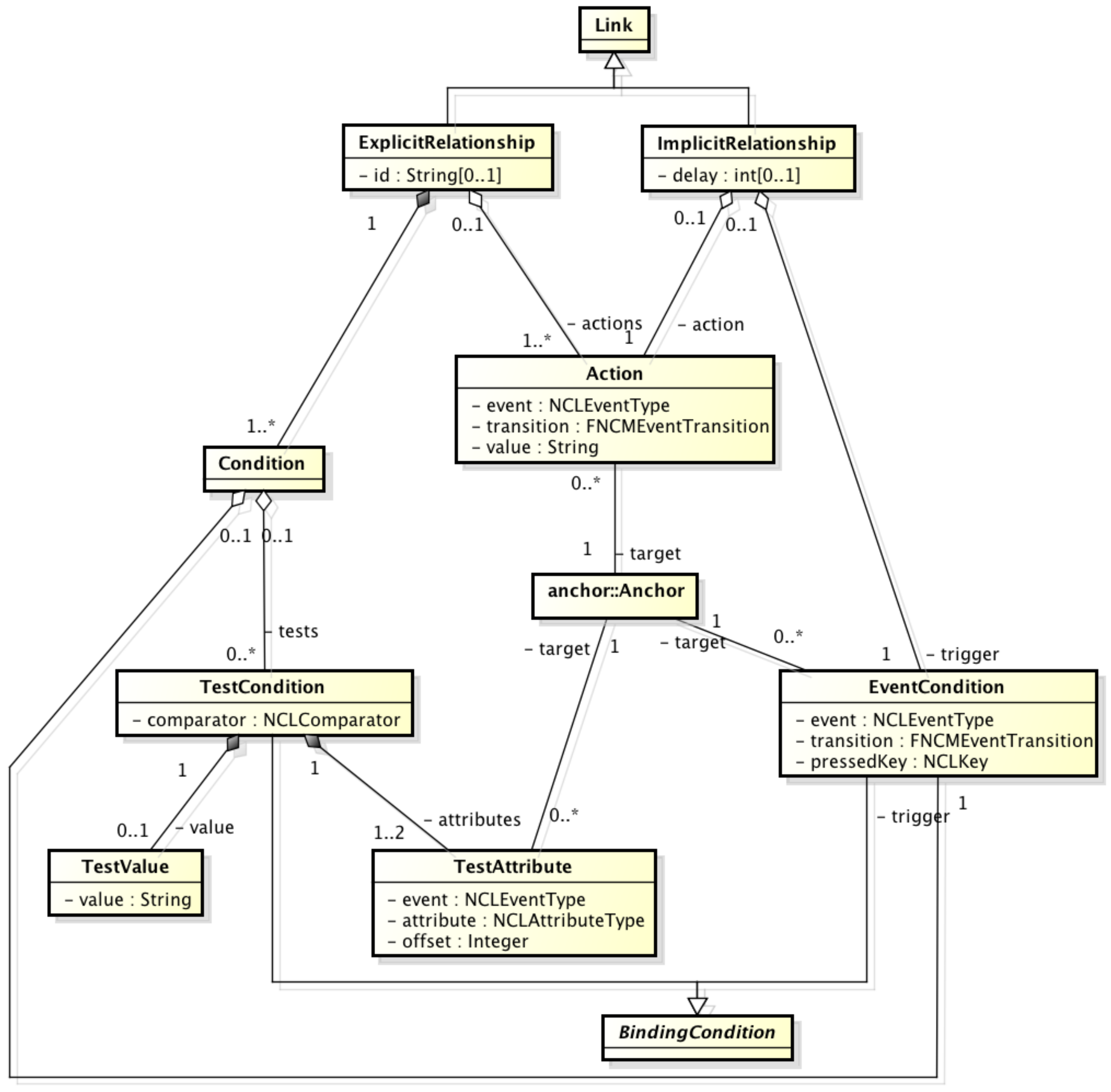


Figure 8.13: SHM implementation link package

An implicit relationship link represents natural event occurrences, like an anchor natural end, or the beginning of a media temporal anchor. It has a trigger condition, which triggers the link execution and an action to be performed over the target anchor. It can also have a delay the link should wait between its condition is satisfied and the action is executed. That delay represents the anchor duration, when the link represents an anchor natural end, or the time when a media temporal anchor is started.

An explicit relationship link represents relationships explicitly defined in an NCL document as link elements. It is composed by one or more conditions and one or more actions. A condition, represented by the *Condition* class, represents a set of link conditions that when satisfied, the link is activated. Those conditions can be an *EventCondition*, which represents an event occurrence, and several *TestConditions*, which represent a comparison of the value of two anchor attributes, possibly the same anchor, or of an anchor attribute and a value.

An action, represented by the *Action* class, has attributes to represent the action to be performed over its target anchor. The event and transition attributes identify the transition that will occur and the event that will have its state changed. The value attribute represents the value to be set in an attribute anchor, in case the action is going to set an attribute value.

8.3.2.2 NCL to SHM transformation

The creation of SHM objects follows the sequence: creation of the document initial actions, creation of the document links (explicit relationships), creation of the document anchors and implicit relationships.

The initial actions creation is done by parsing the document body ports. For each body port found, at least one *Action* object will be created. If a port maps to more than one anchor, for example a port that maps to a context node, one *Action* object for each mapped anchor will be created. Also, if the port maps to an anchor that has an associated condition, for example a port that maps to an anchor inside a switch element, one *ConditionalAction* object will be created for that anchor with a *TestCondition* object representing the associated condition, for example the switch element bind condition (see Figure 8.11).

The creation of SHM links is done by parsing the link elements inside the document. For each link element at least one *ExplicitRelationship* object will be created. At first, link binds are separated into action binds and condition binds. Action binds are the ones whose role is an action role, that is associated to a connector action element, and condition binds are the ones whose role is a condition role, that is associated to a connector condition element. After the binds separation, the connector associated to the link is parsed to determine its *condition domains*. A condition domain represents all the conditions that must be true at the same time so the link is activated. Suppose the connector conditions presented in Listing 8.27.

Listing 8.27: Connector condition domains example

```
<compoundCondition operator='and'>
   <compoundCondition operator='or'>
      <simpleCondition role='A'/>
      <simpleCondition role='B'/>
      <compoundCondition operator='and'>
         <simpleCondition role='C'/>
         <simpleCondition role='D'/>
         <simpleCondition role='E'/>
      </compoundCondition>
   </compoundCondition>
   <compoundStatement operator='and'>
      <assessmentStatement comparator='eq'>
         <attributeAssessment role='F'/>
         <valueAssessment value='abc'/>
      </assessmentStatement>
      <compoundStatement operator='or'>
         <assessmentStatement comparator='eq'>
            <attributeAssessment role='G'/>
            <valueAssessment value='abc'/>
         </assessmentStatement>
         <assessmentStatement comparator='eq'>
            <attributeAssessment role='H'/>
            <valueAssessment value='abc'/>
         </assessmentStatement>
      </compoundStatement>
   </compoundStatement>
</compoundCondition>
```

Figure 8.14 presents the condition roles distribution graphically.

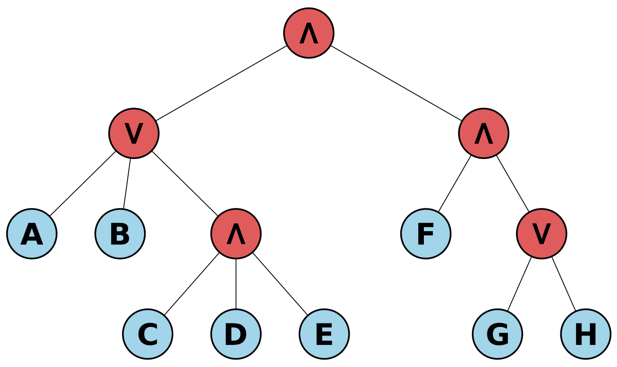


Figure 8.14: Connector condition domains example

The condition domains defined by that connector are: “*AFG*”, “*AFH*”, “*BFG*”, “*BFH*”, “*CDEFG*” and “*CDEFH*”. So, a link that uses that connector will be activated if condition roles A, F and G are satisfied, for example.

After the condition binds are arranged into condition domains, one *Condition* object is created for each domain and, for each bind, a *TestCondition* or *EventCondition* object will be created, depending on the condition defined by the connector. A simpleCondition is transformed into an *EventCondition* object, while an assessmentStatement is transformed into a *TestCondition* object. The *TestCondition* and *EventCondition* target comes from the bind mapped element. The next step is the parsing of the action binds. For each bind an *Action* will be created, gathering the information defined by simpleAction elements defined in the connector.

It is worth to notice that if the NCL link bind defines a descriptor to be used, a new target anchor will be created, merging the anchor defined by the mapped element with the descriptor defined in the bind. Also, if the target anchor has associated conditions, new condition domains may be created, taking into account the anchor conditions.

The creation of SHM anchors and implicit relationship links is done by parsing the nodes defined in the document. If the node is a media node, all its anchors and properties are represented as *Anchor* objects. The created anchor will have the media anchor or property id, the perspective and descriptor defined by the media node. If the media node as a whole is used in a link bind, it is represented as its *all content anchor*. Implicit relations among media node anchors are represented as implicit relationship links. They are, the beginning of a temporal anchor at the time defined by the NCL document, the begin of non-temporal anchors when the whole node begins, the end of those anchors when the whole node ends and the end of an anchor when its duration is reached.

If the node is a context node, the anchors defined by its component nodes are created. The same is done for a switch node, except that a condition is associated to the anchor, representing the condition defined by the switch element for its component activation.

During the SHM objects creation, port and link bind references should be parsed. The parsing of port and link bind references is done as follows.

1. A reference to a media node returns the anchor that represents that media node, that is, its *all content anchor*;
2. A reference to a media node interface point returns the anchor defined by that interface point;
3. A reference to a context node returns the anchors the elements mapped by all context ports return;
4. A reference to a context node interface point returns, if it is a property, the anchor defined by that interface point or, if it is a port, the anchors the element mapped

by that port returns;

5. A reference to a switch node does the same as the context node but adds a condition to each anchor returned, related to the switch bind that maps to that anchor;

6. A reference to a switch node interface point (switch port) returns the anchors mapped by that switch port adding a condition to each anchor returned, related to the switch bind that maps to that anchor;

8.4 Closing remarks

This chapter presented aNaa, an implementation of the validation and verification approaches presented in this work. The tool was implemented to analyze documents specified with the NCM model, using the NCL language. aNaa extends the API aNa, adding to it methods to create the representation of an NCL document as an object diagram and perform its validation, besides methods to create the SHM representation of that document and perform its verification. This chapter also presented the Maude specification to represent an SHM document and the transformation of an NCL document into SHM.

The next chapter presents tests done with aNaa to validate its implementation. The tests were done with documents developed by the NCL user community, to test if the tool could find errors, and some sample documents built to test the tool response time.

Chapter 9

Empirical analysis

To validate the implementation of the analysis proposed in this work, aNaa was used to analyze several sample documents. This was done to test if the tool was capable of identifying specification problems and undesired behaviors. Each sample document was created to fail in one of the properties presented. After those tests, aNaa was used to analyze three documents available in the NCL club[1], a repository for interactive TV applications using NCL. The documents analyzed with aNaa, are the ones supported by the current version of the tool. This chapter: (i) presents the documents tested together with the errors (if any) found in each document; (ii) discusses tests designed to analyze the response time of the Maude model checker when analyzing documents with different sizes; (iii) discusses limitations of the tool.

9.1 The analyzed documents

The first document analyzed describes the “First João” application. “First João” is an interactive TV application developed by TeleMídia Lab, in PUC-Rio, which presents an animation inspired in a chronicle about a famous Brazilian soccer player named Garrincha. As the animation executes, the application plays a sound and presents a background image. At the moment Garrincha dribbles the opponent, a video of kids performing the same dribble is presented. When Garrincha's opponent falls in the ground, a photo of a kid in the same position is presented. At some moment a soccer shoes icon appears. If the TV user presses the remote red key, the animation is resized and a video of a kid thinking about shoes starts playing.

In order to analyze this document, a few changes to the document were necessary, as

[1]http://club.ncl.org.br

follows: moving the definition of the connectors used by the document to the document head, since aNa does not import external bases and link delays were removed, since SHM does not support link delays (Section 7.5). For this document, no specification problems and possible undesired behaviors were found. Every document element is well defined, every anchor is reachable and has an end. Besides, the document as a whole ends. The approximated duration for this document analysis was 4 seconds, 1,5 seconds for the static validation and 2,5 seconds for the dynamic verification.

The second document is “Tic-tac-toe”, also developed by UFMA (Universidade Federal do Maranhão), describing a tic-tac-toe game using just NCL code. It defines nine spaces at the screen where the TV user can mark an X or O. When the TV user selects an empty space, an X or an O is marked, depending on the player of the current round. The game starts with player X. If the TV user presses the remote red key, the game is ended. If the green key is pressed, the game restarts.

This document also had to be modified in order to be analyzed by aNaa. Connector definitions were moved to the document head and a few descriptors using string values in the *focusIndex* attribute had their values changed to numbers, since that attribute in aNa is represented as an Integer value.

This document presented a few specification problems. It was possible to find two rule elements with the same id (*rXLoseDia1_pos4*), making the document to fail in the attribute validation property. Listing 9.1 (lines 4 and 9) shows a fragment of “Tic-tac-toe” where this specification problem appears. It is worth noticing that this problem did not impact the document presentation, since both rules are inside different composite rules and are not used directly.

Listing 9.1: Tic-tac-toe element id specification problem

```
 1 <compositeRule id="RXLoseDia" operator="and">
 2    <compositeRule id="RXLoseDia1" operator="or">
 3       <rule id="rXLoseDia1_pos0" var="_pos0" comparator="ne" value="x" />
 4       <rule id="rXLoseDia1_pos4" var="_pos4" comparator="ne" value="x" />
 5       <rule id="rXLoseDia1_pos8" var="_pos8" comparator="ne" value="x" />
 6    </compositeRule>
 7    <compositeRule id="RXLoseDia2" operator="or">
 8       <rule id="rXLoseDia1_pos2" var="_pos2" comparator="ne" value="x" />
 9       <rule id="rXLoseDia1_pos4" var="_pos4" comparator="ne" value="x" />
10       <rule id="rXLoseDia1_pos6" var="_pos6" comparator="ne" value="x" />
11    </compositeRule>
12 </compositeRule>
```

Another specification problem found was that the document defined a connector with a simpleCondition pointing, through the attribute key, to a connector parameter. However, this parameter does not have its value determined in the link using that connector, making the document to fail in the hierarchy validation property. Listing 9.2 presents the definition of the connector and the link using it (see lines 6 and 22). It is worth noticing that this problem did not impact the document presentation, since the NCL presentation engine uses a default value when the parameter value is not set. This behavior, however, was not described in the standard [ABNT 2007] used in this work.

Listing 9.2: Tic-tac-toe parameter element specification problem

```
 1 <causalConnector id="onKeySelectionPropertyTestStopSetStart">
 2    <connectorParam name="key"/>
 3    <connectorParam name="val"/>
 4    <!-- condition -->
 5    <compoundCondition operator="and">
 6       <simpleCondition role="onSelection" key="$key"/>
 7       <assessmentStatement comparator="eq">
 8          <attributeAssessment role="propertyTest" eventType="attribution"
               attributeType="nodeProperty"/>
 9          <valueAssessment value="$val"/>
10       </assessmentStatement>
11    </compoundCondition>
12    <compoundAction operator="seq">
13       <simpleAction role="stop" max="unbounded" />
14       <simpleAction role="set" value="$val" max="unbounded"/>
15       <simpleAction role="start" max="unbounded" />
16    </compoundAction>
17 </causalConnector>
18
19 ...
20
21 <link xconnector="onKeySelectionPropertyTestStopSetStart">
22    <bind component="empty2" role="onSelection"/>
23    <bind component="noSettings" interface="turn" role="propertyTest">
24       <bindParam name="val" value="x"/>
25    </bind>
26    <bind component="noSettings" interface="_pos2" role="set">
27       <bindParam name="val" value="x"/>
28    </bind>
29    <bind component="noSettings" interface="turn" role="set">
30       <bindParam name="val" value="o"/>
31    </bind>
32    <bind component="x" descriptor="dPos2" role="start"/>
33    <bind component="search_winner" role="start"/>
34    <bind component="empty2" descriptor="dPos2" role="stop"/>
35 </link>
```

aNaa was able to create the SHM document representation. However, the document could not be executed in Maude. Since the document SHM representation is very big and have many conditional equations, it was identified that some equations were tried to be applied a large number of times. This limitation will be commented in Section 9.3. The approximated duration for this document static analysis was 3,5 seconds.

The third document is “Viva Mais”, an application developed by Núcleo de TV Digital Interativa in UFSC (Universidade Federal de Santa Catarina). It presents a TV show discussing several subjects concerning health and welfare and offers some opportunities for an active participation of the TV user.

Once the TV show video starts playing, an interaction icon appears. When the red key is pressed, four different food options appear and the TV user chooses the one he/she prefers to eat. To choose a dish, the TV user presses the colored keys of the remote control. When a dish is chosen, the TV user is informed about the quality of his/her choice, telling if there are missing nutrients or nutrients in excess.

No changes where necessary to analyze the document with aNaa. For this example, no problems in the document syntax were found, since every document element is well defined.

The verification of the document spatio-temporal definition indicated that all document anchors were unreachable, except the anchor mapped by the document body port. The anchor mapped by the body port is an anchor of the application main video. Inside the document, the links that start and stop the remaining anchors are triggered by the video presentation. In the NCL to SHM transformation, an *ImplicitRelationship* link was not created to start the video *all content anchor*, once one of its internal anchors starts its presentation. So the video *all content anchor* does not start its presentation and the document links are not triggered. This limitation will be commented in Section 9.3.

We changed the SHM document that represents the “Viva Mais” NCL document to solve that problem, creating a link to start the video *all content anchor* when its internal anchor starts. After that change, the document could be verified. Possible undesired behaviors were found in that document. In the document test execution, just dish 1 was chosen. Once that dish was chosen, the result for that dish was presented. aNaa verification indicated that the anchors representing the other dishes result were unreachable. This behavior, however, is not related to the document specification, but to a limitation of the tool, which will be commented in Section 9.3.

Also, aNaa indicated that the anchors representing dish 1 and its result do not end, and consequently the document as a whole. Investigating the NCL document, we confirmed that behavior, making the document to fail in the anchor termination and document termination verification properties. The approximated duration for this document analysis was 8 seconds, 2,5 for the static validation and 5,5 seconds for the dynamic verification.

9.2 Maude performance tests

When aNaa calls the Maude model checker to verify the NCL document, it executes the document and searches for a state where the given property holds. Notice that for each property, the document is executed. Since aNaa is designed to be used by NCL authoring tools, it is important to test the response time for documents with different sizes. This section presents performance tests done with the Maude model checker and analyzes those test results.

A document, in the Maude specification, is represented by its anchors and the links among those anchors. For each document anchor, Maude has to test if it will change its related event states and evolve it, for each evolving step (see Section 8.1.2). So, the computations for executing a document should grow with the number of document anchors. Since anchors that are active in the document can have their event states changed (end of an attribution and user selection) freely and its duration has to be incremented (for content anchors), it is possible that active anchors have more influence than inactive ones. Also, for each link Maude has to test if it will change the document state and, of course, execute the link. So, the computations for executing a document should also grow with the number of document links.

Three tests were performed to determine the influence of the number of anchors in Maude response time. The first executes documents with a growing number of active anchors from 1 to 100. The second executes documents with 1 active anchor and a growing number of inactive anchors from 1 to 100. The third executes documents with a growing number of active and inactive anchors from 1 to 50 (giving a total of 100 anchors). The test to determine the influence of links in Maude response time executes documents with one anchor and a growing number of links from 1 to 100. Every link is executed once during the document execution. The duration of the documents (in simulation time) is 1 hour. Both the maximum number of anchors (100) and the document duration (1 hour) were chosen in order to represent very big documents. Figures 9.1 to 9.3 show the results of those tests, presenting the number of Maude rewrites, the number of rewrites/second and the duration (in seconds) of the document execution per anchor/link.

Notice that the duration of the document execution grows very fast with the number

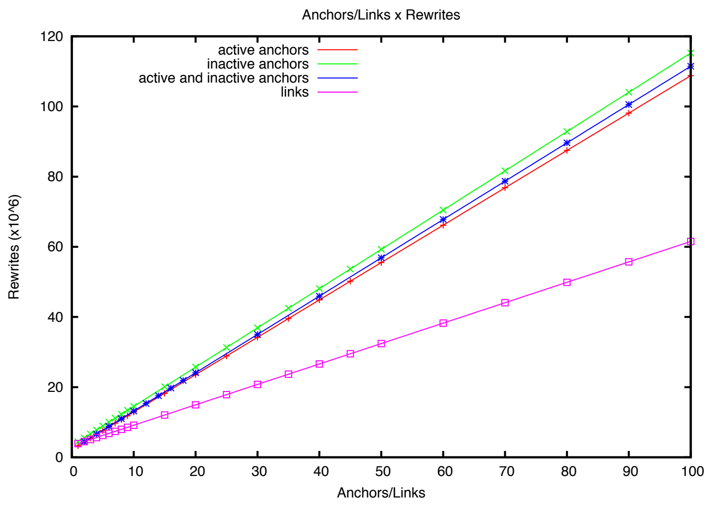


Figure 9.1: Rewrites x anchor/link

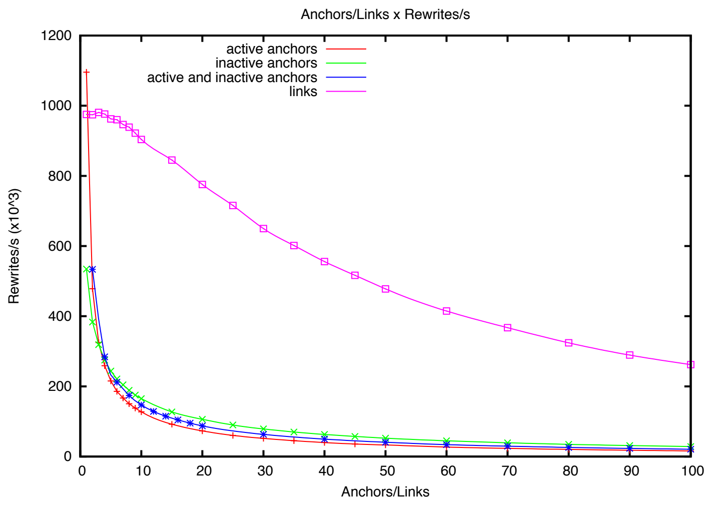


Figure 9.2: Rewrites/second x anchor/link

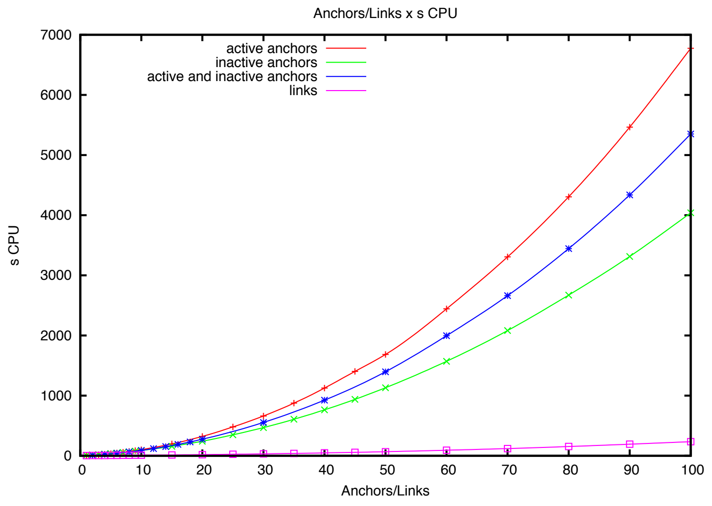


Figure 9.3: Duration x anchor/link

of anchors (Figure 9.3). After a certain number of anchors (80 anchors in average) the document execution duration is bigger than its real execution (3600s). Notice, however, that the number of rewrites/second diminishes also very fast (Figure 9.2) and the number of rewrites grows at a constant rate, as expected (Figure 9.1). It is also possible to observe that links do not affect the increase of rewrites and execution duration as anchors. Links are represented as equations and the way Maude tests the equations that can be applied at a certain moment is not time consuming.

We identified four important problems related to this behavior in our Maude specification representing an SHM document. The following paragraphs discusses these problems.

The first problem identified is the test to determine if the document execution has ended. This test verifies all anchors in the document, testing if that anchor is not active. Since this test is used every time the document evolves, it plays an important role on growing the number of rewrites. It would be better if we could test if the document is active, which is true most of the time, instead of verifying if at least one anchor is active. This can be refined in our Maude specification, by redefining the *ended* operation (see Listing 8.11). Another improvement would be to diminish the use of those tests.

The second problem identified is the boolean operations performed. We identified that a simple boolean operation, like $A \wedge B \vee C$, has a big rewrite cost. Suppose the example presented in Listing 9.3.

Listing 9.3: Boolean test module

```
mod TEST is
   ops var1 var2 var3 : -> Bool .
   eq var1 = true .
   eq var2 = true .
   eq var3 = true .
endm
```

Listing 9.4 shows the result of operations $var1 \wedge var2 \wedge var3$ and $var1 \vee var2 \vee var3$ and the detailed list of rewrites done, using the Maude `profile` command.

Listing 9.4: Boolean test result

```
Maude> red var1 and var2 and var3 .
reduce in TEST : var1 and var2 and var3 .
rewrites: 5 in 0ms cpu (0ms real) (454545 rewrites/second) result Bool: true

Maude> show profile .
eq var1 = true .
rewrites: 1 (20%)

eq var2 = true .
```

```
rewrites: 1 (20%)

eq var3 = true .
rewrites: 1 (20%)

eq true and A:Bool = A:Bool .
rewrites: 2 (40%)



Maude> red var1 or var2 or var3 .
reduce in TEST : var1 or var2 or var3 .
rewrites: 11 in 0ms cpu (0ms real) (~ rewrites/second) result Bool: true

Maude> show profile .
eq var1 = true .
rewrites: 1 (9.09091%)

eq var2 = true .
rewrites: 1 (9.09091%)

eq var3 = true .
rewrites: 1 (9.09091%)

eq true and A:Bool = A:Bool .
rewrites: 2 (18.1818%)

eq false xor A:Bool = A:Bool .
rewrites: 2 (18.1818%)

eq A:Bool xor A:Bool = false .
rewrites: 2 (18.1818%)

eq A:Bool or B:Bool = A:Bool and B:Bool xor A:Bool xor B:Bool .
rewrites: 2 (18.1818%)
```

Notice that the *or* operation takes twice more rewrites that *and*. Diminishing the number of rewrites necessary to get the result of a boolean operation should help diminishing the number of rewrites when executing the document, since that kind of operation is done many times. This could be achieved by defining all boolean operations in normal form or creating functions to represent boolean operations.

The third problem identified, and very important, is regarding the use of conditional equations or rules. This kind of equation (and rule) is mostly used when changing the document evolution step. We identified that equations (and rules) match (Section 4.4) many times, but their condition is not true most of the time. So time and rewrites are spent testing if the equation can be applied. Suppose the example presented in Listing 9.5.

Listing 9.5: Conditional equation test module

```
mod TEST is including STRING .
   sorts anchor state .

   ops sleeping occurring paused : -> state [ctor] .
   op <_,_,_,_> : String state state state -> anchor [ctor] .

   vars s1 s2 s3 : state .

   eq < "a", s1, s2, s3 > = < "b", s1, s2, s3 > .
   ceq < "c", s1, s2, s3 > = < "d", s1, s2, s3 > if s1 == occurring and (s2 == sleeping
       or s3 == paused) .
endm
```

Listing 9.6 shows the result of applying both equations defined in Listing 9.5 and the detailed list of rewrites done, using the Maude `profile` command.

Listing 9.6: Conditional equation result

```
Maude> red < "a", occurring, paused, sleeping > .
reduce in TEST : < "a",occurring,paused,sleeping > .
rewrites: 1 in 0ms cpu (0ms real) (62500 rewrites/second)
result anchor: < "b",occurring,paused,sleeping >

Maude> show profile .
eq < "a",s1,s2,s3 > = < "b",s1,s2,s3 > .
rewrites: 1 (100%)



Maude> red < "c", occurring, paused, sleeping > .
reduce in TEST : < "c",occurring,paused,sleeping > .
rewrites: 8 in 0ms cpu (0ms real) (145454 rewrites/second)
result anchor: < "c",occurring,paused,sleeping >

Maude> show profile .
op _==_ : [state] [state] -> [Bool] .
built-in eq rewrites: 3 (37.5%)

ceq < "c",s1,s2,s3 > = < "d",s1,s2,s3 > if (s2 == sleeping or s3 == paused) and s1 ==
    occurring = true .
lhs matches: 1  rewrites: 0 (0%)
Fragment        Initial tries   Resolve tries   Successes       Failures

eq true and A:Bool = A:Bool .
rewrites: 1 (12.5%)

eq false and A:Bool = false .
rewrites: 1 (12.5%)

eq false xor A:Bool = A:Bool .
rewrites: 2 (25%)

```

```
eq A:Bool or B:Bool = A:Bool and B:Bool xor A:Bool xor B:Bool .
rewrites: 1 (12.5%)
```

Notice that although the conditional equation has no rewrites, when it matches, 8 rewrites are done in order to determine if it can be applied. Diminishing the number of conditional equations and rules in the Maude specification or diminishing the number of rewrites necessary to determine if a conditional equation or rule can be applied should help diminishing the number of rewrites when executing the document.

The fourth problem identified, and the most important, is also regarding the use of conditional equations or rules. In our Maude specification representing an SHM document, some conditional equations were used. Those equations left size define a pattern with a variable to represent the remaining document anchors. When Maude tries to apply those equations and their conditions are not satisfied, it changes the value of the variable representing the remaining document anchors and tries to apply the equation again, due to rewriting modulo associativity and commutativity. As the number of anchors increases, the time and rewrites spent trying to apply those equations increases too. In some cases, as presented in Section 9.1, the time spent is so big that a document can not be executed. Suppose the example presented in Listing 9.7.

Listing 9.7: Conditional equation pattern matching test module

```
mod TEST is including STRING .
   sorts state anchor conf .
   subsort anchor < conf .

   ops sleeping occurring paused : -> state [ctor] .
   op <_,_> : String state -> anchor [ctor] .

   op none : -> conf [ctor] .
   op __ : conf conf -> conf [ctor assoc comm id: none] .

   var s : state .
   var c : conf .

   ceq < "i", s > c = < "ii", occurring > c if s == paused .
endm
```

Listing 9.8 shows the result of applying the equation defined in Listing 9.7 and the detailed list of rewrites done, using the Maude `profile` command.

Listing 9.8: Conditional equation pattern matching result

```
Maude> reduce < "i", sleeping > < "a", sleeping > < "b", sleeping > .
reduce in TEST : < "i",sleeping > < "a",sleeping > < "b",sleeping > .
rewrites: 5 in 0ms cpu (0ms real) (384615 rewrites/second)
```

```
result conf: < "a",sleeping > < "b",sleeping > < "i",sleeping >

Maude> show profile .
op _==_ : [state] [state] -> [Bool] .
built-in eq rewrites: 4 (80%)

ceq c:conf < "i",s:state > = c:conf < "ii",occurring > if s:state == paused = true .
lhs matches: 4    rewrites: 0 (0%)
Fragment          Initial tries    Resolve tries    Successes        Failures
```

Notice that, although the conditional equation has no rewrites, it matches four times when trying to reduce the command. That occurs because Maude tries to apply the conditional equation when variable c is equal to *none*, anchor a, anchor b and both anchors a and b. To solve that problem, conditional equations have to be modified, to define a less general pattern.

9.3 Implementation limitations

Currently, the API used to represent an NCL document (aNa) and the NCL language structure metamodel (presented in Chapter 6) are not able to support NCL documents using full reuse capacity, as documents that reuse external documents.

The current version of aNaa does not consider the NCL reuse features. A node in NCL can reuse another node inside the document reusing its specification and, sometimes, presenting the same behavior. It is important to consider those reuse facilities when creating the SHM representation of an NCL document.

Another limitation is the use of delays in link conditions and actions. Since SHM does not support these delays, they have to be removed from the document before its analysis. Also, as stated in Section 7.5, SHM currently does not fully represent an NCM event. Both the concepts of occurrences and repetitions are not represented by SHM.

aNaa uses JMF (Java Media Framework API)[2] for decoding media objects and getting their duration. When testing the documents presented in the previous section, it was necessary to define the media durations manually in their descriptors, since JMF was not able to return their duration.

As seen in Section 9.1 the “Tic-tac-toe” example could not be executed and we identified that some conditional equations were tried to be applied a large number of times.

[2]more information available at http://www.oracle.com/technetwork/java/javase/specdownload-136569.html

This problem may be solved by refactoring the Maude specification as presented in Section 9.2. It is important, however, to research if the problem that happened with the "Tic-tac-toe" was caused just by a large number of tries for some conditional equations.

Considering the "Viva Mais" example, it was identified that links relating anchors inside the same node were missing. It is important to refactor the transformation to address this limitation. Besides, while verifying that example, just one of the possible dishes was selected. That occurs because the user interaction in the Maude specification was modeled as equations. If, on the other hand, it was modeled as rules, the user interaction would create a new state every time an anchor was selected and consequently a new execution path. With this modification, we could use the Maude model checker to find at least one path were the other dishes were selected. As a consequence, all anchors would be reached.

Chapter 10

Conclusion

Although declarative languages facilitate the creation of interactive applications, when an application has many components and many user interactions, the hypermedia document that describes it gets bigger, with many lines of code. Generally, those documents may become more prone to errors, since the author tends to reuse definitions through code copy. In addition, frequently the author forgets to define some relationships that could make the application not to end, for example, or even defines conflicting relationships, leading the application to an undesirable behavior.

This dissertation discusses a preliminary work on formal validation and verification of NCL documents. The validation and verification may indicate to the author specification problems and possible undesired behaviors. It also provides the author the possibility of correcting them, guaranteeing that the application is well defined before it is made available for the final user. To achieve that validation and verification, a set of validation and verification properties were defined besides the use of an MDA approach for the validation of the document structural definition and the verification of the document behavioral definition. This work also presents aNaa, an analysis tool capable of verifying a multimedia document in order to guarantee its consistency.

Section 10.1 highlights this work's contributions and Section 10.2 presents future and ongoing works.

10.1 Contributions

This work defined a set of properties that, when satisfied in a multimedia document, it can be considered consistent (Chapter 5). The properties here presented were divided in two parts: properties for the validation of the document structural definition and the verification of the document behavioral definition. Those properties were called static and dynamic properties, respectively. There are seven static validation properties: lexical and syntactic, hierarchy, attribute, reference, compositionality, composition nesting and element reuse validation properties. The dynamic verification was divided into four properties: reachability, anchor termination, document termination and resource verification properties. Those properties came from the study of related work and were used as basis for the implementation of an analysis tool.

In Chapters 6 and 7, we presented a model-driven approach for the analysis of multimedia documents. And Chapter 8 presented that approach implementation in an analysis tool called aNaa (API for NCL Authoring and Analysis).

The static validation used a representation of the language structure as a metamodel and defined a set of OCL invariants that represent the static validation properties. It also represented a multimedia document as an instance of that metamodel and used this representation as input to a tool that validated the invariants. The dynamic verification was based on SHM, a general model proposed for representing multimedia document presentation behavior, and its transformation into a transition system, where each temporal property was verified with the use of LTL formulas and a model checker tool.

Once NCL documents have to be transformed into the language metamodel and the SHM notation, it was important to be able to read NCL documents and create a representation for them. To achieve this goal, we developed aNa, an API for representing NCL documents. That API represents the NCL elements with the same characteristics of the NCL Language Structure Metamodel. The API implementation was done so it can be used by NCL authoring tools as a common core.

The analysis tool here presented, aNaa, extends aNa adding to the API the capability of analyzing NCL documents. It implements the static validation and the dynamic verification, transforming an NCL document into the different representations and calling external tools to perform the document analysis. aNa and aNaa were implemented in Java, making its use possible by NCL authoring tools for analyzing NCL documents.

The following topics summarize this work contributions.

- The definition of a set of properties that, when satisfied in a multimedia document, it can be considered consistent;
- The use of a model-driven approach for the analysis of multimedia documents, which brings the following contributions:

  - The definition of a general method for the validation of multimedia document structural definitions;
  - The definition of a model that represents a multimedia document spatio-temporal specification for NCL documents;
  - The definition of a general method for the verification of multimedia document behavioral definition.

- Development of the API aNa (API for NCL Authoring) for representing NCL documents, which brings the following contributions:
  - The creation of a data model specifically for representing NCL documents;
  - The implementation of a common core for NCL authoring tools, making possible to exchange object-oriented data among different tools without the need to generate XML code;
- The development of API aNaa (API for NCL Authoring and Analysis), making possible for authoring tools to analyze an NCL document being authored.

10.2 Future works

Regarding the validation of multimedia document structural definition, a future work is the use of the same method here proposed for NCL for validating documents specified with different languages, such as SMIL and HTML5, by defining an LSM and OCL invariants for those languages. This is important in order to prove the generality of the set of static validation properties.

The validation of multimedia document structural definition is language dependent, since an LSM and OCL invariants have to be defined for each different language. In order to generalize this validation, a future work is the creation of basic multimedia linguistic constructions, general enough to represent multimedia documents described with different languages, to be used as basis for the static definition validation.

The verification of dynamic properties was implemented based on SHM, presented in Chapter 7. Since a transformation of a multimedia document to SHM was required, NCL was chosen as the authoring language in the implementation test case, once it is also based on events, and a simple transformation was required. However, different authoring languages could be used, requiring a document transformation to SHM. Another future work is the implementation of the dynamic properties verification for documents specified with different languages, such as SMIL, in order to prove the generality of the SHM model and of the set of verification dynamic properties.

An SHM document is represented in Maude as a rewrite theory. A future work is the formal proof of the correctness of that transformation.

aNaa is capable of indicating possible undesired behaviors in the authored document. However, it is not capable of verifying if the author desired behavior was described by the analyzed multimedia document. This verification can be achieved from a user description of the expected document behavior and the comparison of that behavior with the one that comes from the Maude rewrite theory execution. Another approach is to make possible to the author to define a temporal property (in temporal logic) to be investigated. That facility was left as future work.

Chapter 8 presented the messages returned by aNaa to indicate a possible specification problem. It is important that the messages returned to the user are clear enough to help even non-expert authors. One possible approach is to simulate the path returned by the model checker counterexample. A future work is to improve these messages making them clearer.

The Maude rewrite theory representation, as the SHM model, is not capable of representing real time properties of multimedia documents, such as link delays. An ongoing work is the improvement of SHM and Maude rewrite theory in order to represent these properties. Besides, the response time of the Maude model checker tool proved to be big for large documents. The analysis of that response time and the behavior of some Maude constructs highlighted in the Maude specification must be refactored. This specification refactoring is an ongoing work. Another ongoing work related to the Maude specification is the modeling of user selection as rules (currently it is modeled as equations). With this modification every user interaction would create a new state and consequently a new execution path. We believe that this behavior is more correct than the one currently in use.

NCL documents may have nodes representing Lua scripts. These nodes are sometimes used to determine the behavior of the NCL document, since Lua scripts can generate events that are interpreted by links in the NCL document. An important future work is the representation of the behavior of those scripts in Maude, when necessary. Modeling that behavior will make possible to determine when, during the node representing a Lua script, an event will be generated, enabling those documents analysis.

Another facility provided by NCL is live editing commands [ABNT 2011]. A live editing command enables a TV broadcaster to edit an NCL document, that is, create elements, change attribute values, etc. A future work is to represent in the SHM model those live editing commands.

Multimedia documents are sometimes executed in platforms with limited resources, such as memory, bandwidth, etc. Such limitation may interfere in the multimedia document execution. For example, consider a media node retrieved from a media server. It is possible that an execution platform with a limited bandwidth could add a delay in that media presentation, changing the behavior of the multimedia document. A future work is to enable the analysis tool user to define some characteristics of the execution platform prior to the document analysis.

At last, an important ongoing work is to solve aNaa limitations presented in Section 9.3. Also, in order to guarantee that the tool works properly, another important future work is to define a set of multimedia documents to be used as benchmark.

APPENDIX A – OCL invariant list

Listing A.1: Hierarchy OCL invariants

```
------------------------------------------------------------------------------------------
-- A document must have at least a head or a body
context NCLDoc inv:
    self.head->notEmpty() or self.body->notEmpty()


------------------------------------------------------------------------------------------
-- The document body can not be empty
context NCLBody inv:
    self.nodes->notEmpty() or self.properties->notEmpty() or self.metas->notEmpty() or self.metadatas->notEmpty()


------------------------------------------------------------------------------------------
-- The document body can not have links without having nodes or properties
context NCLBody inv:
    (self.ports->notEmpty() or self.links->notEmpty()) implies (self.nodes->notEmpty() or self.properties->notEmpty())


------------------------------------------------------------------------------------------
-- The document head can not be empty
context NCLHead inv:
    self.importedDocumentBase->notEmpty() or self.ruleBase->notEmpty() or self.transitionBase->notEmpty()
    or self.regionBase->notEmpty() or self.descriptorBase->notEmpty() or self.connectorBase->notEmpty()


------------------------------------------------------------------------------------------
-- A assessmentStatement must have two attributeAssessment or an attributeAssessment and a valueAssessment
context NCLAssessmentStatement inv:
    if self.valueAssessment->notEmpty() then
        self.attributeAssessments->size() = 1
    else
        self.attributeAssessments->size() = 2
    endif


------------------------------------------------------------------------------------------
-- A causalConnector must have a condition
context NCLCausalConnector inv:
    self.condition->notEmpty()


------------------------------------------------------------------------------------------
-- A causalConnector must have an action
context NCLCausalConnector inv:
    self.action->notEmpty()


------------------------------------------------------------------------------------------
-- A compoundAction must have at least two inner actions
context NCLCompoundAction inv:
    self.actions->size() >= 2


------------------------------------------------------------------------------------------
```

```
-- A compoundCondition must have at least two inner conditions and/or assertives
context NCLCompoundCondition inv:
    (self.conditions->size() + self.statements->size()) >= 2


---------------------------------------------------------------------------------
-- A compoundStatement must have at least two assertives
context NCLCompoundStatement inv:
    self.statements->size() >= 2


---------------------------------------------------------------------------------
-- A connectorBase can not be empty
context NCLConnectorBase inv:
    self.connectors->notEmpty() or self.imports->notEmpty()


---------------------------------------------------------------------------------
-- A descriptorBase can not be empty
context NCLDescriptorBase inv:
    self.descriptors->notEmpty() or self.imports->notEmpty()


---------------------------------------------------------------------------------
-- A descriptorSwitch must have at least one descriptor and one bindRule
context NCLDescriptorSwitch inv:
    self.descriptors->size() >= 1 and self.binds->size() >= 1


---------------------------------------------------------------------------------
-- A regionBase can not be empty
context NCLRegionBase inv:
    self.regions->notEmpty() or self.imports->notEmpty()


---------------------------------------------------------------------------------
-- A importedDocumentBase can not be empty
context NCLImportedDocumentBase inv:
    self.imports->notEmpty()


---------------------------------------------------------------------------------
-- A ruleBase can not be empty
context NCLRuleBase inv:
    self.rules->notEmpty() or self.imports->notEmpty()


---------------------------------------------------------------------------------
-- A compositeRule mut have at least two inner rules
context NCLCompositeRule inv:
    self.rules->size() >= 2


---------------------------------------------------------------------------------
-- A transitionBase can not be empty
context NCLTransitionBase inv:
    self.transitions->notEmpty() or self.imports->notEmpty()


---------------------------------------------------------------------------------
-- A switchPort must have at least one mapping
context NCLSwitchPort inv:
    self.mappings->notEmpty()

---------------------------------------------------------------------------------
-- A context must have at least one inner node or property or refer another context
context NCLContext inv:
    self.nodes->notEmpty() or self.properties->notEmpty() or self.refer->notEmpty()


---------------------------------------------------------------------------------
-- A switch must have at least one inner node and one bind
context NCLSwitch inv:
    self.binds->notEmpty() and self.nodes->notEmpty()


------------------------------------------------------------------------------------------------------------
-- A link must have at least two binds
context NCLLink inv:
        self.binds->size() >= 2


------------------------------------------------------------------------------------------------------------
-- The number of link binds must respect the cardinality defined by the link connector
context NCLLink inv:
    self.xconnector.getRolesFromCausalConnector()->forAll(role | (role.getMax() <> -1 implies self.getBindsFromRole(
         role)->size() <= role.getMax()) and (role.getMin() <> -1 implies self.getBindsFromRole(role)->size() >= role.
         getMin()))


------------------------------------------------------------------------------------------------------------
-- The link must define a value to all connector parameters
context NCLLink inv:
    self.getParameters()->collect(p | p.name)->flatten()->asSet()->size() = self.xconnector.conn_params->size()
```

Listing A.2: Attribute OCL invariants

```
------------------------------------------------------------------------------------------------------------
-- An identifiable element id must be unique inside the whole document (except for properties)
context NCLIdentifiableElement inv:
    if not (self.oclIsKindOf(NCLProperty) or self.oclIsKindOf(NCLConnectorParam)) and self.id->notEmpty() then
        NCLIdentifiableElement.allInstances()->forAll(d |
            if d.id->notEmpty() and d <> i then
                d.id.value <> self.id.value
            else
                true
            endif)
    else
        true
    endif


------------------------------------------------------------------------------------------------------------
-- A property id must be unique inside its parent media
context NCLMedia inv:
    self.properties->forAll(p1 : NCLProperty, p2 : NCLProperty | p1 <> p2 implies p1.name <> p2.name)


------------------------------------------------------------------------------------------------------------
-- A descriptor focusIndex must be unique inside the whole document
context NCLDescriptor inv:
    if self.focusIndex->notEmpty() then
        NCLDescriptor.allInstances()->forAll(d:NCLDescriptor |
            if d <> i and d.focusIndex->notEmpty() then d.focusIndex.value <> i.focusIndex.value else true endif)
    else
        true
    endif


------------------------------------------------------------------------------------------------------------
-- A connector role must be unique inside the connector
context NCLCausalConnector inv:
    self.getRolesFromCausalConnector()->forAll(r1, r2 | r1 <> r2 implies r1.differ(r2)))



------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------
-- An identifiable element must define the id attribute (except for the ones listed)
context NCLIdentifiableElement inv:
    not (self.oclIsKindOf(NCLImportedDocumentBase) or
         self.oclIsKindOf(NCLRuleBase) or
         self.oclIsKindOf(NCLTransitionBase) or
         self.oclIsKindOf(NCLRegionBase) or
         self.oclIsKindOf(NCLDescriptorBase) or
```

```
          self.oclIsKindOf(NCLConnectorBase) or
          self.oclIsKindOf(NCLBody) or
          self.oclIsKindOf(NCLLink)) implies self.id->notEmpty()


-----------------------------------------------------------------------
-- A document must define the xmlns attribute
context NCLDoc inv:
    self.xmlns <> NCLNamespace::NULL


-----------------------------------------------------------------------
-- An assessmentStatement must define the comparator attribute
context NCLAssessmentStatement inv:
    self.comparator <> NCLComparator::NULL


-----------------------------------------------------------------------
-- An attributeAssessment must define a role
context NCLAttributeAssessment inv:
    self.role->notEmpty()


-----------------------------------------------------------------------
-- An attributeAssessment must define the eventType attribute
context NCLAttributeAssessment inv:
    self.eventType <> NCLEventType::NULL


-----------------------------------------------------------------------
-- A compoundAction must define the operator attribute
context NCLCompoundAction inv:
    self.operator <> NCLActionOperator::NULL


-----------------------------------------------------------------------
-- A compoundCondition must define the operator attribute
context NCLCompoundCondition inv:
    self.operator <> NCLConditionOperator::NULL


-----------------------------------------------------------------------
-- A compoundStatement must define the operator attribute
context NCLCompoundStatement inv:
    self.operator <> NCLOperator::NULL


-----------------------------------------------------------------------
-- A simpleAction must define a role
context NCLSimpleAction inv:
    self.role->notEmpty()


-----------------------------------------------------------------------
-- A valueAssessment must define the value attribute
context NCLValueAssessment inv:
    self.value->notEmpty() or self.defValue <> NCLDefaultValueAssessment::NULL or self.parValue->notEmpty()


-----------------------------------------------------------------------
-- A simpleCondition must define a role
context NCLSimpleCondition inv:
    self.role->notEmpty()


-----------------------------------------------------------------------
-- A descriptorParam must define the name attribute
context NCLDescriptorParam inv:
    self.name <> NCLAttributes::NULL


-----------------------------------------------------------------------
-- A descriptorParam must define the value attribute
```

```
context NCLStringDescriptorParam inv:
    self.value->notEmpty()


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLBooleanDescriptorParam inv:
    self.value->notEmpty()


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLColorDescriptorParam inv:
    self.value <> NCLColor::NULL


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLPercentDescriptorParam inv:
    self.value->notEmpty()


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLFitDescriptorParam inv:
    self.value <> NCLFit::NULL


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLScrollDescriptorParam inv:
    self.value <> NCLScroll::NULL


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLPlayerLifeDescriptorParam inv:
    self.value <> NCLPlayerLife::NULL


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLFontWeightDescriptorParam inv:
    self.value <> NCLFontWeight::NULL


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLFontVariantDescriptorParam inv:
    self.value <> NCLFontVariant::NULL


------------------------------------------------------------
-- A descriptorParam must define the value attribute
context NCLDoubleDescriptorParam inv:
    self.value->notEmpty()


------------------------------------------------------------
-- The descriptorSwitch bindRule must define the rule attribute
context NCLDescriptorBindRule inv:
    self.rule->notEmpty()

------------------------------------------------------------
-- The descriptorSwitch bindRule must define the constituent attribute
context NCLDescriptorBindRule inv:
    self.constituent->notEmpty()


------------------------------------------------------------
-- An import must define the alias attribute
context NCLImport inv:
    self.alias->notEmpty()
```

```


-------------------------------------------------------------------------------------
-- An import must define the documentURI attribute
context NCLImport inv:
    self.documentURI->notEmpty()


-------------------------------------------------------------------------------------
-- A compositeRule must define the operator attribute
context NCLCompositeRule inv:
    self.operator <> NCLOperator::NULL


-------------------------------------------------------------------------------------
-- A rule must define the var attribute
context NCLRule inv:
    self.var->notEmpty()


-------------------------------------------------------------------------------------
-- A rule must define the comparator attribute
context NCLRule inv:
    self.comparator <> NCLComparator::NULL


-------------------------------------------------------------------------------------
-- A rule must define the value attribute
context NCLRule inv:
    self.value->notEmpty()


-------------------------------------------------------------------------------------
-- A transition must define the type attribute
context NCLTransition inv:
    self.type <> NCLTransitionType::NULL


-------------------------------------------------------------------------------------
-- A metadata must not be empty
context NCLMetadata inv:
    self.rdfTree->notEmpty()


-------------------------------------------------------------------------------------
-- A meta must define the name attribute
context NCLMeta inv:
    self.name->notEmpty()


-------------------------------------------------------------------------------------
-- A meta must define the content attribute
context NCLMeta inv:
    self.mcontent->notEmpty()


-------------------------------------------------------------------------------------
-- A port must define the component attribute
context NCLPort inv:
    self.component->notEmpty()


-------------------------------------------------------------------------------------
-- A bindRule must define the rule attribute
context NCLSwitchBindRule inv:
    self.rule->notEmpty()


-------------------------------------------------------------------------------------
-- A bindRule must define the constituent attribute
context NCLSwitchBindRule inv:
    self.constituent->notEmpty()

```

```

---------------------------------------------------------------------------------------
-- A mapping must define the component attribute
context NCLMapping inv:
    self.component->notEmpty()


---------------------------------------------------------------------------------------
-- A link must define the xconnector attribute
context NCLLink inv:
        self.xconnector->notEmpty()


---------------------------------------------------------------------------------------
-- A bind must define a role
context NCLBind inv:
    self.role->notEmpty()


---------------------------------------------------------------------------------------
-- A bind must define the component attribute
context NCLBind inv:
    self.component->notEmpty()


---------------------------------------------------------------------------------------
-- A link or bind parameter must define the name attribute
context NCLParam inv:
    self.name->notEmpty()


---------------------------------------------------------------------------------------
-- A link or bind parameter must define the value attribute
context NCLParam inv:
    self.value->notEmpty()


---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
-- An assessmentStatement must compare two values of the same type
context NCLAssessmentStatement inv:
    if self.attributeAssessments->size() = 2 then
        self.attributeAssessments->forAll(i1 : NCLAssessmentStatement, i2 : NCLAssessmentStatement | i1.attributeType =
              i2.attributeType)
    else
        self.attributeAssessments->asOrderedSet()->first().attributeType = NCLAttributeType::STATE implies self.
              valueAssessment.defValue <> NCLDefaultValueAssessment::NULL
    endif


---------------------------------------------------------------------------------------
-- An attributeAssessment can define the key attribute when the eventType attribute is equals to selection
context NCLAttributeAssessment inv:
    if self.eventType = NCLEventType::SELECTION then
        self.key <> NCLKey::NULL
    else
        self.key = NCLKey::NULL
    endif


---------------------------------------------------------------------------------------
-- An attributeAssessment must define the attributeType attribute equals to occurrences or state when the eventType
     attribute is equals to selection
context NCLAttributeAssessment inv:
    self.eventType = NCLEventType::SELECTION implies (self.attributeType = NCLAttributeType::OCCURRENCES or self.
          attributeType = NCLAttributeType::STATE)


---------------------------------------------------------------------------------------
-- An attributeAssessment must define the attributeType attribute different of nodeProperty when the eventType
     attribute is equals to presentetion
context NCLAttributeAssessment inv:
    self.eventType = NCLEventType::PRESENTATION implies self.attributeType <> NCLAttributeType::NODE_PROPERTY
```

```



-------------------------------------------------------------------------------------------------------
-- A simpleAction must define a standard role or the eventType and actionType attributes
context NCLSimpleAction inv:
    if self.role->notEmpty() then
        self.role.aname <> NCLDefaultActionRole::NULL or (self.eventType <> NCLEventType::NULL and self.actionType <>
             NCLEventAction::NULL)
    else
        true
    endif


-------------------------------------------------------------------------------------------------------
-- A simpleAction can define the value attribute when the action is an attribution action
context NCLSimpleAction inv:
    if self.role->notEmpty() then
        self.value->notEmpty() implies (self.role.aname = NCLDefaultActionRole::SET or (self.eventType = NCLEventType::
             ATTRIBUTION and self.actionType = NCLEventAction::START))
    else
        true
    endif


-------------------------------------------------------------------------------------------------------
-- A simpleAction can define the duration attribute when the action is an attribution action
context NCLSimpleAction inv:
    if self.role->notEmpty() then
        self.duration->notEmpty() implies (self.role.aname = NCLDefaultActionRole::SET or (self.eventType =
             NCLEventType::ATTRIBUTION and self.actionType = NCLEventAction::START))
    else
        true
    endif


-------------------------------------------------------------------------------------------------------
-- A simpleAction can define the repeat attribute when the action is a presentation action
context NCLSimpleAction inv:
    if self.role->notEmpty() then
        self.repeat->notEmpty() implies (self.role.aname = NCLDefaultActionRole::START or (self.eventType =
             NCLEventType::PRESENTATION and self.actionType = NCLEventAction::START))
    else
        true
    endif


-------------------------------------------------------------------------------------------------------
-- A simpleAction can define the by attribute when the duration attribute is defined
context NCLSimpleAction inv:
    self.by->notEmpty() implies self.duration->notEmpty()


-------------------------------------------------------------------------------------------------------
-- A simpleAction can define the repeatDelay attribute when the repeat attribute is defined
context NCLSimpleAction inv:
    self.repeatDelay->notEmpty() implies self.repeat->notEmpty()


-------------------------------------------------------------------------------------------------------
-- A simpleAction can define the qualifier attribute when the max attribute is defined and is bigger than 1
context NCLSimpleAction inv:
    self.qualifier <> NCLActionOperator::NULL implies (self.max.value > 1 or self.max.value = -1)


-------------------------------------------------------------------------------------------------------
-- A simpleCondition must define a standard role or the eventType and transition attributes
context NCLSimpleCondition inv:
    if self.role->notEmpty() then
        self.role.cname <> NCLDefaultConditionRole::NULL or (self.eventType <> NCLEventType::NULL and self.transition
             <> NCLEventTransition::NULL)
    else
        true
    endif
```

```



-----------------------------------------------------------------------------------------------------
-- A simpleCondition can define the key attribute when the condition is a selection condition
context NCLSimpleCondition inv:
    if self.role.cname <> NCLDefaultConditionRole::ONSELECTION and self.eventType <> NCLEventType::SELECTION then
        self.key = NCLKey::NULL and self.parKey->isEmpty()
    else
        true
    endif



-----------------------------------------------------------------------------------------------------
-- A simpleCondition can define the qualifier attribute when the max attribute is defined and is bigger than 1
context NCLSimpleCondition inv:
    self.qualifier <> NCLConditionOperator::NULL implies
     if i.max->notEmpty() then
        (i.max.value > 1 or i.max.value = -1)
    else
        false
    endif



-----------------------------------------------------------------------------------------------------
-- The import element must have the type NCL when it represents an importNCL and BASE when it represents an importBase
context NCLImport inv:
    if self.parentImportBase->notEmpty() then
        self.type = NCLImportType::NCL
    else
        self.type = NCLImportType::BASE
    endif



-----------------------------------------------------------------------------------------------------
-- A region must not define three equivalent attributes at the same time
context NCLRegion inv:
    not (self.left->notEmpty() and self.right->notEmpty() and self.width->notEmpty())
    and
    not (self.top->notEmpty() and self.bottom->notEmpty() and self.height->notEmpty())



-----------------------------------------------------------------------------------------------------
-- A transition can define the fadeColor attribute when its type is fade with color
context NCLTransition inv:
    if self.type = NCLTransitionType::FADE and (self.subType = NCLTransitionSubtype::FADE_FROM_COLOR or self.subType =
          NCLTransitionSubtype::FADE_TO_COLOR) then
        self.fadeColor <> NCLColor::NULL
    else
        self.fadeColor = NCLColor::NULL
    endif



-----------------------------------------------------------------------------------------------------
-- A transition must define its subtype attribute accordint to its type attribute
context NCLTransition inv:
    (self.type = NCLTransitionType::BAR and (self.subType = NCLTransitionSubtype::LEFT_TO_RIGHT or self.subType =
          NCLTransitionSubtype::TOP_TO_BOTTOM))
     or
    (self.type = NCLTransitionType::IRIS and (self.subType = NCLTransitionSubtype::RECTANGLE or self.subType =
          NCLTransitionSubtype::DIAMOND))
     or
    (self.type = NCLTransitionType::CLOCK and (self.subType = NCLTransitionSubtype::CLOCKWISE_TWELVE or self.subType =
          NCLTransitionSubtype::CLOCKWISE_THREE or self.subType = NCLTransitionSubtype::CLOCKWISE_SIX or self.subType =
           NCLTransitionSubtype::CLOCKWISE_NINE))
     or
    (self.type = NCLTransitionType::SNAKE and (self.subType = NCLTransitionSubtype::TOP_LEFT_HORIZONTAL or self.subType
           = NCLTransitionSubtype::TOP_LEFT_VERTICAL or self.subType = NCLTransitionSubtype::TOP_LEFT_DIAGONAL or self.
          subType = NCLTransitionSubtype::TOP_RIGHT_DIAGONAL or self.subType = NCLTransitionSubtype::
          BOTTOM_RIGHT_DIAGONAL or self.subType = NCLTransitionSubtype::BOTTOM_LEFT_DIAGONAL))
     or
    (self.type = NCLTransitionType::FADE and (self.subType = NCLTransitionSubtype::CROSSFADE or self.subType =
          NCLTransitionSubtype::FADE_TO_COLOR or self.subType = NCLTransitionSubtype::FADE_FROM_COLOR))

```

```

-----------------------------------------------------------------------------------------
-- An area can define the position attribute when the text attribute is defined
context NCLArea inv:
    self.position->notEmpty() implies self.text->notEmpty()


-----------------------------------------------------------------------------------------
-- An area can define the coords attribute when it represents a spatial anchor
context NCLArea inv:
    if self.parentMedia->notEmpty() then
        self.coords->notEmpty() implies (self.parentMedia.mediaType = MediaType::IMAGE or self.parentMedia.mediaType =
             MediaType::VIDEO)
    else
        true
    endif


-----------------------------------------------------------------------------------------
-- An area can define the begin or end attributes when it is a temporal anchor
context NCLArea inv:
    if self.parentMedia->notEmpty() then
        (self.begin->notEmpty() or self.end->notEmpty()) implies (self.parentMedia.mediaType = MediaType::AUDIO or self
             .parentMedia.mediaType = MediaType::VIDEO)
    else
        true
    endif


-----------------------------------------------------------------------------------------
-- An area can define the first or last attributes when it is a temporal anchor
context NCLArea inv:
    if self.parentMedia->notEmpty() then
        (self.first->notEmpty() or self.last->notEmpty()) implies (self.parentMedia.mediaType = MediaType::AUDIO or
             self.parentMedia.mediaType = MediaType::VIDEO)
    else
        true
    endif


-----------------------------------------------------------------------------------------
-- An area can define the text or position attributes when it is a textual anchor
context NCLArea inv:
    if self.parentMedia->notEmpty() then
        (self.text->notEmpty() or self.position->notEmpty()) implies self.parentMedia.mediaType = MediaType::TEXT
    else
        true
    endif


-----------------------------------------------------------------------------------------
-- An area can define the label attribute when it is a procedural anchor
context NCLArea inv:
    if self.parentMedia->notEmpty() then
        self.label->notEmpty() implies self.parentMedia.mediaType = MediaType::PROCEDURAL
    else
        true
    endif


-----------------------------------------------------------------------------------------
-- An area can not have more than one type
context NCLArea inv:
    (self.coords->notEmpty() implies (self.begin->isEmpty() and self.end->isEmpty() and self.first->isEmpty() and self.
         last->isEmpty() and self.text->isEmpty() and self.position->isEmpty() and self.label->isEmpty()))
    and
    ((self.begin->notEmpty() or self.end->notEmpty()) implies (self.coords->isEmpty() and self.first->isEmpty() and
         self.last->isEmpty() and self.text->isEmpty() and self.position->isEmpty() and self.label->isEmpty()))
    and
    ((self.first->notEmpty() or self.last->notEmpty()) implies (self.coords->isEmpty() and self.begin->isEmpty() and
         self.end->isEmpty() and self.text->isEmpty() and self.position->isEmpty() and self.label->isEmpty()))
    and
    ((self.text->notEmpty() or self.position->notEmpty()) implies (self.coords->isEmpty() and self.begin->isEmpty() and
        self.end->isEmpty() and self.first->isEmpty() and self.last->isEmpty() and self.label->isEmpty()))
    and
    (self.label->notEmpty() implies (self.coords->isEmpty() and self.begin->isEmpty() and self.end->isEmpty() and self.
        first->isEmpty() and self.last->isEmpty() and self.text->isEmpty() and self.position->isEmpty()))


-------------------------------------------------------------------------------------
-- An area can not define the end attribute smaller that the begin attribute
context NCLArea inv:
    if self.end->notEmpty() and self.begin->notEmpty() then
        self.begin.getTime().lessThan(self.end.getTime())
    else
        true
    endif

-------------------------------------------------------------------------------------
-- An area can not define the first and last attribute with different types
context NCLArea inv:
    if self.last->notEmpty() and self.first->notEmpty() then
        self.last.type = self.first.type
    endif


-------------------------------------------------------------------------------------
-- An area can not define the last attribute smaller that the first attribute
context NCLArea inv:
    if self.last->notEmpty() and self.first->notEmpty() then
        self.last.value.value > self.first.value.value
    else
        true
    endif


-------------------------------------------------------------------------------------
-- A media can define the instance attribute when its refer attribute is defined
context NCLMedia inv:
    self.instance <> NCLInstanceType::NULL implies self.refer->notEmpty()


-------------------------------------------------------------------------------------
-- The param element must have the type LINKPARAM when it represents a linkParam and BINDPARAM when it represents a
    bindParam
context NCLParam inv:
    if self.parenLink->notEmpty() then
        self.paramType = NCLParamInstance::LINKPARAM
    else
        self.paramType = NCLParamInstance::BINDPARAM
    endif
```

Listing A.3: Reference OCL invariants

```
-------------------------------------------------------------------------------------
-- A rule element must refer to a settings node property through its var attribute
context NCLRule inv:
    if self.var->forAll(v | v.parentNCLMedia->notEmpty()) then
        self.var->forAll(v | v.parentNCLMedia->forAll(p | p.type = NCLMimeType::APPLICATION_X_GINGA_SETTINGS))
    else
        true
    endif


-------------------------------------------------------------------------------------
-- A parameter of a link or bind must refer to a parameter of the connector used by the link
context NCLParam inv:
    if self.parenLink->notEmpty() then
        self.parentLink->forAll(l | l.xconnector->forAll(c | c.conn_params->exists(a | a = self.name)))
    else
        self.parentBind.parentLink.xconnector.conn_params->exists(self.name)
        self.parentBind->forAll(b | b.parentLink->forAll(l | l.xconnector->forAll(c | c.conn_params->exists(a | a =
            self.name))))
    endif

```

```

--------------------------------------------------------------------------------
-- A link bind must refer to a role defined by the connector used by the link
context NCLLink inv:
    self.binds.role->forAll(r | self.xconnector.getRolesFromCausalConnector()->includes(r))
```

Listing A.4: Compositionality OCL invariants

```
--------------------------------------------------------------------------------
-- A bindRule constituent attribute must refer to a descriptor inside the descriptorSwitch
context NCLDescriptorBindRule inv:
    if self.parentDescriptorSwitch->notEmpty() then
        self.parentDescriptorSwitch->forAll(q | q.descriptors->exists(a | a = self.constituent))
    else
        true
    endif


--------------------------------------------------------------------------------
-- A switch defaultDescriptor must refer to a descriptor inside the descriptorSwitch
context NCLDescriptorSwitch inv:
    self.descriptors->exists(a | a = self.defaultDescriptor)


--------------------------------------------------------------------------------
-- A port component attribute must refer to a node inside the port parent context or body
context NCLPort inv:
    if self.parentBody->notEmpty() then
        self.parentBody.nodes->exists(a | a = self.component)
    else
        self.parentContext.nodes->exists(a | a = self.component)
    endif


--------------------------------------------------------------------------------
-- A port interface attribute must refer to a interface point of its referred node
context NCLPort inv:
    if self.interface->notEmpty() then
        if self.component.oclIsKindOf(NCLMedia) then
            self.component->forAll(c | c.oclAsType(NCLMedia).areas->exists(a | a = self.interface))
            or
            self.component->forAll(c | c.oclAsType(NCLMedia).properties->exists(a | a = self.interface))
        else if self.component.oclIsKindOf(NCLContext) then
            self.component->forAll(c | c.oclAsType(NCLContext).ports->exists(a | a = self.interface))
            or
            self.component->forAll(c | c.oclAsType(NCLContext).properties->exists(a | a = self.interface))
        else
            self.component->forAll(c | c.oclAsType(NCLSwitch).ports->exists(a | a = self.interface))
        endif endif
    else
        true
    endif


--------------------------------------------------------------------------------
-- A mapping component attribute must refer to a node inside it parent switch
context NCLMapping inv:
    if self.parentNCLSwitchPort->notEmpty() then
        if self.parentNCLSwitchPort->forAll(p | p.parentNCLSwitch->notEmpty()) then
            self.parentNCLSwitchPort->forAll(p | p.parentNCLSwitch->forAll(s | s.nodes->exists(a | a = self.component))
                )
        else
            true
        endif
    else
        true
    endif


--------------------------------------------------------------------------------
-- A mapping interface attribute must refer to a interface point of its referred node
context NCLMapping inv:
    if self.interface->notEmpty() then
        if self.component.oclIsKindOf(NCLMedia) then
                self.component.areas->exists(self.interface) or self.component.properties->exists(self.interface)
        else if self.component.oclIsKindOf(NCLContext) then
                self.component.ports->exists(self.interface) or self.component.properties->exists(self.interface)
        else
            self.component.ports->exists(self.interface)
        endif endif
    else
        true
    endif

    if self.interface->notEmpty() then
        if self.component.oclIsKindOf(NCLMedia) then
            self.component->forAll(c | c.oclAsType(NCLMedia).areas->exists(a | a = self.interface))
            or
            self.component->forAll(c | c.oclAsType(NCLMedia).properties->exists(a | a = self.interface))
        else if self.component.oclIsKindOf(NCLContext) then
            self.component->forAll(c | c.oclAsType(NCLContext).ports->exists(a | a = self.interface))
            or
            self.component->forAll(c | c.oclAsType(NCLContext).properties->exists(a | a = self.interface))
        else
            self.component->forAll(c | c.oclAsType(NCLSwitch).ports->exists(a | a = self.interface))
        endif endif
    else
        true
    endif


----------------------------------------------------------------------------------------------------
-- A switch defaultComponent must refer to a node inside the switch
context NCLSwitch inv:
    if self.defaultComponent->notEmpty() then
        self.nodes->exists(a | a = self.defaultComponent)
    else
        true
    endif


----------------------------------------------------------------------------------------------------
-- A bindRule constituent attribute must refer to a node inside the switch
context NCLSwitchBindRule inv:
    if self.parentSwitch->notEmpty() then
        self.parentSwitch->forAll(s | s.nodes->exists(a | a = self.constituent))
    else
        true
    endif


----------------------------------------------------------------------------------------------------
-- A bind component attribute must refer to a node inside the link parent context or body
context NCLBind inv:
    if self.parentLink->notEmpty() then
        if self.parentLink.parentBody->notEmpty() then
            self.parentLink.parentBody.nodes->exists(a | a = self.component)
        else
            self.parentLink.parentContext.nodes->exists(a | a = self.component)
        endif
    else
        true
    endif


----------------------------------------------------------------------------------------------------
-- A bind interface attribute must refer to a interface point of its referred node
context NCLBind inv:
    if self.interface->notEmpty() then
        if self.component.oclIsKindOf(NCLMedia) then
                self.component->forAll(c | c.oclAsType(NCLMedia).areas->exists(a | a = self.interface)) or self.
                    component->forAll(c | c.oclAsType(NCLMedia).properties->exists(a | a = self.interface))
        else if self.component.oclIsKindOf(NCLContext) then
                self.component->forAll(c | c.oclAsType(NCLContext).ports->exists(a | a = self.interface)) or self.
                    component->forAll(c | c.oclAsType(NCLContext).properties->exists(a | a = self.interface))
        else
            self.component->forAll(c | c.oclAsType(NCLSwitch).ports->exists(a | a = self.interface))
        endif
        endif
    else
        true
    endif
```

Listing A.5: Composition nesting OCL invariants

```
-----------------------------------------------------------------------------------------
-- A context element can not nest itself
context NCLContext inv:
    self.nodes->forAll(n:NCLNode | n.loops())


-----------------------------------------------------------------------------------------
-- A switch element can not nest itself
context NCLSwitch inv:
    self.nodes->forAll(n:NCLNode | n.loops())
```

Listing A.6: Element reuse OCL invariants

```
-----------------------------------------------------------------------------------------
-- A context element can not refer itself
context NCLContext inv:
    self.refer->forAll(c:NCLContext | c.oclAsType(NCLNode).referLoops())


-----------------------------------------------------------------------------------------
-- A media element can not refer itself
context NCLMedia inv:
    self.refer->forAll(m:NCLMedia | m.oclAsType(NCLNode).referLoops())


-----------------------------------------------------------------------------------------
-- A switch element can not refer itself
context NCLSwitch inv:
    self.refer->forAll(s:NCLSwitch | s.oclAsType(NCLNode).referLoops())
```

Listing A.7: Functions used by OCL invariants

```
-----------------------------------------------------------------------------------------
-- Function to test if an integer is smaller than another
context Integer::lessThan(i:Integer):Boolean body:
    if self.value < i.value then
        true
    else
        false
    endif


-----------------------------------------------------------------------------------------
-- Function to test if a time ir prior than another
context NCLTime::beforeThan(t:NCLTime):Boolean body:
    if self.year.value = t.year.value then
        if self.month.value = t.month.value then
            if self.day.value = t.day.value then
                if self.hour.value = t.hour.value then
                    if self.minute.value = t.minute.value then
                        if self.second.value = t.second.value then
                            if self.fraction.value = t.fraction.value then
                                false
                            else
                                self.fraction.lessThan(t.fraction)
                            endif
                        else
                            self.second.lessThan(t.second)
                        endif
                    else
                        self.minute.lessThan(t.minute)
                    endif
                else
                    self.hour.lessThan(t.hour)
                endif
            else
                self.day.lessThan(t.day)
            endif
        else
            self.month.lessThan(t.month)
        endif
    else
        self.year.lessThan(t.year)
    endif


------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
-- Functions used in nesting loop verification
context NCLNode::loops():Boolean body:
    if self.hasParent() then
        self.oclAsType(NCLElement).getParent().verifyLoop(self)
    else
        true
    endif


context NCLElement::verifyLoop(node:NCLElement):Boolean body:
    if self = node then
        false
    else
        if self.hasParent() then
            self.getParent().verifyLoop(self)
        else
            true
        endif
    endif


------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------
-- Functions used in refer loop verification
context NCLNode::referLoops():Boolean body:
    if self.oclAsType(NCLElement).hasRefer() then
        self.oclAsType(NCLElement).getRefer().verifyReferLoop(self)
    else
        true
    endif


context NCLElement::verifyReferLoop(node:NCLElement):Boolean body:
    if self = node then
        false
    else
        if self.oclAsType(NCLElement).hasRefer() then
            self.oclAsType(NCLElement).getRefer().verifyLoop(self)
        else
            true
        endif
    endif



------------------------------------------------------------------------------------------
-- Function to get an element parent
context NCLElement::getParent():NCLElement body:
    if self.oclAsType(NCLNode).parentContext->notEmpty() then
        self.oclAsType(NCLNode).parentContext
    else
        if self.oclAsType(NCLNode).parentSwitch->notEmpty() then
            self.oclAsType(NCLNode).parentSwitch
        else
            self.oclAsType(NCLNode).parentBody
        endif
    endif


------------------------------------------------------------------------------------------------
-- Function to test if an element has a parent
context NCLElement::hasParent():Boolean body:
    if self.oclAsType(NCLNode).parentContext->notEmpty() then
        true
    else
        if self.oclAsType(NCLNode).parentSwitch->notEmpty() then
            true
        else
            if self.oclAsType(NCLNode).parentBody->notEmpty() then
                true
            else
                false
            endif
        endif
    endif


------------------------------------------------------------------------------------------------
-- Function to get the element referred by a node refer attribute
context NCLElement::getRefer():NCLElement body:
    if self.oclIsTypeOf(NCLMedia) then
        self.oclAsType(NCLMedia).refer
    else
        if self.oclIsTypeOf(NCLContext) then
            self.oclAsType(NCLContext).refer
        else
            self.oclAsType(NCLSwitch).refer
        endif
    endif


------------------------------------------------------------------------------------------------
-- Function to test if a node refer attribute is defined
context NCLElement::hasRefer():Boolean body:
    if self.oclIsTypeOf(NCLMedia) then
        self.oclAsType(NCLMedia).refer->notEmpty()
    else
        if self.oclIsTypeOf(NCLContext) then
            self.oclAsType(NCLContext).refer->notEmpty()
        else
            self.oclAsType(NCLSwitch).refer->notEmpty()
        endif
    endif


------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------
-- Functions to get a region location on the screen
context NCLRegion::getX():Integer body:
    if self.left->notEmpty() then
        self.left.value
    else
        self.right.value - self.width.value
    endif


context NCLRegion::getY():Integer body:
    if self.top->notEmpty() then
        self.top.value
    else
        self.bottom.value - self.height.value
    endif


context NCLRegion::getW():Integer body:
    if self.width->notEmpty() then
        self.width.value
    else
```

```
          self.right.value − self.left.value
     endif


context NCLRegion::getH():Integer body:
     if self.height−>notEmpty() then
          self.height.value
     else
          self.bottom.value − self.top.value
     endif


−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
−− Function to test if two regions collide
context NCLRegion::collide(r:NCLRegion):Boolean body:
     if (self.getX() > (r.getX() + r.getW())) or (self.getY() > (r.getY() + r.getH())) or
     ((self.getX() + self.getW) < r.getX()) or ((self.getY() + self.getH()) < r.getY()) then
          false
     else
          true
     endif


−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
−− Function to get a role min cardinality
context NCLRole::getMin():Integer body:
     if self.parentAction−>notEmpty() then
          if self.parentAction.min−>notEmpty() then
                  self.parentAction.min.value
          else
                  1
          endif
     else
          if self.parentCondition−>notEmpty() then
                  if self.parentCondition.min−>notEmpty() then
                          self.parentCondition.min.value
                  else
                          1
                  endif
          else
                  1
          endif
     endif


−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
−− Function to get a role max cardinality
context NCLRole::getMax():Integer body:
     if self.parentAction−>notEmpty() then
          if self.parentAction.max−>notEmpty() then
                  self.parentAction.max.value
          else
                  −1
          endif
     else
          if self.parentCondition−>notEmpty() then
                  if self.parentCondition.max−>notEmpty() then
                          self.parentCondition.max.value
                  else
                          −1
                  endif
          else
                  1
          endif
     endif


−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
−− Function to get all medias in the document that overlaps
context NCLMedia::getAllOverlappedMedia():Set(NCLMedia) body:
     NCLMedia.allInstances()−>select(media | media.descriptor−>notEmpty())−>select(media | self.descriptor.getAllRegions
          ()−>exists(r | media.descriptor.getAllRegions()−>exists(r2 | r2.collide(r))))

```

```

------------------------------------------------------------------------
-- Function to get all regions referred by a descriptor or descriptorSwitch
context NCLLayoutDescriptor::getAllRegions():Set(NCLRegion) body:
    if self.oclIsTypeOf(NCLDescriptorSwitch) then
        self.oclAsType(NCLDescriptorSwitch).descriptors->collect(desc | desc.getAllRegions())->flatten()->asSet()
    else
        NCLRegion.allInstances()->asOrderedSet()->first().emptySet()->including(self.oclAsType(NCLDescriptor).region)
    endif


------------------------------------------------------------------------
-- Function to get all roles defined inside a connector
context NCLCausalConnector::getRolesFromCausalConnector():Set(NCLRole) body:
    self.action.getRolesFromAction()->union(self.condition.getRolesFromCondition())


------------------------------------------------------------------------
-- Function to get the role defined by an action or compoundAction
context NCLAction::getRolesFromAction():Set(NCLRole) body:
    if self.oclIsTypeOf(NCLCompoundAction) then
        self.oclAsType(NCLCompoundAction).actions->collect(ac | ac.getRolesFromAction())->flatten()->asSet()
    else
        self.oclAsType(NCLSimpleAction).role->select(r | true)->asSet()
    endif


------------------------------------------------------------------------
-- Function to get the role defined by a condition or compoundCondition
context NCLCondition::getRolesFromCondition():Set(NCLRole) body:
    if self.oclIsTypeOf(NCLCompoundCondition) then
         self.oclAsType(NCLCompoundCondition).conditions->collect(ac | ac.getRolesFromCondition())->flatten()->asSet()
             ->union(self.oclAsType(NCLCompoundCondition).statements->collect(st | st.getRolesFromStatement())->
             flatten()->asSet())
    else
        self.oclAsType(NCLSimpleCondition).role->select(r | true)->asSet()
    endif


------------------------------------------------------------------------
-- Function to get the role defined by an assessmentStatement or compoundStatement
context NCLStatement::getRolesFromStatement():Set(NCLRole) body:
    if self.oclIsTypeOf(NCLCompoundStatement) then
        self.oclAsType(NCLCompoundStatement).statements->collect(st | st.getRolesFromStatement())->flatten()->asSet()
    else
        self.oclAsType(NCLAssessmentStatement).attributeAssessments->collect(as | as.role->select(r | true))->flatten()
            ->asSet()
    endif


------------------------------------------------------------------------
-- Function to test if two roles are different
context NCLRole::differ(role:NCLRole):Boolean body:
if self.name->notEmpty() and role.name->notEmpty() then
    self.name.value <> role.name.value
else
    if (self.cname <> NCLDefaultConditionRole::NULL) and (role.cname <> NCLDefaultConditionRole::NULL) then
        self.cname <> role.cname
    else
        if (self.aname <> NCLDefaultActionRole::NULL) and (role.aname <> NCLDefaultActionRole::NULL) then
            self.aname <> role.aname
        else
            true
        endif
    endif
endif


------------------------------------------------------------------------
-- Function to get the link binds that use the same role
context NCLLink::getBindsFromRole(role:NCLRole):Set(NCLBind) body:
    self.binds->select(b | b.role = role)

```

```

------------------------------------------------------------------------------------------------
-- Function to get all parameters inside a link
context NCLLink::getParameters():Set(NCLParam) body:
    self.binds->collect(b | b.bindParams)->flatten()->asSet()->union(self.linkParams)
```

References

[ABNT 2007]ABNT. *Digital terrestrial television - Data coding and transmission specification for digital broadcasting - Part 2: Ginga-NCL for fixed and mobile receivers - XML application language for application coding.* 2007. ABNT NBR 15606-2:2007 standard.

[ABNT 2011]ABNT. *Digital terrestrial television - Data coding and transmission specification for digital broadcasting - Part 2: Ginga-NCL for fixed and mobile receivers - XML application language for application coding.* 2011. ABNT NBR 15606-2:2011 standard.

[Adobe Systems 2010]Adobe Systems. *ActionScript references and documentation.* 2010. http://www.adobe.com/devnet/actionscript/references.html.

[Allen 1983]ALLEN, J. F. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, ACM, v. 26, n. 11, p. 832–843, 1983.

[Apple Inc. 2010]Apple Inc. *iMovie 11.* 2010. http://www.apple.com/ilife/imovie/.

[Araújo et al. 2008]ARAÚJO, E. C.; AZEVEDO, R. G. A.; NETO, C. S. S. NCL-validator: um processo para validação sintática e semântica de documentos multimídia NCL . In: *Jornada de Informática do Maranhão.* [S.l.: s.n.], 2008. In portuguese.

[Azevedo et al. 2009]AZEVEDO, R. G. A.; TEIXEIRA, M. M.; NETO, C. S. S. NCL Eclipse: Ambiente Integrado para o Desenvolvimento de Aplicações para TV Digital Interativa em Nested Context Language. In: *Salão de ferramentas - Simpósio Brasileiro de Redes Computadores 2009.* [S.l.]: SBRC, 2009. In portuguese.

[Bertino et al. 2005]BERTINO, E.; FERRARI, E.; PEREGO, A.; SANTI, D. A Constraint-Based Approach for the Authoring of Multi-Topic Multimedia Presentations. In: *IEEE International Conference on Multimedia and Expo.* [S.l.]: IEEE Computer Society, 2005. p. 578–581.

[Blakowski and Steinmetz 1996]BLAKOWSKI, G.; STEINMETZ, R. A Media Synchronization Survey: Reference Model, Specification and Case Studies. *Journal on Selected Areas in Communications*, IEEE, v. 14, n. 1, p. 5–35, January 1996.

[Boll 2001]BOLL, S. *ZYX - Towards flexible multimedia document models for reuse and adaptation.* Ph.D. Thesis Vienna University of Technology, 2001.

[Bossi and Gaggi 2007]BOSSI, A.; GAGGI, O. Enriching SMIL with assertions for temporal validation. In: *Proceedings of the 15th International Conference on Multimedia.* [S.l.]: ACM, 2007. p. 107–116.

[Braga et al. 2011]BRAGA, C. O.; MENEZES, R. W.; COMICIO, T.; SANTOS, C.; LANDIM, E. Transformation Contracts in Practice. *IET Software*, 2011.

[Buchanan and Zellweger 2005]BUCHANAN, M. C.; ZELLWEGER, P. T. Automatic Temporal Layout Mechanisms Revisited. *ACM Transactions on Multimedia Computing, Communications and Applications (TOMCCAP)*, ACM, v. 1, n. 1, p. 60–88, February 2005.

[Bulterman and Hardman 2005]BULTERMAN, D. C. A.; HARDMAN, L. Structured multimedia authoring. *ACM Transactions on Multimedia Computing, Communications and Applications (TOMCCAP)*, ACM, v. 1, n. 1, p. 89–109, February 2005.

[Clarke et al. 2000]CLARKE, E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking.* [S.l.]: The MIT Press, 2000.

[Clavel et al. 2007]CLAVEL, M.; EKER, S.; DURÁN, F.; LINCOLN, P.; MARTÍ-OLIET, N.; MESEGUER, J. *All about Maude - A High-performance Logical Framework: how to Specify, Program, and Verify Systems in Rewriting Logic.* [S.l.]: Springer-Verlag New York Inc, 2007.

[Díaz et al. 2001]DÍAZ, P.; AEDO, I.; PANETSOS, F. Modeling the Dynamic Behavior of Hypermedia Applications. *IEEE Transactions on Software Engineering*, IEEE, v. 27, n. 6, p. 550–572, June 2001.

[Eidenberger 2003]EIDENBERGER, H. SMIL and SVG in teaching. In: *Proceedings of Internet Imaging V.* [S.l.]: SPIE, 2003. p. 69–80.

[Elias et al. 2006]ELIAS, S.; EASWARAKUMAR, K.; CHBEIR, R. Dynamic consistency checking for temporal and spatial relations in multimedia presentations. In: *Proceedings of the 2006 ACM symposium on Applied computing.* [S.l.]: ACM, 2006. p. 1380–1384.

[Felix 2004]FELIX, M. F. *Formal Analysis of Software Models Oriented by Architectural Abstractions.* D.Sc. Thesis Pontifícia Universidade Católica do Rio de Janeiro, 2004. In Portuguese.

[Furuta and Stotts 2001]FURUTA, R.; STOTTS, P. D. Trellis: a Formally-defined Hypertextual Basis for Integrating Task and Information. *Coordination Theory and Collaboration Technology*, Lawrence Erlbaum Associates, p. 341–367, 2001.

[Honorato and Barbosa 2010]HONORATO, G. S. C.; BARBOSA, S. D. J. NCL-Inspector: Towards Improving NCL Code. In: *Proceedings of the 2010 ACM Symposium on Applied Computing.* [S.l.]: ACM, 2010. p. 1946–1947.

[ISO/IEC 2005]ISO/IEC. *Information technology - Coding of audio-visual objects - Part 11: Scene description and application engine.* 2005. ISO/IEC 14496-11:2005.

[ITU 1997]ITU. *Audiovisual and multimedia systems.* 1997. http://www.itu.int/ITU-T/studygroups/com16/index.asp. Study Group 16 - Multimedia Coding, Systems and Applications.

[ITU 2009]ITU. *Nested Context Language (NCL) and Ginga-NCL for IPTV services.* 2009. http://www.itu.int/rec/T-REC-H.761-200904-S. ITU-T Recommendation H.761.

[Jansen and Bulterman 2009]JANSEN, J.; BULTERMAN, D. C. A. SMIL State: an architecture and implementation for adaptative time-based web applications. *Multimedia Tools and Applications*, Springer, v. 43, n. 3, p. 203–224, 2009.

[Jourdan et al. 1998]JOURDAN, M.; LAYAIDA, N.; ROISIN, C.; SABRY-ISMAIL, L.; TARDIF, L. Madeus, an authoring environment for interactive multimedia documents. In: *Proceedings of the 6th ACM International Conference on Multimedia*. [S.l.]: ACM, 1998. p. 267–272.

[Lima et al. 2010]LIMA, B.; AZEVEDO, R. G. A.; MORENO, M.; SOARES, L. F. G. Composer 3: Ambiente de autoria extensível, adaptável e multiplataforma. In: *Workshop of Interactive Digital TV (WTVDI)*. [S.l.: s.n.], 2010. In portuguese.

[Ma and Shin 2004]MA, H.; SHIN, K. G. Checking consistency in multimedia synchronization constraints. *IEEE Transactions on Multimedia*, v. 6, p. 565–574, Agosto 2004.

[Mellor et al. 2003]MELLOR, S.; CLARK, A.; FUTAGAMI, T. Guest editors' introduction: Model driven development. *IEEE Software*, v. 20, p. 14–18, 2003.

[Muchaluat-Saade and Soares 2002]MUCHALUAT-SAADE, D. C.; SOARES, L. F. G. XConnector & XTemplate: Improving the Expressiveness and Reuse in Web Authoring Languages. *The New Review of Hypermedia and Multimedia Journal*, Taylor Graham, v. 8, n. 1, p. 139–169, 2002.

[Na and Furuta 2001]NA, J.; FURUTA, R. Dynamic documents: authoring, browsing, and analysis using a high-level petri net-based hypermedia system. In: *Proceedings of the 2001 ACM Symposium on Document engineering*. [S.l.]: ACM, 2001. p. 38–47.

[Oliveira et al. 2001]OLIVEIRA, M. de; TURINE, M.; MASIERO, P. A statechart-based model for hypermedia applications. *ACM Transactions on Information Systems (TOIS)*, ACM, v. 19, n. 1, p. 52, 2001.

[OMG 2003]OMG. *MDA Guide Version 1.0.1.* june 2003. http://www.omg.org/docs/omg/03-06-01.pdf. Object Management Group.

[OMG 2010]OMG. *OMG Unified Modeling Language (OMG UML) Infrastructure Version 2.3.* 2010. http://www.omg.org/spec/UML/2.3/ Infrastructure/PDF. Object Management Group.

[Peterson 1981]PETERSON, J. L. *Petri Net Theory and Modeling of systems.* [S.l.]: Prentice-Hall, 1981.

[Pnueli 1977]PNUELI, A. The temporal logic of programs. In: IEEE. *18th Annual Symposium on Foundations of Computer Science.* [S.l.], 1977. p. 46–57. ISSN 0272-5428.

[Pérez-Luque and Little 1996]PÉREZ-LUQUE, M. J.; LITTLE, T. D. C. A Temporal Reference Framework for Multimedia Synchronization. *Journal on Selected Areas in Communications*, IEEE, v. 14, n. 1, p. 36–51, January 1996.

[Rodrigues et al. 2002]RODRIGUES, L.; ANTONACCI, M. J.; RODRIGUES, R. F.; MUCHALUAT-SAADE, D. C.; SOARES, L. F. G. Improving SMIL with NCM Facilities. *Multimedia Tools and Applications*, Springer, v. 16, n. 1, p. 29–54, 2002.

[Santos et al. 1998]SANTOS, C.; SOARES, L. F. G.; SOUZA, G. L. de; COURTIAT, J. P. Design methodology and formal validation of hypermedia documents. In: *Proceedings of the sixth ACM International Conference on Multimedia.* [S.l.]: ACM, 1998. p. 39–48.

[Santos and Muchaluat-Saade 2011]SANTOS, J. A. F. dos; MUCHALUAT-SAADE, D. C. XTemplate 3.0: spatio-temporal semantics and structure reuse for hypermedia compositions. *Multimedia Tools and Applications*, Springer, 2011. published online - http://www.springerlink.com/content/m3932258853567j0/.

[Soares and Rodrigues 2005]SOARES, L. F. G.; RODRIGUES, R. F. *Nested Context Model 3.0 Part 1 - NCM Core*. Rio de Janeiro, May 2005.

[Soares et al. 2000]SOARES, L. F. G.; RODRIGUES, R. F.; MUCHALUAT-SAADE, D. C. Modeling, authoring and formatting hypermedia documents in the HyperProp system. *Multimedia Systems*, Springer-Verlag, 2000.

[W3C 1999]W3C. *XML Stylesheet Language Transformations (XSLT) Version 1.0.* 1999. http://www.w3.org/TR/xslt. World-Wide Web Consortium Recommendation.

[W3C 2000]W3C. *Document Object Model (DOM) Level 2 Core Specification.* 2000. World-Wide Web Consortium Recommendation DOM-Level-2-Core-20001113.

[W3C 2008a]W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* 2008. http://www.w3.org/TR/2008/REC-xml-20081126/. World-Wide Web Consortium Recommendation.

[W3C 2008b]W3C. *Synchronized Multimedia Integration Language - SMIL 3.0 Specification.* 2008. http://www.w3c.org/TR/SMIL3. World-Wide Web Consortium Recommendation.

[W3C 2011]W3C. *HTML5: A vocabulary and associated APIs for HTML and XHTML.* 2011. http://www.w3.org/TR/html5/. World-Wide Web Consortium Working Draft.

[Wahl and Rothermel 1994]WAHL, T.; ROTHERMEL, K. Representing Time in Multimedia Systems. In: *Proceedings of International Conference on Multimedia Computing and Systems*. [S.l.]: IEEE Computer Society Press, 1994.

[Warmer and Kleppe 1999]WARMER, J.; KLEPPE, A. *The Object Constraint Language.* [S.l.]: Addison–Wesley, 1999.

[Willrich et al. 2001]WILLRICH, R.; SAQUI-SANNES, P. de; SÉNAC, P.; DIAZ, M. Design and management of multimedia information systems. In: RAHMAN, S. M. (Ed.). [S.l.]: IGI Publishing, 2001. cap. HTSPN: an experience in formal modeling of multimedia applications coded in MHEG or Java, p. 380–411.