University of Passau
Faculty for Mathematics and Informatics

Diploma Thesis

Model Checking Temporal Description Logics

Christian Schönberg
Duvenhorst 14
D-26127 Oldenburg
mail@cschoenberg.de

1st Corrector:	Prof. Dr. Burkhard Freitag
2nd Corrector:	Prof. Dr.-Ing. Gregor Snelting

Tutor:	Dipl-Inf. Franz Weitl
	Prof. Dr. Burkhard Freitag
	Chair for Information Management
	Faculty for Mathematics and Informatics
	University of Passau

Abstract

This Diploma Thesis will extend the Model Checking capabilities of CTL to the temporal description logic $\mathcal{ALC}\mathsf{CTL}$, and use it to verify properties of web documents.
The main goal is to implement two different approaches: To reduce an $\mathcal{ALC}\mathsf{CTL}$ model and $\mathcal{ALC}\mathsf{CTL}$ formulas to CTL so that the model checking tools for CTL can be used, and to re-implement the CTL model checking algorithm for $\mathcal{ALC}\mathsf{CTL}$. The performance and usability of each approach will be measured and analysed in the course of this work.

Contents

Chapter 1

Introduction

This Diploma Thesis seeks solutions to the model checking problem for temporal description logics. In the course of this work I will develop such solutions for the temporal description logic $\mathcal{ALC}\mathsf{CTL}$.

The temporal description logic $\mathcal{ALC}\mathsf{CTL}$ has been developed from the temporal logic CTL and the description logic $\mathcal{ALC}$ by Prof. Dr. Burkhard Freitag and Dipl.-Inf. Franz Weitl [WF04a]. The goal was to combine the expression power of description logics with that of branching-time temporal logics and to harness it for representing and verifying semantic properties of web documents, while still keeping the language decidable.

The goal of this Diploma Thesis is to implement the Model Checking problem for $\mathcal{ALC}\mathsf{CTL}$. Model checking is the process of verifying properties against a model. In this case, I will only regard certain types of models, namely web documents used for web-based training (WBT) or E-Learning. The primary source for such documents will be WWR[1] documents, coded in *Lmml*[2].

The checking of web documents, particularly WBTs, has been one of the primary use-cases since the development of $\mathcal{ALC}\mathsf{CTL}$ and its predecessors begun at the University of Passau [WF04c]. Due to their complex structure and potential large size, they represent a major challenge that will test the limits of any verification environment. On the other hand, it is also very tedious – at best – to verify structural properties of web documents manually. An automated approach would make life easier, and might possibly even improve the overall quality of such documents. This promises to be especially useful for E-Learning documents that inherently have many important and verifiable properties and require a high degree of correctness at the same

[1] *Wissenswerkstatt Rechensysteme* (Knowledge Factory for Computer Systems), http://www.wwr-project.de/

[2] Learning Material Markup Language, http://www.lmml.de

time.

A part of this correctness can already be achieved by defining a syntactical corset to which the WBT source documents can be made to conform to. In terms of XML, this might be a document type definition (DTD) or an XML schema definition (XSD). Checking a document against such a definition is a way to ensure correctness on a very basic level: Its main use is to enforce a valid structure and some restrictions on the content. On the same level, automatic checking for e.g. invalid references and the like can be performed. But to validate a document on a semantic level, e.g. to ensure that there is an example available for every definition, more complex techniques are required. $\mathcal{ALC}$CTL model checking is one such technique.

Why use a temporal description logic for web documents in the first place? Because the narrative structure of such documents can be easily represented as a branching-time temporal model! Put yourself in the place of the reader/user of such a document: You start reading at the beginning, follow the general structure, branching off at times, following links forwards and backwards, thus skipping or repeating sections, and you finally reach an end point. What you did was to follow one of many (possibly infinite) available paths through the document. Using a temporal model, it is now possible to represent *all* of those paths and to reason about them. On the other hand, the description logics (DL) part of the temporal description logic is very useful when it comes to actually expressing properties about specific pages, such as referencing definitions or examples. In short, description logics can describe the pages of a web document, while temporal logics can describe the connections between them [WF04a, WF04b, WF04c].

In order to be able to verify properties against a model, three things are needed: First, a representation of the model. Second, a representation of the properties (usually in the form of a formula). And third, a method to check the formula against the model. I will describe how an $\mathcal{ALC}$CTL model can be implemented, how such a model can be extracted from an E-Learning source, how an $\mathcal{ALC}$CTL formula can be modelled, and, mainly, how model checking can be done with such a model and such a formula. In the course of this work I will develop two different approaches to solve this primary problem.

One such approach is to reduce the entire $\mathcal{ALC}$CTL structure to CTL and use the existing model checking tools for CTL. The main problem here is that the reduction scales exponentially to the interpretation domain Δ^{I}.
The other possibility is to implement the model checking algorithm for $\mathcal{ALC}$CTL. The CTL variant does not allow for sets or for roles. This has to be taken into account when implementing the formula structure and the algorithm itself for $\mathcal{ALC}$CTL. Another problem is finding a useful counter example in case the formula does not hold against the model: The user will

probably want to know why it does not hold.

Literature research has shown that there are no alternate solutions for this particular problem, against which the solutions of this thesis could be measured. There are, however, various CTL model checking approaches. Most are based on ordered binary decision diagrams [McM93, CCGR00], but there are also methods using the satisfiability problem SAT [BCCZ99, CCGR+02]. Using model checking on web documents has been proposed by e.g. [SFC98]. Nonetheless, using model checking on a combination of description logics and temporal logics to verify properties of web documents has not been implemented before. It will be interesting to see whether one (or possibly both) of the proposed approaches to model checking $\mathcal{ALC}$CTL can be used for problems from the real world.

Chapter 2

Basics

First, I would like to cover the technical basics. I will define the terms Description Logics and Temporal Logics. Then I will describe the syntax and use of the language CTL. I will proceed to explain what model checking is, and how it is done with CTL. After that, I will show how $\mathcal{ALC}$CTL and CTL are related, and where $\mathcal{ALC}$CTL surpasses its ancestor. Finally, I will provide a brief overview about the web document definition languages *Lmml* and $<ML^3>$.

Verification in general requires a model of the system – in this case, of the web documents – and a specification of the properties that are to be verified against that model. CTL offers a formal way to define a model, and it allows for the definition of properties.
Both CTL and $\mathcal{ALC}$CTL have separate specifications for models and properties. Properties are defined in the way of formulas, with $\mathcal{ALC}$CTL formulas being more powerful than CTL formulas.

2.1 Description Logics and Temporal Logics

Definition 1 (Description Logics) *An elementary Description Logic (DL) is a fragment of a predicate logic without functions, with only unary and binary operators, and with at most two different variables. While the DL itself does not have variables, its syntax can be mapped to that of the predicate logic.*
A DL-system contains a list of terminology (usually referred to as a "TBox") and a list of assertions (usually referred to as an "ABox"). The ABox-assertions use the contents of the TBox.
The ABox consists of both atomic concepts and atomic roles. Atomic concepts refer to sets of objects, while atomic roles specify binary relations be-

tween objects. Both are denominated by names, concepts conventionally by nouns (written with a capital first letter), and roles conventionally by verbs (written in all lower case).
Valid DL concepts are $\top$, $\bot$, A, $\neg A$, $C_1 \sqcap C_2$, $\forall R.C$, $\exists R.\top$, where A is an atomic concept, R is an atomic role, and C is a DL concept. For a definition of the semantics, see definition 9 in chapter 2.4.
The interpretation function I assigns each concept a subset of the interpretation domain Δ^I, while it assigns each atomic role a subset of the binary relation $\Delta^I \times \Delta^I$. Since roles are usually bound by a quantor, the general signature of I is $C^I \subseteq \Delta^I$, where C is a concept [Fre03].

Definition 2 (Temporal Logics) *A Temporal Logic models time as a sequence of states. There are two distinctions to be made about the view of time. First, whether it is linear or branching, and second, whether it is discrete or continuous. Branching time allows different futures, or alternate time lines. The temporal logics that will be used in this Diploma Thesis handles time as branching and discrete.*
Temporal logic employs the usual Boolean operators, but introduces some additional modal (or temporal) operators like X, F, G or U (see definitions 7 and 9 in chapter 2.4 for details and the semantics) [HR00].

2.2 CTL

CTL is a discrete branching-time temporal logic. It employs the usual Boolean operators, as well as some temporal operators.

Valid CTL formulas ϕ are $\top$, $\bot$ and p, where p is any valid predicate. Valid formulas are also the negation of a formula ($\neg\phi$), as well as the conjunction, the disjunction and the implication between two formulas ($\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\phi_1 \rightarrow \phi_2$). Those are the standard Boolean operators; more interesting are the following temporal operators.

In order to understand temporal operators, it is advisable to regard the temporal structure of CTL first. CTL uses a discrete temporal model that is often visualised by a state-transition diagram. The states represent different points in time, while the transitions represent possible branchings of the time line, thus modelling different possible futures for each state. Predicates are annotated to each state where they are valid.

As an example, regard a web document that has an introduction, a section defining a Deterministic Finite Automaton (DFA), a section stating an example of a DFA, and a conclusion. The reader can skip the example after the definition and proceed directly to the conclusion. This document can be

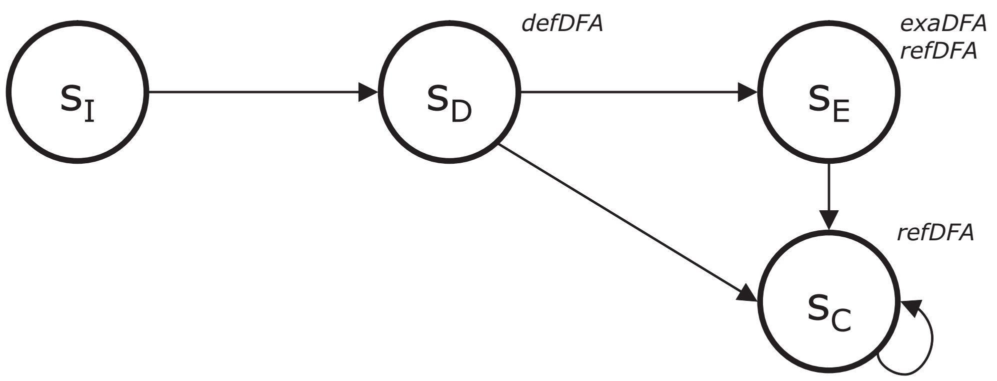


Figure 2.1: A CTL example model with four states.

represented as a temporal model, where the different sections are points in time, and the structure consists of the possible time-lines (or the state transitions) [SFC98]. Thus, the document can be modelled as a system with the states s_{Intro}, s_{Def}, s_{Exa} and s_{Conc}, with transitions between s_{Intro} and s_{Def}, s_{Def} and s_{Exa}, s_{Def} and s_{Conc}, and s_{Exa} and s_{Conc}. Since in a CTL model every state needs to have at least one successor, there is also a transition between s_{Conc} and s_{Conc} itself. The predicate $defDFA$, which indicates that a DFA is defined in the current state, is annotated to the state s_{Def}, while the predicate $exaDFA$ is annotated to the state s_{Exa}. Figure 2.1 illustrates this model.

Temporal operators in CTL always consist of two letters: First an A or an E, followed by a second one. The A can be described as a temporal all-quantor: It means that for a formula to be true, it must be true for *all* possible futures. In our example, regarding state s_{Def}, a formula would have to hold in both s_{Exa} and s_{Conc} for A to hold. E on the other hand resembles a temporal existence-quantor: The formula has to hold for at least one possible future, either s_{Exa} or s_{Conc} in the example above.

For the second letter, there are several possibilities. The first is the letter X, which stands for *next*. It means that a formula only has to hold in exactly the next state, and nothing is said about any other states. If it has to hold in only one next state or in all of them (since there are branching time lines, a state may have more than one immediate successor) depends on the first letter (A or E).
Another possible letter is F, for future, which means that the formula must hold in some future state - not necessarily the next one. The combination EF could thus be interpreted as “at some time”, it is the most unspecified operator.

The letter G (globally) is more general: It requires the formula to hold in every single following state - whether on all or just one time line depends as usual on the first letter. AG can be interpreted as "all the time".
The second last possibility is the letter U: The until-operator. It expects two arguments $E\,[\phi_1\,U\,\phi_2]$ or $A\,[\phi_1\,U\,\phi_2]$ and means that ϕ_1 has to hold until – at some point – ϕ_2 holds.
Finally, B stands for the before-operator. It is binary as well and means that ϕ_1 has to hold at least once before ϕ_2 does [HR00].

Let's have a look at some examples. A formula that says "There has to be an example of a DFA somewhere" could be written in CTL as $EF\,exaDFA$. A formula that says "There must be an example of a DFA in the next state (e.g. paragraph)" can be written as $EX\,exaDFA$. However, it will only be valid in state s_{Def}, since no other state has a direct successor with an example in it. A bit more complicated would be a formula expressing "Anywhere there must a DFA be defined either in the current or in the next state, until there is an example of one": $A\,[(defDFA \,\vee\, EX\,defDFA)\;U\,exaDFA]$.

Definition 3 (CTL Formula) *A CTL formula ϕ has the following form:*

$$\begin{aligned}\phi ::= \;& \bot \,|\, \top \,|\, p \,|\, \neg\phi \,|\, \phi \wedge \phi \,|\, \phi \vee \phi \,|\, \phi \rightarrow \phi \,|\\ & AX\phi \,|\, EX\phi \,|\, A[\phi\,U\,\phi] \,|\, E[\phi\,U\,\phi] \,|\, AG\phi \,|\, EG\phi \,|\, AF\phi \,|\, EF\phi \,|\\ & A[\phi\,B\,\phi] \,|\, E[\phi\,B\,\phi],\end{aligned}$$

where p is an atomic description.
Note that the operators $A[\phi_1\,B\,\phi_2]$ and $E[\phi_1\,B\,\phi_2]$ are usually not included in the formal definition of CTL formulas: They are abbreviations for $\neg E[\neg\phi_1\,U\,\phi_2]$ and $\neg A[\neg\phi_1\,U\,\phi_2]$, respectively [HR00, WF04a].

Definition 4 (CTL Model) *A CTL model M consists of three parts.*
First, a set of states S.
Second, a binary state-transition relation $\longrightarrow$ that specifies at least one successor for every state. Formally: $\forall s_i \in S : \exists s_j \in S : s_i \longrightarrow s_j$. This implies that there are paths with an infinite length, possibly even an infinite number of them.
Third, a labelling function L that annotates all predicates to a state s that are valid in that state. Formally: $\forall s_i \in S : L(s_i) \subseteq AtomicDescriptions$.
Thus, $M = (S, \longrightarrow, L)$.
Note: Since an infinite number of objects cannot be represented with a computer, I will only regard finite sets S.

Definition 5 (Semantics of CTL Operators) *Let $M = (S, \longrightarrow, L)$ be some CTL model, and $s \in S$. The question if a formula ϕ holds in s can be written as $M, s \models \phi$. $\models$ is defined by structural induction over all CTL formulas in table 2.1 for a model $M = (S, \longrightarrow, L)$, $s \in S$, where p is an atomic description [HR00].*

$\top$:	$M, s \models \top$
$\bot$:	$\neg(M, s \models \bot)$
p :	$M, s \models p$ iff $p \in L(s)$
$\neg\phi$:	$M, s \models \neg\phi$ iff $\neg(M, s \models \phi)$
$\phi_1 \wedge \phi_2$:	$M, s \models \phi_1 \wedge \phi_2$ iff $M, s \models \phi_1$ and $M, s \models \phi_2$
$\phi_1 \vee \phi_2$:	$M, s \models \phi_1 \vee \phi_2$ iff $M, s \models \phi_1$ or $M, s \models \phi_2$
$\phi_1 \rightarrow \phi_2$:	$M, s \models \phi_1 \rightarrow \phi_2$ iff $\neg(M, s \models \phi_1)$ or $M, s \models \phi_2$
$AX\phi$:	$M, s \models AX\phi$ iff $\forall s_1$, with $s \longrightarrow s_1 : M, s_1 \models \phi$
$EX\phi$:	$M, s \models EX\phi$ iff $\exists s_1$, with $s \longrightarrow s_1 : M, s_1 \models \phi$
$AG\phi$:	$M, s \models AG\phi$ iff for all s_i along all paths $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s : M, s_i \models \phi$
$EG\phi$:	$M, s \models EG\phi$ iff for all s_i along some path $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s : M, s_i \models \phi$
$AF\phi$:	$M, s \models AF\phi$ iff on all paths $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi$
$EF\phi$:	$M, s \models EF\phi$ iff on some path $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi$
$A[\phi_1\, U\, \phi_2]$:	$M, s \models A[\phi_1\, U\, \phi_2]$ iff on all paths $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi_2$, and $\forall j < i : M, s_j \models \phi_1$
$E[\phi_1\, U\, \phi_2]$:	$M, s \models E[\phi_1\, U\, \phi_2]$ iff on some path $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi_2$, and $\forall j < i : M, s_j \models \phi_1$

Table 2.1: Semantics of CTL operators ϕ

2.3 CTL Model Checking

Now that we know what a CTL model looks like and how a CTL formula is specified, we can advance to the actual model checking technique. Model checking is the process of verifying a formula against a model. There is a standard algorithm for CTL that accomplishes this task. The principle of this algorithm is fairly simple: You start with the innermost part of the formula and annotate it to all the states where it is valid. Now you work your way outward, annotating ever more complex sub-formulas, until you have annotated the whole formula. Finally, all you have to do is check to which states you annotated the whole formula – those are the ones where the formula holds.

Let's regard our example (figure 2.1) from the last section again, and check the formula $A\,[(defDFA \;\vee\; EX\,defDFA)\;U\,exaDFA]$ against it. The two predicates $defDFA$ and $exaDFA$ are already annotated, so we can skip them, and take the next outward step.
The right-hand side of the U is already done, but there is still some work on the left-hand side: $defDFA \;\vee\; EX\,defDFA$. $defDFA$ is behind us now, so we start with the innermost sub-formula that is one step up from that: $EX\,defDFA$. We know all the states where $defDFA$ is annotated (namely, s_{Def}), so it's easy to find all states that are immediate predecessors to those states: s_{Intro}. Therefore, we annotate the sub-formula $EX\,defDFA$ to the state s_{Intro} (see figure 2.2).
Now we can regard the entire part $defDFA \;\vee\; EX\,defDFA$. Again, we know where its sub-formulas are valid (we just finished with the last one, $EX\,defDFA$), so all we have to do is to find all states where either the left sub-formula is true, or the right one. In s_{Intro}, $defDFA$ is not annotated (which means that the predicate does not hold there). However, since it is annotated to its successor s_{Def}, $EX\,defDFA$ is annotated in s_{Intro}. Therefore we can annotate the disjunction to both s_{Intro} and s_{Def}. It can easily be seen that it does not hold in the two other states (see figure 2.3).
The next step is to annotate the whole formula: We have to find all states where $A\,[sf_1\,U\,sf_2]$ holds, with sf_1 and sf_2 being the two sub-formulas we already checked. A brief look at the definition of the AU operator will reveal that the formula is indeed valid in all states but the last one, s_{Conc}. That is to be expected, since there are no more states after it where the until-condition could be satisfied (see figure 2.4).

Definition 6 (CTL Model Checking) CTL *model checking is the process of deciding whether* $M, s \models \phi$*, for some model* $M = (S, \longrightarrow, L)$*, a state* $s \in S$*, and some* CTL *formula* ϕ*. The selection of* s *is often limited to any*

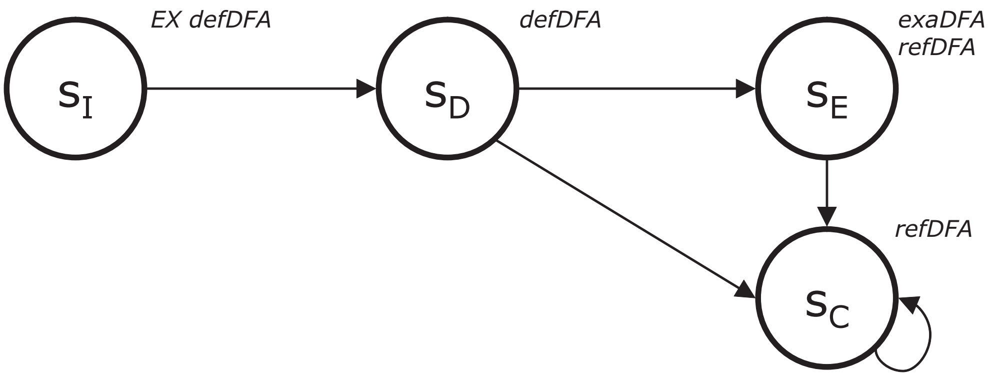


Figure 2.2: CTL model checking (1).

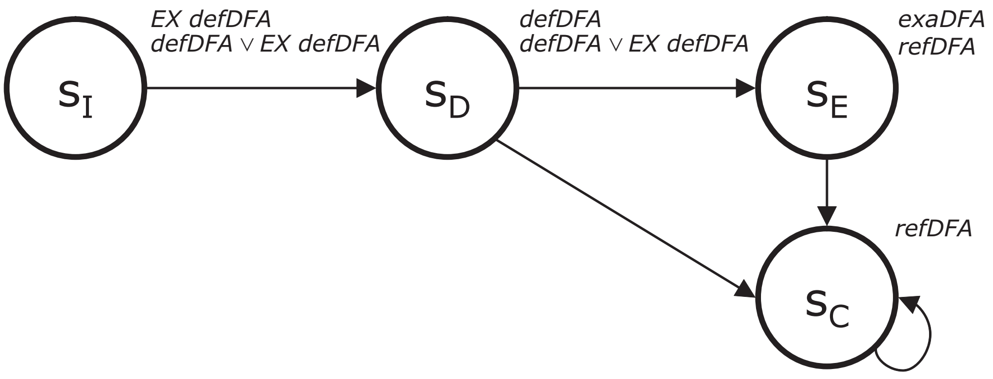


Figure 2.3: CTL model checking (2).

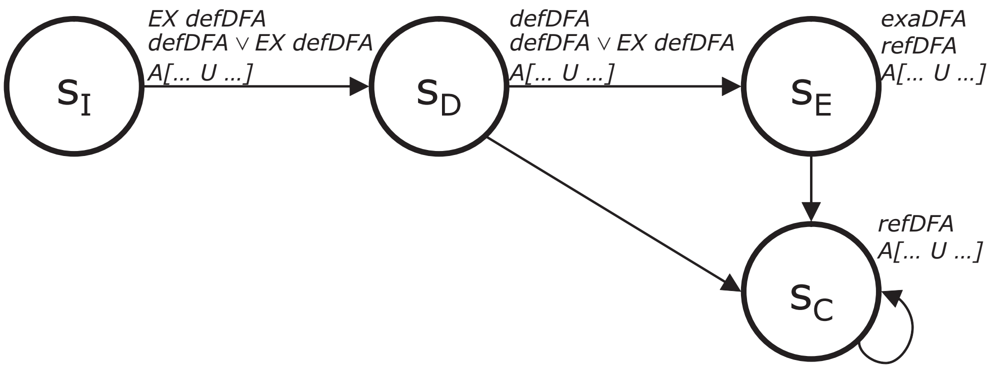


Figure 2.4: CTL model checking (3).

s_0*, where* s_0 *is a starting state of* M.

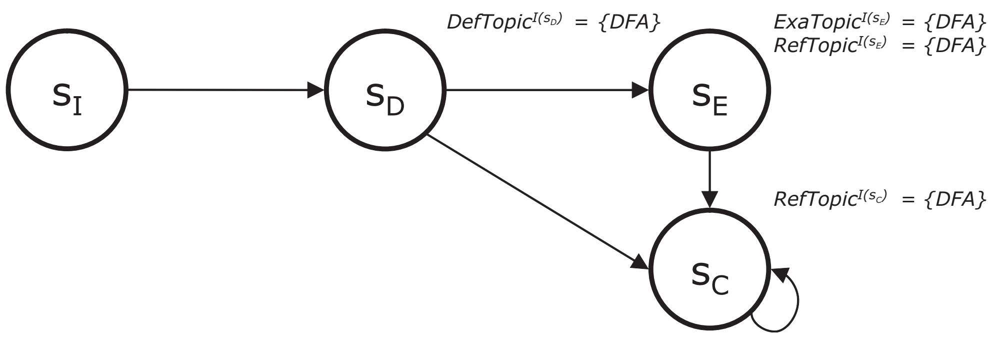
2.4 $\mathcal{ALC}$CTL

So much for CTL. $\mathcal{ALC}$CTL is a bit more complex than that. It is a combination of CTL and description logics. Before I define a formula, I would like to introduce a few basics about $\mathcal{ALC}$CTL. $\mathcal{ALC}$CTL deals primarily with sets of objects instead of predicates. The potential elements of such a set are pooled in the interpretation domain Δ^I. For an E-Learning document about automata theory, Δ^I might contain the objects that are subjects of the document: DFA ("Deterministic Finite Automaton"), NFA ("Non-Deterministic Finite Automaton") and TM ("Turing Machine"). But it will probably also contain objects that describe the different paragraphs of the text, e.g. $Chapter1Section2Paragraph3$ or $ParagraphExampleForNFA$. Scaling parameters (cf. chapter 2.5) can also be part of Δ^I, for example $TargetGroupStudent$, $TargetGroupTeacher$ or $IntensityAdvanced$. Informally, Δ^I can be regarded as the base vocabulary of a model.

The next important thing about $\mathcal{ALC}$CTL is the concept. It can have a name (like "$TopicOfDefinition$", or shorter "$DefTopic$") and is interpreted, using an interpretation function $I(s)$. Each concept is interpreted at every state of a model, returning a set of objects (a subset of Δ^I). Concept "$DefTopic$" interpreted at state s_{Def} might yield the set $\{DFA, NFA\}$ as a result, formally: $DefTopic^{I(s_{Def})} = \{DFA, NFA\}$.

A concept is always the inner part of an $\mathcal{ALC}$CTL formula. As such, it is called a "state concept". A basic state concept can be either $\top$, $\bot$ or an atomic concept. $\top$ is the equivalent of the tautology for sets: It is always interpreted as the entire interpretation domain Δ^I. $\bot$ is correspondingly always interpreted as the empty set. An atomic concept is one that is referenced by name (such as "$DefTopic$"), and that does not need recursive interpretation.

There are, however, other state concepts that operate on concepts. Since concepts return – when interpreted – sets, it is not surprising that the union, intersect and complement operators are among them: $\psi_1 \sqcup \psi_2$, $\psi_1 \sqcap \psi_2$ and $\neg\psi$ for state concepts ψ. In addition, the temporal operators that we already know from CTL are state concepts as well (with adjusted semantics): $AX\psi$, $EX\psi$, $A[\psi_1 U \psi_2]$, $EF\psi$ and so on. The remaining two state concepts originate in description logics rather than temporal logics. They are the role concepts $\forall R.\psi$ and $\exists R.\psi$, with R being an atomic role. In our example (see above) a role might look like this: $\exists\, topicOf.Paragraph$, while its interpretation could return {"*Example for an NFA*"}.



$DefTopic^{I(s_D)} = \{DFA\}$
$ExaTopic^{I(s_E)} = \{DFA\}$
$RefTopic^{I(s_E)} = \{DFA\}$
$RefTopic^{I(s_C)} = \{DFA\}$
s_I
s_D
s_E
s_C

Figure 2.5: An $\mathcal{ALC}$CTL model with several interpreted concepts.

A full $\mathcal{ALC}$CTL formula is built on top of such concepts. A formula has a Boolean value, as it does in CTL. Therefore, valid formulas ϕ include $true$, $false$, $\neg\phi$, $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$ and $\phi_1 \rightarrow \phi_2$. The temporal operators are defined for formulas as well. The connection to concepts is made by the subset and equals operators: $\psi_1 \dot{\sqsubseteq} \psi_2$ and $\psi_1 \doteq \psi_2$ are both valid formulas.

Here are a few more examples of $\mathcal{ALC}$CTL formulas:

$$\bot \dot{\sqsubseteq} DefTopic \tag{2.1}$$

$$DefTopic \doteq ExampleTopic \tag{2.2}$$

$$DefTopic \dot{\sqsubseteq} EX\,(ExampleTopic) \tag{2.3}$$

$$DefTopic \dot{\sqsubseteq} EF\,(ExampleTopic) \tag{2.4}$$

$$AG\left(DefTopic \dot{\sqsubseteq} EX\,(ExampleTopic \sqcup ParaTopic)\right) \tag{2.5}$$

$$\neg E\left[true\; U\; \neg\left(DefTopic \dot{\sqsubseteq} EX\,(ExampleTopic \sqcup ParaTopic)\right)\right] \tag{2.6}$$

Formula 2.1 is a tautology, since the empty set (to which $\bot$ yields) is subset of every set. Formula 2.2 demands that at every state, where some topic is defined, there must be an example of it as well, and vice versa ($\doteq$). If no topic is defined, no example is required, but if more than one topic is defined, there must be an example for every one. The same is true for zero or more than one example: There must be the same number of definitions. Formula 2.3 is similar, except that the example must come in one of the successor states, and that there may be examples of topics that have not been defined (at least not in the state before). Formula 2.4 is less stringent, it allows the example to be placed anywhere after the definition (EF instead of EX). Formula 2.5 requires that directly after each definition there must be an example of it, or its topic must at least be discussed some more ($ParaTopic$ – topic of the paragraph). This must be true in every state for the formula to be true (AG). The last formula, formula 2.6, is equivalent to formula 2.5. The AG operator has been replaced with a $\neg E\,[true\ U\ \neg \ldots]$ construct. This expresses semantically exactly the same, but with a different syntax.

Definition 7 ($\mathcal{ALC}$CTL Formula) *An $\mathcal{ALC}$CTL formula ϕ has the following form:*

$\phi ::=\ true \mid false \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid$
$AX\phi \mid EX\phi \mid A[\phi\, U\, \phi] \mid E[\phi\, U\, \phi] \mid AG\phi \mid EG\phi \mid AF\phi \mid EF\phi \mid$
$A[\phi\, B\, \phi] \mid E[\phi\, B\, \phi] \mid \psi_1 \dot{\sqsubseteq} \psi 2 \mid \psi_1 \doteq \psi_2$*, where ψ is a state concept.*

A state concept ψ has the following form:

$\psi ::=\ \top \mid \bot \mid C \mid \neg\psi \mid \psi_1 \sqcap \psi_2 \mid \psi_1 \sqcup \psi_2 \mid$
$AX\psi \mid EX\psi \mid A[\psi\, U\, \psi] \mid E[\psi\, U\, \psi] \mid AG\psi \mid EG\psi \mid AF\psi \mid EF\psi \mid$
$A[\psi\, B\, \psi] \mid E[\psi\, B\, \psi] \mid \forall R.\psi \mid \exists R.\psi \mid$*, where C is an atomic concept and R is an atomic role.*

Note that again the operators $A[\psi_1\, B\, \psi_2]$ and $E[\psi_1\, B\, \psi_2]$ are usually not included in the formal definition of $\mathcal{ALC}$CTL formulas: They are abbreviations for $\neg E[\neg\psi_1\, U\, \psi_2]$ and $\neg A[\neg\psi_1\, U\, \psi_2]$, respectively [WF04a].

Definition 8 ($\mathcal{ALC}$CTL Model) *An $\mathcal{ALC}$CTL model M consists of four parts.*
First, a set of states S.
Second, a binary state-transition relation $\longrightarrow$ that specifies at least one successor for every state. Formally: $\forall s_i \in S : \exists s_j \in S : s_i \longrightarrow s_j$. This implies that there are paths with an infinite length, possibly even an infinite number of them. $\longrightarrow$ is also often denominated as R.
Third, an interpretation function I that interprets each concept locally at any $s \in S$ and yields a subset of the interpretation domain Δ^I in return. Formally: $\forall s_i \in S : C^{I(s_i)} \subseteq \Delta^I$, where C is a state concept. Δ^I as the interpretation domain is a state independant set of objects.
When interpreting a role R on its own, the interpretation is specified as $R^{I(s)} \subseteq \Delta^I \times \Delta^I$. This is only of indirect relevance for the state concepts, however, since there it is always combined with either a $\forall$ or an $\exists$, thus binding one of the two Δ^I.
Fourth, the interpretation domain Δ^I itself.
Thus, $M = (S, \longrightarrow, I, \Delta^I)$ [WF04a].
Note: Since an infinite number of objects cannot be represented with a computer, I will only regard finite sets S and Δ^I.

Definition 9 (Semantics of $\mathcal{ALC}$CTL Operators) *Let $M = (S, \longrightarrow, I, \Delta^I)$ be some $\mathcal{ALC}$CTL model, and $s \in S$. The question if a formula ϕ holds in s can be written as $M, s \models \phi$. $\models$ is defined by structural induction over all $\mathcal{ALC}$CTL formulas in table 2.2 for a model M, $s \in S$,*

$true$:	$M, s \models true$
$false$:	$\neg(M, s \models false)$
$\neg\phi$:	$M, s \models \neg\phi$ iff $\neg(M, s \models \phi)$
$\psi_1 \dot{\sqsubseteq} \psi_2$:	$M, s \models \psi_1 \dot{\sqsubseteq} \psi_2$ iff $\psi_1^{I(s)} \subseteq \psi_2^{I(s)}$
$\psi_1 \doteq \psi_2$:	$M, s \models \psi_1 \doteq \psi_2$ iff $\psi_1^{I(s)} = \psi_2^{I(s)}$
$\phi_1 \wedge \phi_2$:	$M, s \models \phi_1 \wedge \phi_2$ iff $M, s \models \phi_1$ and $M, s \models \phi_2$
$\phi_1 \vee \phi_2$:	$M, s \models \phi_1 \vee \phi_2$ iff $M, s \models \phi_1$ or $M, s \models \phi_2$
$\phi_1 \rightarrow \phi_2$:	$M, s \models \phi_1 \rightarrow \phi_2$ iff $\neg(M, s \models \phi_1)$ or $M, s \models \phi_2$
$AX\phi$:	$M, s \models AX\phi$ iff $\forall s_1$, with $s \longrightarrow s_1 : M, s_1 \models \phi$
$EX\phi$:	$M, s \models EX\phi$ iff $\exists s_1$, with $s \longrightarrow s_1 : M, s_1 \models \phi$
$AG\phi$:	$M, s \models AG\phi$ iff for all s_i along all paths $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s : M, s_i \models \phi$
$EG\phi$:	$M, s \models EG\phi$ iff for all s_i along some path $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s : M, s_i \models \phi$
$AF\phi$:	$M, s \models AF\phi$ iff on all paths $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi$
$EF\phi$:	$M, s \models EF\phi$ iff on some path $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi$
$A[\phi_1 \, U \, \phi_2]$:	$M, s \models A[\phi_1 \, U \, \phi_2]$ iff on all paths $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi_2$, and $\forall j < i : M, s_j \models \phi_1$
$E[\phi_1 \, U \, \phi_2]$:	$M, s \models E[\phi_1 \, U \, \phi_2]$ iff on some path $s_0 \longrightarrow s_1 \longrightarrow s_2$ $\longrightarrow \ldots$, with $s_0 = s$, there is an $s_i : M, s_i \models \phi_2$, and $\forall j < i : M, s_j \models \phi_1$

Table 2.2: Semantics of $\mathcal{ALC}$CTL operators for formulas ϕ

where ψ is a state concept. Table 2.3 defines the semantics for $\mathcal{ALC}$CTL state concepts that are used in table 2.2. Let $M = (S, \longrightarrow, I, \Delta^I)$, $s \in S$ and C be an atomic concept in table 2.3 [WF04a].

Definition 10 ($\mathcal{ALC}$CTL Model Checking) $\mathcal{ALC}$CTL *model checking is the process of deciding whether $M, s \models \phi$, for some model $M = (S, \longrightarrow, I, \Delta^I)$, a state $s \in S$, and some $\mathcal{ALC}$CTL formula ϕ. The selection of s is often limited to any s_0, where s_0 is a starting state of M.*

$\top$:	$\top^{I(s)}$	$=_{def}\ \Delta^{I}$
$\bot$:	$\bot^{I(s)}$	$=_{def}\ \emptyset$
C :	$C^{I(s)}$	$=_{def}\ C^{I(s)}$
$\neg\psi$:	$(\neg\psi)^{I(s)}$	$=_{def}\ \Delta^{I}\backslash\psi^{I(s)}$
$\psi_1 \sqcap \psi_2$:	$(\psi_1 \sqcap \psi_2)^{I(s)}$	$=_{def}\ \psi_1^{I(s)} \cap \psi_2^{I(s)}$
$\psi_1 \sqcup \psi_2$:	$(\psi_1 \sqcup \psi_2)^{I(s)}$	$=_{def}\ \psi_1^{I(s)} \cup \psi_2^{I(s)}$
$A[\psi_1\, U\, \psi_2]$:	$A[\psi_1\, U\, \psi_2]^{I(s)}$	$=_{def}\ \bigcap_{s_k \in S,\, s \longrightarrow \ldots \longrightarrow s_k} \Big\{a \in \Delta^{I} \mid \exists i \in \mathbb{N}_0 \Big[a \in \psi_2^{I(s_i)} \wedge \forall j \in \mathbb{N}_0 \Big(j < i \rightarrow a \in \psi_1^{I(s_j)}\Big)\Big]\Big\}$
$E[\psi_1\, U\, \psi_2]$:	$E[\psi_1\, U\, \psi_2]^{I(s)}$	$=_{def}\ \bigcup_{s_k \in S,\, s \longrightarrow \ldots \longrightarrow s_k} \Big\{a \in \Delta^{I} \mid \exists i \in \mathbb{N}_0 \Big[a \in \psi_2^{I(s_i)} \wedge \forall j \in \mathbb{N}_0 \Big(j < i \rightarrow a \in \psi_1^{I(s_j)}\Big)\Big]\Big\}$
$AX\psi$:	$(AX\psi)^{I(s)}$	$=_{def}\ \bigcap_{s_i \in S,\, s \longrightarrow s_i} \psi^{I(s_i)}$
$EX\psi$:	$(EX\psi)^{I(s)}$	$=_{def}\ \bigcup_{s_i \in S,\, s \longrightarrow s_i} \psi^{I(s_i)}$
$AF\psi$:	$(AF\psi)^{I(s)}$	$=_{def}\ A[\top\, U\, \psi]^{I(s)}$
$EF\psi$:	$(EF\psi)^{I(s)}$	$=_{def}\ E[\top\, U\, \psi]^{I(s)}$
$AG\psi$:	$(AG\psi)^{I(s)}$	$=_{def}\ \neg EF\neg\psi^{I(s)}$
$EG\psi$:	$(EG\psi)^{I(s)}$	$=_{def}\ \neg AF\neg\psi^{I(s)}$
$\forall R.\psi$:	$(\forall R.\psi)^{I(s)}$	$=_{def}\ \big\{a \in \Delta^{I} \mid \forall b.(a,b) \in R^{I(s)} \rightarrow b \in \psi^{I(s)}\big\}$
$\exists R.\psi$:	$(\exists R.\psi)^{I(s)}$	$=_{def}\ \big\{a \in \Delta^{I} \mid \exists b.(a,b) \in R^{I(s)} \wedge b \in \psi^{I(s)}\big\}$

Table 2.3: Semantics of $\mathcal{ALC}$CTL operators for state concepts ψ

2.5 *Lmml* and <ML^3>

Lmml[1] is an XML language that can be used to specify web documents, particularly E-Learning documents. It provides ways to define the structure and the content of such a document. The structure runs along two tracks: On the one hand, it is the usual division into chapters and subsections. On the other hand, it is a division into a number of learning units that are each subdivided into several presentation units. Both structures are usually closely related, but are used for different purposes [Sue05].

The content is written as text, with definitions, examples and so on indicated accordingly. Images and animations can be inserted, as well as web applications. *Lmml* not only supports direct referencing, but also semantic references, i.e. references to a certain concept or learning objective [Sue05].

Yet the most striking and powerful feature of *Lmml* is its scalability. There are four different scaling axes: The intensity (difficulty and extensiveness of the content), the target group (student or teacher), the context form (script or slide variant) and the output medium (screen or paper). All content can be written for any combination of these variant options, allowing for very specialised content, created with a minimum of effort [Sue05].
At the University of Passau, *Lmml* has been widely used to encode E-Learning documents in the context of the WWR-Project[2] [WFG+04].

However, the "official" language of the WWR-Project is <ML^3>[3], an XML dialect that is somewhat less powerful than *Lmml*, but easier to learn for the novice. It offers less scaling options and fewer content distinctions, while still being adequate for encoding web-based training data [KLT+04].

[1]Learning Material Markup Language, http://www.lmml.de
[2]*Wissenswerkstatt Rechensysteme* (Knowledge Factory for Computer Systems), http://www.wwr-project.de/
[3]Multidimensional LearningObjects and Modular Lectures Markup Language, http://www.ml-3.org/

Chapter 3

Task

As mentioned above, the task of this Diploma Thesis is to facilitate model checking for $\mathcal{ALC}$CTL, with a use case in verifying properties of web documents. We now know the basics about $\mathcal{ALC}$CTL, what model checking is, and how to represent a web document as a temporal model. How is it now possible to actually achieve model checking for $\mathcal{ALC}$CTL?
Model checking for CTL is already routinely done, there are several tools for that (e.g. Spin, SMV, NuSMV) [Hol91, CCGR00, McM93]. Those tools are highly optimised, regarding both efficiency and usability. Since it is possible to reduce $\mathcal{ALC}$CTL formulas and models to CTL, it might be possible to capitalise on that. However, since such a transformation usually has its drawbacks, the straight-forward approach of re-implementing the model checking algorithm for $\mathcal{ALC}$CTL should not be dismissed either.

3.1 Possible Approach I: Reduction

In [WF04a] there is a list of CTL equivalences for $\mathcal{ALC}$CTL expressions. Using these "translation templates", it should be possible to reduce $\mathcal{ALC}$CTL formulas. Another matter is the model, where care must be taken to convert the graph-structure of a typical $\mathcal{ALC}$CTL model into the rather flat variant expected by most CTL model checkers.
There is, however, a problem in principle. $\mathcal{ALC}$CTL deals – at least on the concept level – with sets, while CTL recognises only Boolean types. That means that all sets, that is, all interpretations of concepts, must be expanded to a list of Boolean expressions. This expansion must move along the line of all possible elements of each set: Namely, Δ^I. For a concept "definedTopic" and $\Delta^I = \{DFA, NFA, TM\}$, those expressions might be $DFA\_is\_Defined$, $NFA\_is\_Defined$ and $TM\_is\_Defined$. Doing this at every state and for

every concept will enlarge the formula considerably. Obviously, this scales badly: The formula grows exponentially.

3.2 Possible Approach II: Algorithmic

Implementing the model checking algorithm for $\mathcal{ALC}$CTL avoids this problem. On the other hand, in the scope of this Diploma Thesis, it will be impossible to achieve a degree of optimisation even close to that of the model checking tools for CTL. To find out if one effect offsets the other will be one of the questions in my analysis.

Chapter 4

General Concepts and Procedure

Now, how to go about this task? The first part is reading the model, that is, extracting the $\mathcal{ALC}\mathsf{CTL}$ model representation from an XML data source. The second part is reading and processing the $\mathcal{ALC}\mathsf{CTL}$ formulas, while the third part is to actually check a formula against the model.
In theory, the XML source for the model could be any number of things. However, for this particular task, I will only regard E-Learning documents, namely documents encoded in *Lmml* and, to a lesser extend, $<ML^3>$. All XML formats will be converted into a standard intermediary content format which holds the content uniformly as lists of terminology, assertions and states. Each presentation unit of the web document is represented as a different state.

From this intermediary format the final model will be extracted. It consists of a set of states, annotated with interpretations, predicates, roles and a list of successors. It also includes a set of concepts: The interpretation domain Δ^I.

Preprocessing of the XML data files can be done via XSL transformation, while the XML parsing can be handled with the Java 1.5 XML interface JAXP.
The $\mathcal{ALC}\mathsf{CTL}$ formulas can be parsed from simple Strings (or files of Strings). It seems advisable to create a Java implementation of a formula that takes all possible different aspects of $\mathcal{ALC}\mathsf{CTL}$ formulas into account. This leads to a basic formula object with descendants that represent different quantors or logical operators, and a basic concept object with role- and temporal quantors or set operators as inheritors.

4.1 Model Extraction

When extracting the model, the interpretation domain Δ^I and both the atomic concepts and the atomic roles need to be specified. Δ^I will simply be extracted from the local "vocabulary", that is, from definitions, examples, attributes and so on. Among the atomic concepts that can be extracted will be $definedTopic$ and $exemplifiedTopic$ (both of which yield to one or more topics), while possible atomic roles are *scaleTo* (which relates scaling information to fragments), *hasScaling* (which relates a fragment to scaling information), $topicOf$ and *hasTopic* (which relate a topic to a fragment and vice versa), and $definedAt$ and $exemplifiedAt$ (which connect a fragment to a definition or an example). See appendix A.5 for a full listing.

During model extraction, the main operation for the model is adding: States are added to the model, successors and interpretations are added to states, etc. During model checking, the main operation is data retrieval, like iterating over states or interpreting concepts. Those differences in performance requirements should be reflected by the implementation. The semantic model will make use of lists, which can be quickly expanded. The final $\mathcal{ALC}$CTL model will employ hash tables to simplify access.

A major problem in the extraction process is the handling of the various scaling variants (see chapter 2.5). First, considering different variants, it is possible that a model has more than one starting state (e.g. one each for the screen and the slide variants, as seen in the *Lmml* Fuzzy module). Since model checking can easily be done for more than one starting state, that is not really an issue. More of a problem is the question of succession. Can a state that is marked as "advanced" have a successor that is only "basic"? Can a state that is marked as "basic" have a successor that is marked as "basic" *and* "advanced"?
There are two extreme ways to tackle this problem: Integrating all variants as closely as possible vs. separating them as much as possible. Maximum integration guarantees compactness of the model, at the cost of losing most of the benefits that prompted the scaling in the first place, such as target group optimised content. Extreme separation on the other hand would lead to a lot of duplicate data when e.g. all states that are marked as "basic" and "advanced" would be split in two and put into two different branches of the model. In fact, there would have to be a separate branch for every possible combination of variants. For *Lmml* alone, that would be 24 paths.

I have chosen a middle approach that will hopefully combine the advantages of both ways, while negating the disadvantages. All states will be represented only once, so no splitting will occur, saving memory space and processing time. However, the calculation of successor states will follow the

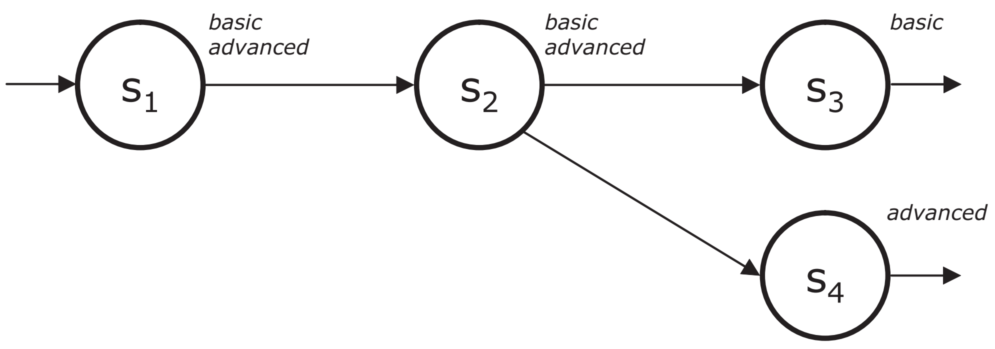


Figure 4.1: Model states and their successors.

concept of maximum separation: A state marked as “basic” and “advanced” will have two immediate successors – the next state that is marked “basic” *and* the next state that is marked “advanced” (see figure 4.1). Additionally, all scaling information will be annotated to the model states with the roles *scaleTo* and *hasScaling*. This way, the structure of the model is preserved, while the variants can still be clearly distinguished.

4.2 Implementation Method

Finally, there is the question of how to go about the implementation itself. The established waterfall model and its derivates come to mind. However, I have selected another approach, namely a variation of the rapid prototyping model. I will create a first prototype as quickly as possible. This prototype will have all the basic functionality of the final version, but not in the same degree. It will support fewer operators, and offer only rudimentary support for *Lmml*, and no support for $<ML^3>$ at all. Implementing and working with this version will soon reveal the most obvious design flaws, with regard to both the object oriented model and to the usability. These flaws will determine the changes that are to be made for the next version. As much of the source code as possible will be re-used, but everything that has to go through major changes will be re-implemented from scratch.
This process will be repeated until the implementation appears sound, and no more errors or design flaws are detectable. In retrospect, this state was achieved after two major reimplementations and a few minor ones.

Why did I choose this somewhat exotic implementation method instead of “doing it by the book”? There are several reasons for this. For one, I felt that the waterfall method was unnecessary cumbersome for a project that

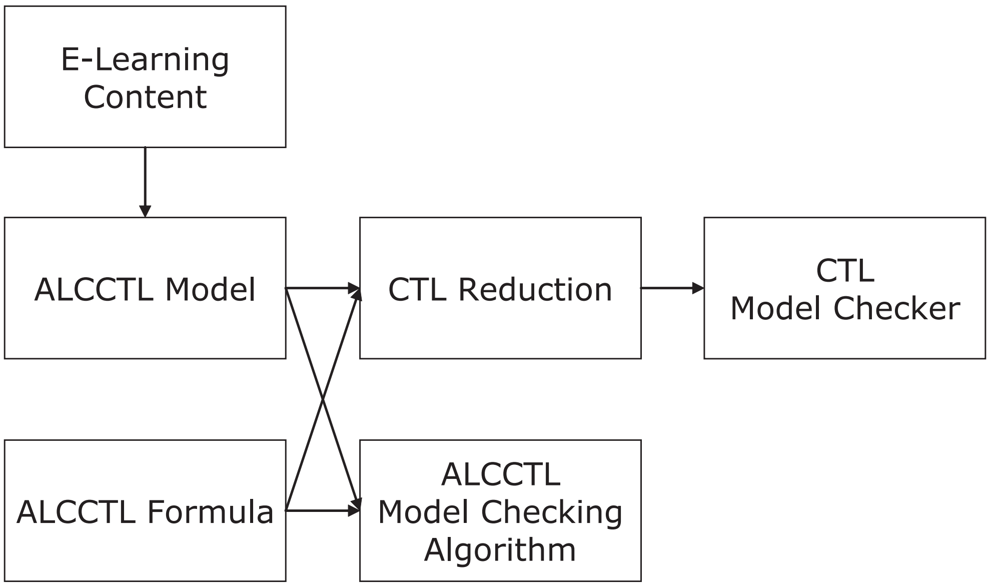


Figure 4.2: Overview of the general procedure of model checking.

needed to respond flexibly to any possible changes (for example, the whole counter example issue from chapter 6.4 was introduced into the project at a rather advanced point). Another reason is that with the rapid-prototyping approach, I was able to devote most of my time to the actual implementation. There were no extensive specifications to update when, for example, at some point I decided to change the base of temporal operators. All the time that would have gone into planning and preparation went directly into problem solving and fine-tuning. The fact that since I started testing the implementation, I have yet to find a single error is silent testimony to that. It is likely, however, that someone with more experience concerning the waterfall method might have spotted most of the problems beforehand. Nonetheless, I still consider the trade-off worthwhile.
Last but not least, I was frankly curious if this approach would work for a project such as this. My previous experience with programming has shown this method to be very effective for small projects. As it appears, it can be easily adapted to suit the needs of a thesis-level project as well. I do not believe that it would be practical for large projects or for teams, though.

4.3 Specification

Figure 4.2 shows a general overview of the procedure of model checking, including both approaches.

Before any attempt at model checking can be made, there has to be a representation of $\mathcal{ALC}$CTL formulas and models, as well as a way to enter them into the system. This representation can then be used for both approaches.

$\mathcal{ALC}$CTL formulas have a tree-like structure: Each formula has one or two subformulas or subconcepts, and each concept is either a leaf (atomic concepts) or has one or two subconcepts. The operator EX for example expects one parameter, while the operators EU and $\dot{\sqsubseteq}$ both expect two parameters. The model checking algorithm makes use of this structure by traversing the tree from bottom to top, and the reduction rules for CTL are defined for tree-like nodes as well. Therefore it makes sense to give the internal representation of a formula a tree structure. Any formula operator can be the root element, followed by a subtree of any number of further formula operators. At the end of each branch there must be either a $\dot{\sqsubseteq}$ or an $\doteq$ operator to make the transition from formula to concept level. Below the $\dot{\sqsubseteq}$ or $\doteq$ operator follows another subtree of concept operators, with atomic concepts at the very end.
Since formulas will be present as text, there has to be a parser that can create such a tree from the linear syntactical representation.

An $\mathcal{ALC}$CTL model is an infinite graph of interconnected states. Each state has a name, and a list of its direct successors. It also needs a way to annotate the interpretations of concepts, as well as roles and formulas. The model itself also has to include the interpretation domain Δ^{I}.
Lmml is a very powerful language that offers the author of web documents many different ways of expression, syntax abbreviations and other things. That makes it rather hard to extract a uniform model from it. To make this a little easier, I have decided to employ XSL[1] preprocessing. The original XML content is transformed using several XSL stylesheets (one at a time) into a more unified form of the same content. Instead of extracting the $\mathcal{ALC}$CTL model now, a generic semantic model can be extracted, which offers additional room for unification. Theoretically, a reasoning system could be connected to the model at this point, generating additional benefits. It is only now, from the generic model, that the actual $\mathcal{ALC}$CTL model is exported.

I have decided to create three main Java packages to pool the different aspects of the implementation. The package `ALCCTL` will contain the formula representation and the model. It has a subpackage `ALCCTL.Parser` that

[1]XML Stylesheet Language

holds the formula parser components. The second main package, `ELearning`, contains the entire model extraction chain, from the XML parser over the XSLT processing to the generic semantical model. The last package contains tools and utilities, as well as debugging capabilities. It is aptly named `Utils`.

4.3.1 Formula

The interface `GeneralFormula` is a very generic formula – it provides methods for model checking against a `GeneralModel` (see chapter 6.2.1), but nothing else. It can be used to represent any formula, including CTL formulas. The base class for an $\mathcal{ALC}$CTL formula is the abstract class `Formula`. It implements `GeneralFormula`, and introduces several methods that are useful for $\mathcal{ALC}$CTL formulas. The `parse` method parses a string representation of a formula and returns the corresponding `GeneralFormula` object. The `toALCCTL`, `toCTL` and `toLaTeX` methods return a string representing the $\mathcal{ALC}$CTL version of the formula, the CTL version of the formula, or a LaTeX formatted version of the formula, respectively. They are abstract in `Formula`, and are implemented by its non-abstract descendants. The `translate` method is also abstract. It returns a version of the formula where all temporal operators have been reduced to three base operators (see chapter 6.1.1). The class `Formula` is extended, among others, by `FormulaTrue`, `FormulaFalse`, `FormulaAnd`, `FormulaAF`, `FormulaAG`, `FormulaEU` and `FormulaSubset`.
For an overview of the formula implementation, see figure 4.3. Figure 4.4 shows an example formula tree of the formula $AG\left(defTopic \,\dot{\sqsubseteq}\, EF\, exaTopic\right)$.

The classes `FormulaSubset` and `FormulaEquals` both make use of the abstract class `Concept`, which represents a basic state concept. It too has the `toALCCTL`, `toCTL` and `toLaTeX` methods, but since it is indigenous to $\mathcal{ALC}$CTL and is not a complete formula, it does not implement the `GeneralFormula` interface. Since the signature of the `toCTL` and `translate` methods differ between formula and concept and since there are only two public methods remaining that are shared between formula and concept, there is no common interface for them.

4.3.2 Model

The interface `GeneralModel` represents a basic state-based model. Even though it includes an interpretation domain, it could still be used for e.g. CTL. It provides accessor methods for the states and for Δ^I, and has a method that allows for the automatic detection of starting states.

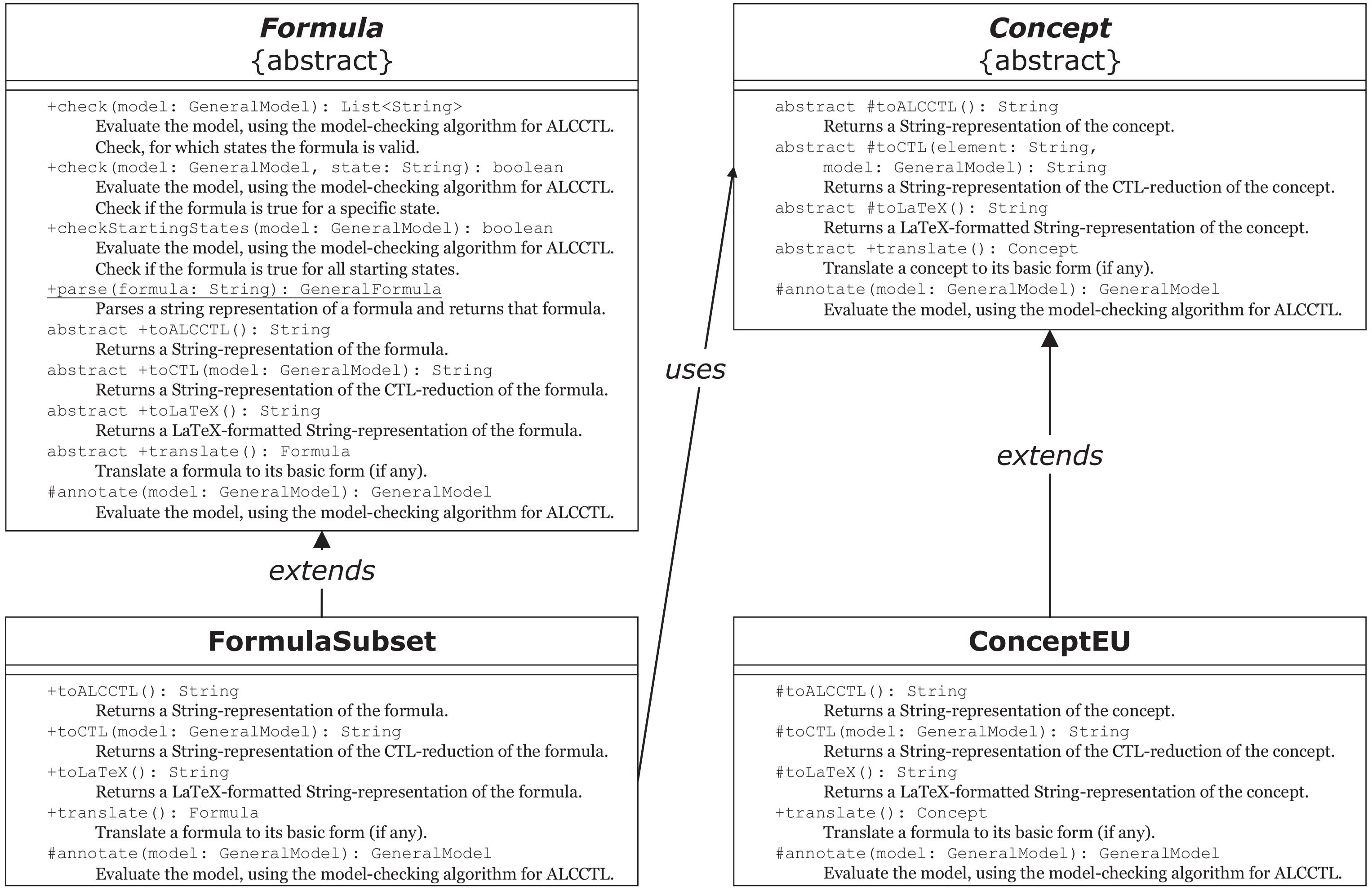

Formula
{abstract}
+check(model: GeneralModel): List<String>
Evaluate the model, using the model-checking algorithm for ALCCTL.
Check, for which states the formula is valid.
+check(model: GeneralModel, state: String): boolean
Evaluate the model, using the model-checking algorithm for ALCCTL.
Check if the formula is true for a specific state.
+checkStartingStates(model: GeneralModel): boolean
Evaluate the model, using the model-checking algorithm for ALCCTL.
Check if the formula is true for all starting states.
+parse(formula: String): GeneralFormula
Parses a string representation of a formula and returns that formula.
abstract +toALCCTL(): String
Returns a String-representation of the formula.
abstract +toCTL(model: GeneralModel): String
Returns a String-representation of the CTL-reduction of the formula.
abstract +toLaTeX(): String
Returns a LaTeX-formatted String-representation of the formula.
abstract +translate(): Formula
Translate a formula to its basic form (if any).
#annotate(model: GeneralModel): GeneralModel
Evaluate the model, using the model-checking algorithm for ALCCTL.
extends
FormulaSubset
+toALCCTL(): String
Returns a String-representation of the formula.
+toCTL(model: GeneralModel): String
Returns a String-representation of the CTL-reduction of the formula.
+toLaTeX(): String
Returns a LaTeX-formatted String-representation of the formula.
+translate(): Formula
Translate a formula to its basic form (if any).
#annotate(model: GeneralModel): GeneralModel
Evaluate the model, using the model-checking algorithm for ALCCTL.
uses
Concept
{abstract}
abstract #toALCCTL(): String
Returns a String-representation of the concept.
abstract #toCTL(element: String,
model: GeneralModel): String
Returns a String-representation of the CTL-reduction of the concept.
abstract #toLaTeX(): String
Returns a LaTeX-formatted String-representation of the concept.
abstract +translate(): Concept
Translate a concept to its basic form (if any).
#annotate(model: GeneralModel): GeneralModel
Evaluate the model, using the model-checking algorithm for ALCCTL.
extends
ConceptEU
#toALCCTL(): String
Returns a String-representation of the concept.
#toCTL(model: GeneralModel): String
Returns a String-representation of the CTL-reduction of the concept.
#toLaTeX(): String
Returns a LaTeX-formatted String-representation of the concept.
+translate(): Concept
Translate a formula to its basic form (if any).
#annotate(model: GeneralModel): GeneralModel
Evaluate the model, using the model-checking algorithm for ALCCTL.

Figure 4.3: Overview of the Formula Implementation.

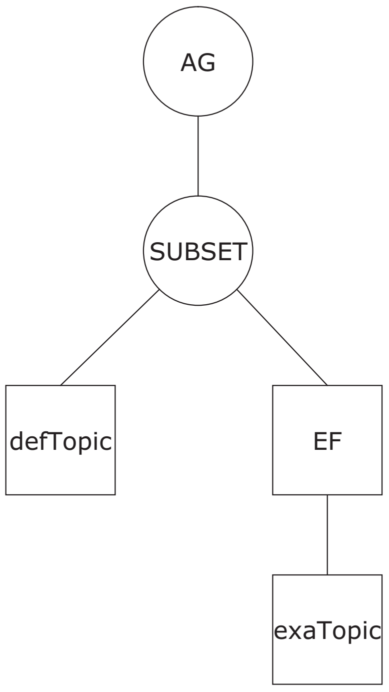
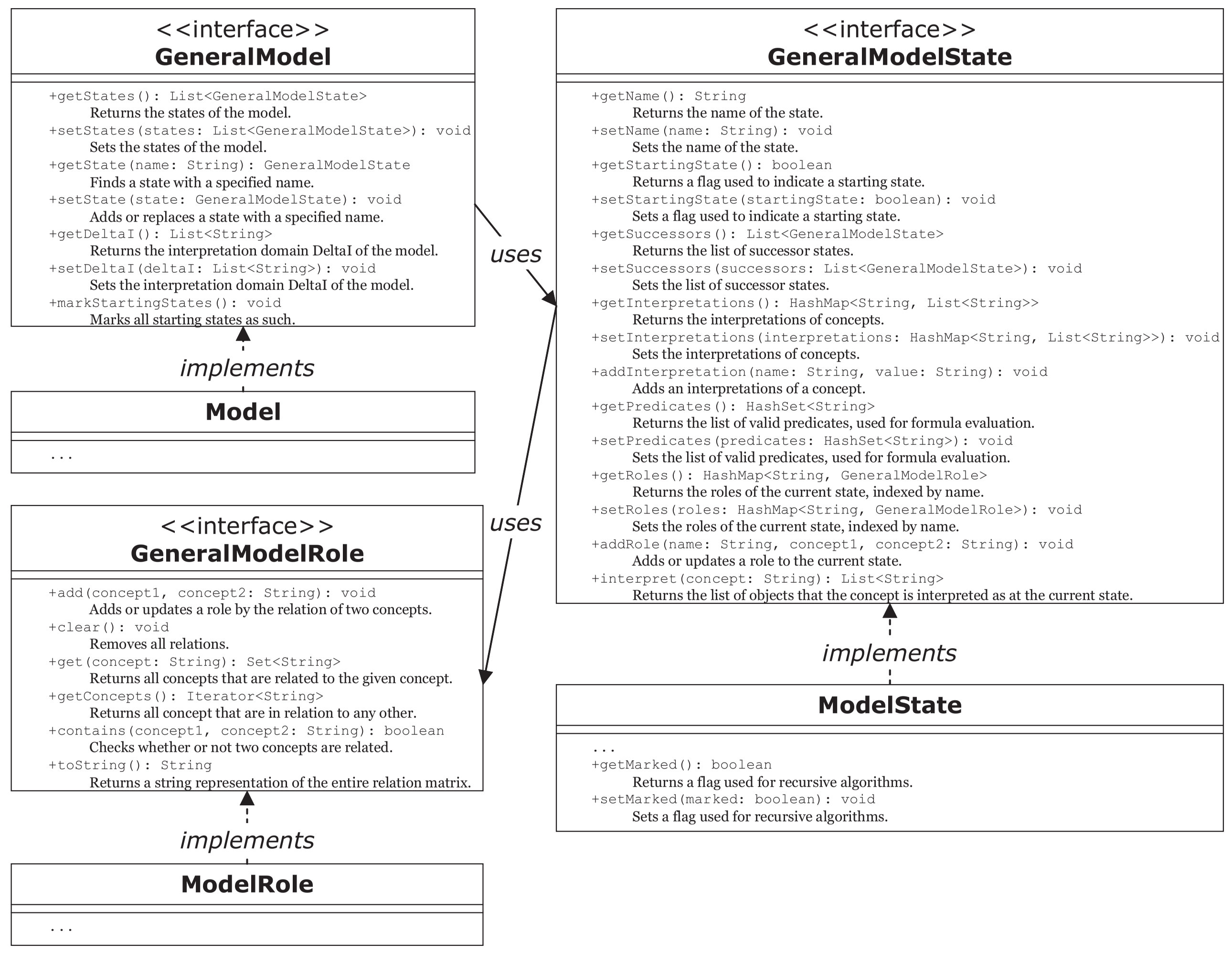


Figure 4.4: Example of a Formula Tree.

The implementing class `Model` adds methods for collecting a counter example (see chapter 6.4). For the states of the model there is an interface called `GeneralModelState`, which includes methods for annotating the state with interpretations, predicates and roles. It also provides accessor methods for the state's name. The actual $\mathcal{ALC}$CTL implementation of `GeneralModelState`, `ModelState`, adds to that several methods needed for the counter example (again, see below), as well as several methods needed for the algorithmic model checking. The interface for roles, `GeneralModelRole`, is implemented by `ModelRole`, and contains methods to add, remove and find pairings of objects.
For an overview of the model implementation, see figure 4.5.

4.3.3 Model Extraction

The model extraction is handled by the `ELearning` package. The first class that is needed is an implementation of the abstract class `DocumentAdapter`. It provides the general functionality for reading a web document from an XML file, transforming that file via XSL stylesheets, and extracting a `Content` object (the generic semantical model) from it.



```
<<interface>> GeneralModel
+getStates(): List<GeneralModelState>
        Returns the states of the model.
+setStates(states: List<GeneralModelState>): void
        Sets the states of the model.
+getState(name: String): GeneralModelState
        Finds a state with a specified name.
+setState(state: GeneralModelState): void
        Adds or replaces a state with a specified name.
+getDeltaI(): List<String>
        Returns the interpretation domain DeltaI of the model.
+setDeltaI(deltaI: List<String>): void
        Sets the interpretation domain DeltaI of the model.
+markStartingStates(): void
        Marks all starting states as such.

Model
...

<<interface>> GeneralModelRole
+add(concept1, concept2: String): void
        Adds or updates a role by the relation of two concepts.
+clear(): void
        Removes all relations.
+get(concept: String): Set<String>
        Returns all concepts that are related to the given concept.
+getConcepts(): Iterator<String>
        Returns all concept that are in relation to any other.
+contains(concept1, concept2: String): boolean
        Checks whether or not two concepts are related.
+toString(): String
        Returns a string representation of the entire relation matrix.

ModelRole
...

<<interface>> GeneralModelState
+getName(): String
        Returns the name of the state.
+setName(name: String): void
        Sets the name of the state.
+getStartingState(): boolean
        Returns a flag used to indicate a starting state.
+setStartingState(startingState: boolean): void
        Sets a flag used to indicate a starting state.
+getSuccessors(): List<GeneralModelState>
        Returns the list of successor states.
+setSuccessors(successors: List<GeneralModelState>): void
        Sets the list of successor states.
+getInterpretations(): HashMap<String, List<String>>
        Returns the interpretations of concepts.
+setInterpretations(interpretations: HashMap<String, List<String>>): void
        Sets the interpretations of concepts.
+addInterpretation(name: String, value: String): void
        Adds an interpretations of a concept.
+getPredicates(): HashSet<String>
        Returns the list of valid predicates, used for formula evaluation.
+setPredicates(predicates: HashSet<String>): void
        Sets the list of valid predicates, used for formula evaluation.
+getRoles(): HashMap<String, GeneralModelRole>
        Returns the roles of the current state, indexed by name.
+setRoles(roles: HashMap<String, GeneralModelRole>): void
        Sets the roles of the current state, indexed by name.
+addRole(name: String, concept1, concept2: String): void
        Adds or updates a role to the current state.
+interpret(concept: String): List<String>
        Returns the list of objects that the concept is interpreted as at the current state.

ModelState
...
+getMarked(): boolean
        Returns a flag used for recursive algorithms.
+setMarked(marked: boolean): void
        Sets a flag used for recursive algorithms.
```

Figure 4.5: Overview of the Model Implementation.

However, the `DocumentAdapter` is only the front-end for its subclasses `LmmlDocumentAdapter` and `Ml3DocumentAdapter`. Those classes contain the knowledge about which stylesheets to use, and they perform the actual extraction process. Thus it is possible to create other subclasses of `DocumentAdapter` that can import some other kind of document, without having to alter a single line of code anywhere else.
The abstract class `ContentAdapter` extracts a `GeneralModel` from the `Content` object returned by the `DocumentAdapter`. The actual $\mathcal{ALC}$CTL model is extracted by its subclass `ALCCTLContentAdapter`. It would also be possible to return a different kind of model, e.g. a CTL model, by creating another subclass of `ContentAdapter`.
It would also be possible to make more use of the semantic model (class `Content`), for example by connecting it to a reasoner (refer to appendix A.8 for details).
For an overview of the model extraction implementation, see figure 4.6.

4.3.4 Miscellaneous

The class `InputOutput` in the `ALCCTL` package provides methods for reading and writing formulas and models. It can parse a `Formula` from a string (using the classes of the `ALCCTL.Parser` subpackage), read a list of formulas from a file, write a formula to a string (using the `Formula.toALCCTL()` method), or write one or more formulas to a file. It can read a `GeneralModel` from an XML file and save it to an XML file (note that this is an $\mathcal{ALC}$CTL model XML file; this has nothing to do with the model extraction described above). It can create an SVG[2] view of a model, and another SVG view of the model with the steps of the algorithmic approach annotated.
For an overview of the `InputOutput` class, see figure 4.7.

The class `Utils` in the package of the same denominator provides several auxiliary methods, including two (`startTiming` and `stopTiming`) that can be used for time measuring.

Last but not least, the class `Debug` allows for multi-level debugging. All debug commands throughout the code are called with a debug level, and are only executed if this level is within the level that is currently specified. Since the standard level is `LV_NONE`, no debug messages will appear in the output of a program, unless the level is changed. The different levels allow the debugging (or checking) of different aspects, without having to comment out all the other messages.

[2]Scalable Vector Graphics, an XML based vector graphics format

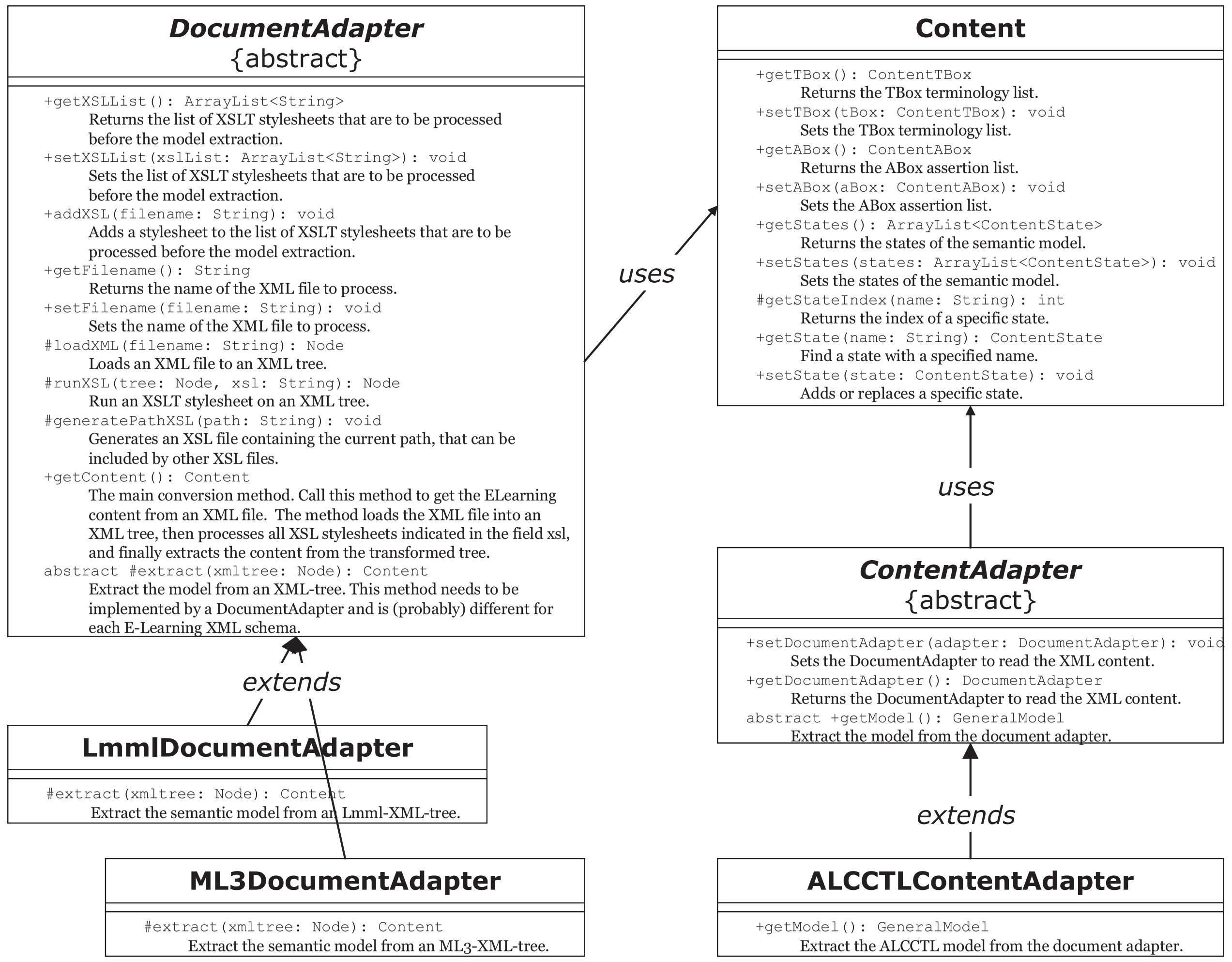
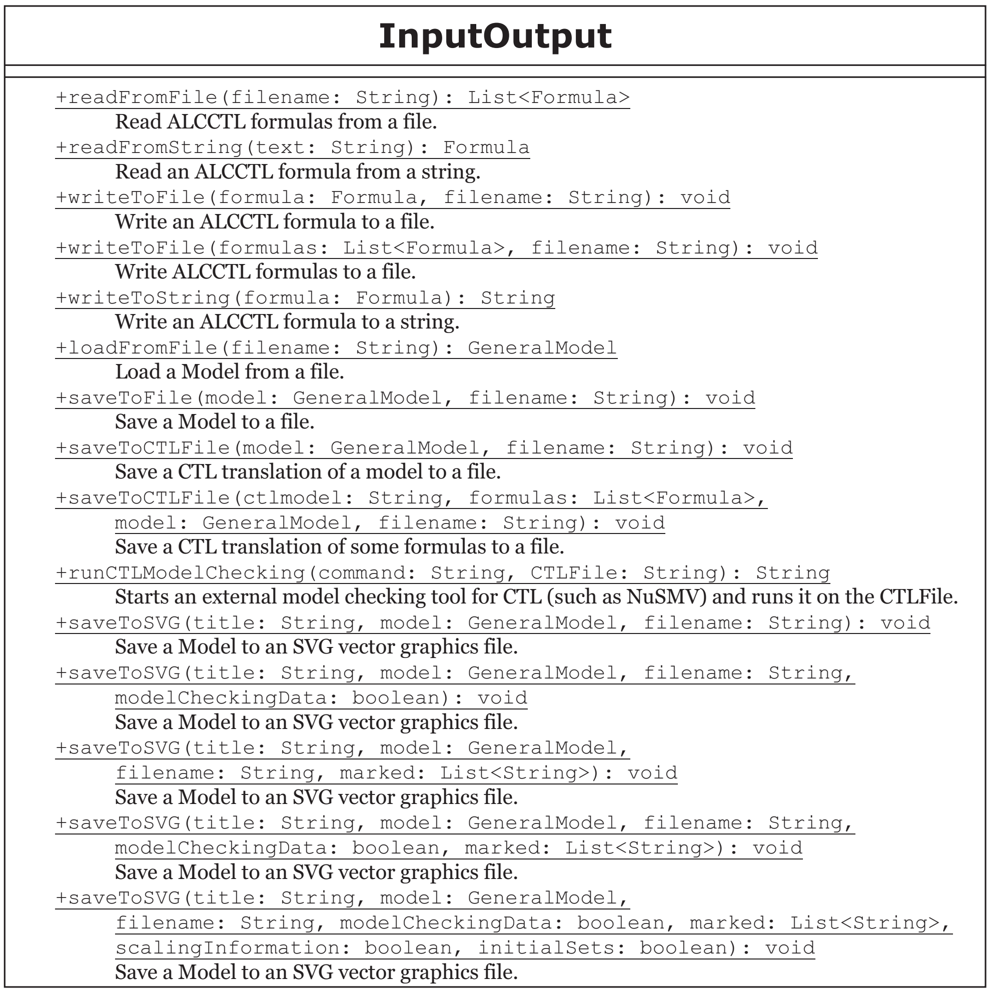

DocumentAdapter
{abstract}
+getXSLList(): ArrayList<String>
Returns the list of XSLT stylesheets that are to be processed before the model extraction.
+setXSLList(xslList: ArrayList<String>): void
Sets the list of XSLT stylesheets that are to be processed before the model extraction.
+addXSL(filename: String): void
Adds a stylesheet to the list of XSLT stylesheets that are to be processed before the model extraction.
+getFilename(): String
Returns the name of the XML file to process.
+setFilename(filename: String): void
Sets the name of the XML file to process.
#loadXML(filename: String): Node
Loads an XML file to an XML tree.
#runXSL(tree: Node, xsl: String): Node
Run an XSLT stylesheet on an XML tree.
#generatePathXSL(path: String): void
Generates an XSL file containing the current path, that can be included by other XSL files.
+getContent(): Content
The main conversion method. Call this method to get the ELearning content from an XML file. The method loads the XML file into an XML tree, then processes all XSL stylesheets indicated in the field xsl, and finally extracts the content from the transformed tree.
abstract #extract(xmltree: Node): Content
Extract the model from an XML-tree. This method needs to be implemented by a DocumentAdapter and is (probably) different for each E-Learning XML schema.
uses
Content
+getTBox(): ContentTBox
Returns the TBox terminology list.
+setTBox(tBox: ContentTBox): void
Sets the TBox terminology list.
+getABox(): ContentABox
Returns the ABox assertion list.
+setABox(aBox: ContentABox): void
Sets the ABox assertion list.
+getStates(): ArrayList<ContentState>
Returns the states of the semantic model.
+setStates(states: ArrayList<ContentState>): void
Sets the states of the semantic model.
#getStateIndex(name: String): int
Returns the index of a specific state.
+getState(name: String): ContentState
Find a state with a specified name.
+setState(state: ContentState): void
Adds or replaces a specific state.
uses
ContentAdapter
{abstract}
+setDocumentAdapter(adapter: DocumentAdapter): void
Sets the DocumentAdapter to read the XML content.
+getDocumentAdapter(): DocumentAdapter
Returns the DocumentAdapter to read the XML content.
abstract +getModel(): GeneralModel
Extract the model from the document adapter.
extends
LmmlDocumentAdapter
#extract(xmltree: Node): Content
Extract the semantic model from an Lmml-XML-tree.
ML3DocumentAdapter
#extract(xmltree: Node): Content
Extract the semantic model from an ML3-XML-tree.
extends
ALCCTLContentAdapter
+getModel(): GeneralModel
Extract the ALCCTL model from the document adapter.

Figure 4.6: Overview of the Model Extraction Implementation.



InputOutput

```
+readFromFile(filename: String): List<Formula>
     Read ALCCTL formulas from a file.
+readFromString(text: String): Formula
     Read an ALCCTL formula from a string.
+writeToFile(formula: Formula, filename: String): void
     Write an ALCCTL formula to a file.
+writeToFile(formulas: List<Formula>, filename: String): void
     Write ALCCTL formulas to a file.
+writeToString(formula: Formula): String
     Write an ALCCTL formula to a string.
+loadFromFile(filename: String): GeneralModel
     Load a Model from a file.
+saveToFile(model: GeneralModel, filename: String): void
     Save a Model to a file.
+saveToCTLFile(model: GeneralModel, filename: String): void
     Save a CTL translation of a model to a file.
+saveToCTLFile(ctlmodel: String, formulas: List<Formula>,
     model: GeneralModel, filename: String): void
     Save a CTL translation of some formulas to a file.
+runCTLModelChecking(command: String, CTLFile: String): String
     Starts an external model checking tool for CTL (such as NuSMV) and runs it on the CTLFile.
+saveToSVG(title: String, model: GeneralModel, filename: String): void
     Save a Model to an SVG vector graphics file.
+saveToSVG(title: String, model: GeneralModel, filename: String,
     modelCheckingData: boolean): void
     Save a Model to an SVG vector graphics file.
+saveToSVG(title: String, model: GeneralModel,
     filename: String, marked: List<String>): void
     Save a Model to an SVG vector graphics file.
+saveToSVG(title: String, model: GeneralModel, filename: String,
     modelCheckingData: boolean, marked: List<String>): void
     Save a Model to an SVG vector graphics file.
+saveToSVG(title: String, model: GeneralModel,
     filename: String, modelCheckingData: boolean, marked: List<String>,
     scalingInformation: boolean, initialSets: boolean): void
     Save a Model to an SVG vector graphics file.
```

Figure 4.7: Overview of the class `InputOutput`.

4.4 Implementation

4.4.1 Formula

The formula parser was generated with JLex[3] and CUP[4] from a `.lex` and a `.cup` file. The functionality can be accessed via the `InputOutput readFromString` or the `Formula parse` methods without having to call the parser directly.
Figure 4.8 shows a sequence diagram of the formula parsing process.

4.4.2 Model

In the `Model`, the list of states is implemented as a `List<GeneralModelState>`, namely an `ArrayList`. This optimises the most common use of the states: Iteration over all of them. The same is true for the interpretation domain Δ^I. In each `ModelState`, the list of successors is defined as a `List<GeneralModelState>`, and again and for the same reason is instantiated as an `ArrayList`. This is especially efficient for the algorithmic approach (see chapter 6).

The DTD of the model XML format can be seen in table A.1 in appendix A.7. Figures 4.9 and 4.10 show examples of the SVG view of a model.

4.4.3 Model Extraction

The model extraction process makes heavy use of the Java XML API, JAXP[5]. It is used for both XML parsing and XSL transformation. Even though there may be "better", or more convenient XML interfaces like e.g. the Apache[6] project Xerces[7], JAXP has the distinct advantage that it does not require any additonal packages to be installed – a simple Run-Time installation of Java will be sufficient.

During the implementation of the model extraction, I encountered a number of small problems. One of them was that all WWR *Lmml* XML files link to a Document Type Definition (DTD) with an absolute path that points to a directory on the current hard drive. I decided against disabling the DTD check, because a syntactical check before a complicated extraction process

[3]Lexical Analyzer Generator for Java,
http://www.cs.princeton.edu/ appel/modern/java/JLex/

[4]LALR Parser Generator in Java, Version 11, http://www2.cs.tum.edu/projects/cup/

[5]Java API for XML Processing

[6]Apache Software Foundation, http://www.apache.org

[7]http://xerces.apache.org/

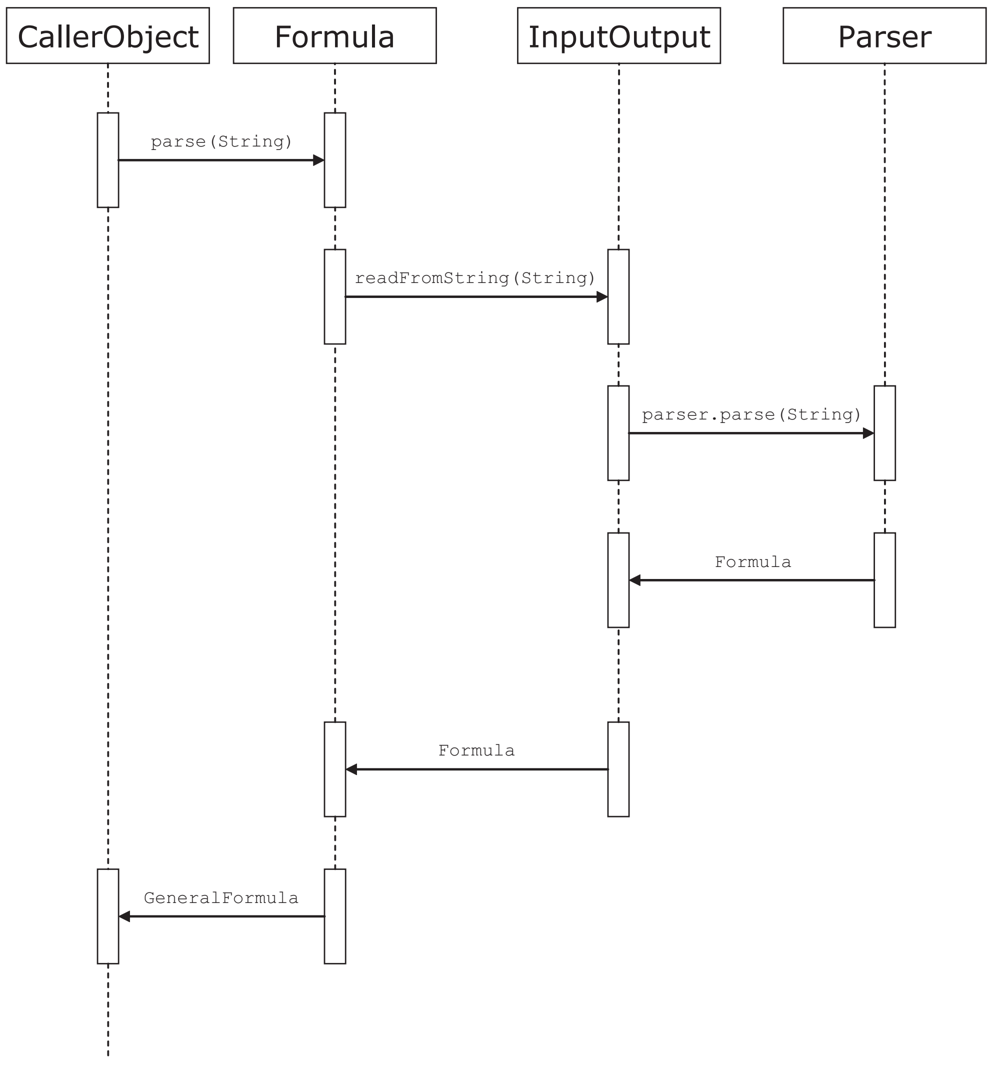


Figure 4.8: Sequence diagram of the formula parsing process.

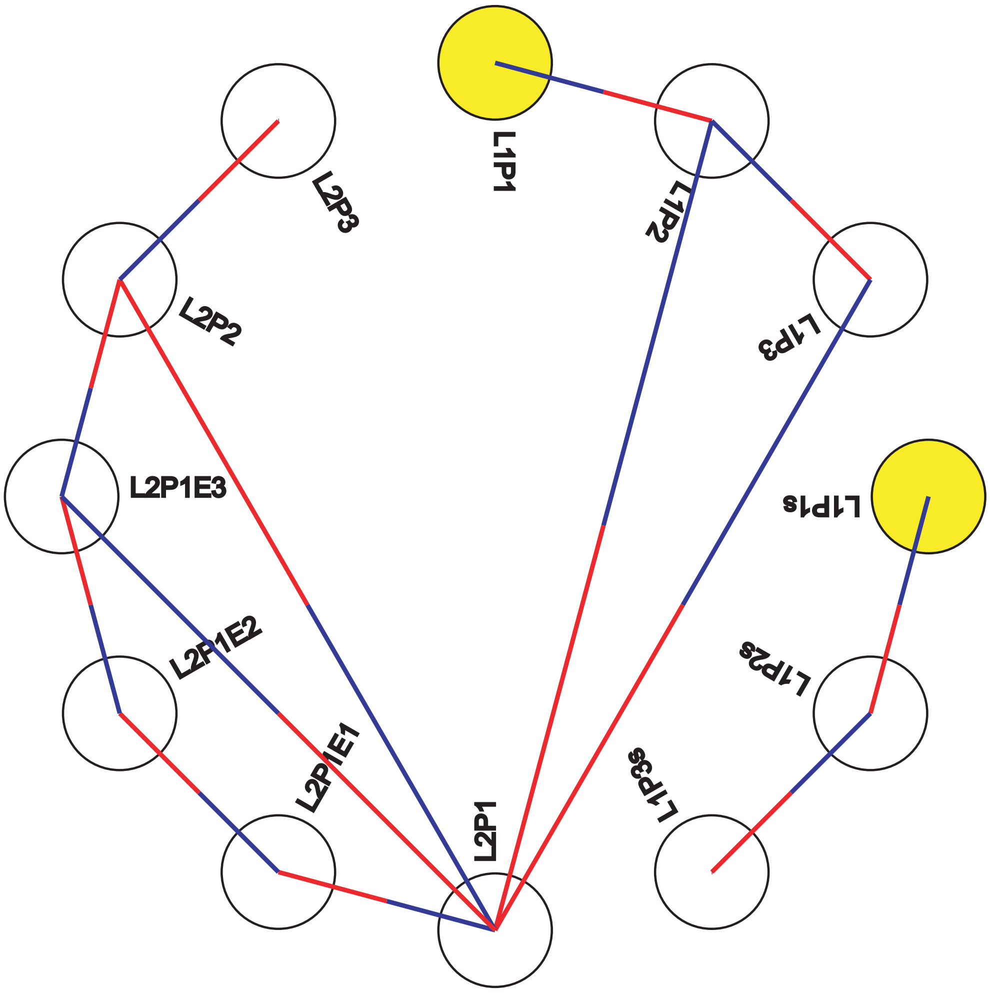


Figure 4.9: SVG View of the Test Module (without Scaling Information and State Initialisations).

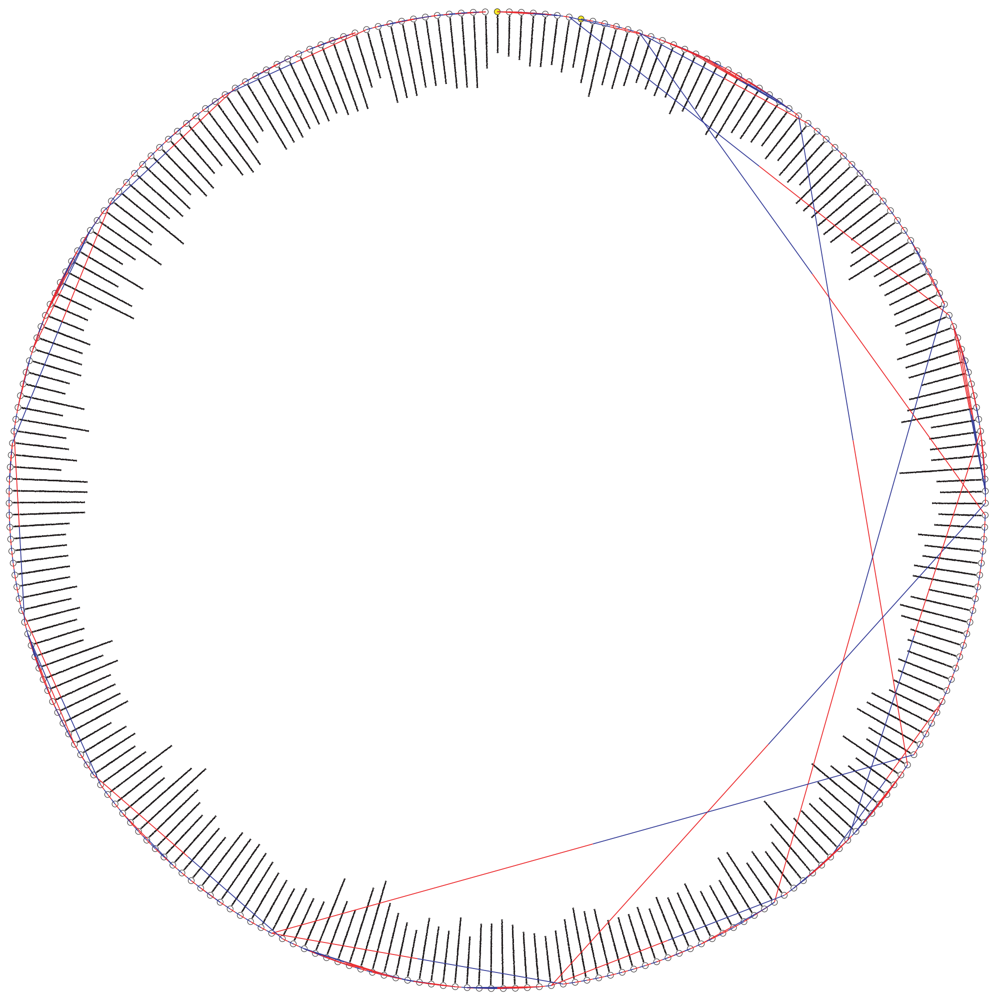


Figure 4.10: SVG View of the Fuzzy Module (without State Initialisations).

obviously seems like a good idea. So any user who wants to import *Lmml* files on his or her system will need to have the files required by *Lmml* in the appropriate local directory.
Another problem was that the XSL stylesheet that resolves the *Lmml* `include` commands needs to know the base path of the XML files. To resolve this, every time this stylesheet is called, an XSL file that contains a single variable holding the path information is dynamically created and imported by the include stylesheet.
Last but not least, since the structures of *Lmml* and $<ML^3>$ are very different, I needed to create a completely new set of stylesheets for each language, as well as two quite dissimilar `DocumentAdapter` implementations.

Let's regard an example of how model extraction works. Here is a fragment of an *Lmml* document:

```
<section title="Grundlegende Erklärungen"
    label="L1_grundlegende_erklaerungen">
  <style type="presentationUnit"/>
  <keyStatement>...</keyStatement>
  <definition terms="Fuzzy-Logik, Erweiterung der zweiwertigen Logik"
      title="Fuzzy-Logik">
    <text>
      <defined>Fuzzy-Logik</defined>
      erweitert die <iref terms="klassische Logik">klassische
      zweiwertige</iref> Logik ...
    </text>
  </definition>
  ...
  <description>
    <text>
      Bei den beiden
      <ref href="#L1_zwei_beispiele">vorangegangenen Beispielen</ref>
      handelt es sich um <iref>Fuzzy-System\e</iref>.
    </text>
  </description>
</section>
```

First, this document is run through several stylesheets. The first one (LmmlIncludes) resolves all include directives. The second (LmmlPUs) marks all presentations units and tags each element with a unique id, based on its title and position in the document. The third stylesheet (LmmlAbbreviations) resolves abbreviated syntax and inherited variant information (like intensity or target group). Large parts of that stylesheet have been copied

from the *Lmml* WWRPUB environment. The next stylesheet (LmmlScaling) converts the variant information to simple Boolean attributes like "intensity_basic=yes". It can also convert indirect references (such as irefs) to direct ones, but this feature is currently deactivated because it would increase the complexity of the model dramatically, while having no real meaning for the structure. Irefs are used to see where a definition or concept is used, but they are not meant to provide an additional narrative path.
The last stylesheet (LmmlSuccessors) attempts to find all successor states for each state. The main problem is that it has to find a successor for each possible variant instead of just accepting the next state that fits. It also has to take extensions into account, and resolve references, since those point to a successor state as well:

```
<xsl:if test="@form_script='yes'">
  <xsl:if test="@group_student='yes'">
    <xsl:if test="@intensity_basic='yes'">
      <xsl:if test="@medium_screen='yes'">
        <xsl:for-each select="following::section[@style_type='pu'
            and not(@type='extension') and @form_script='yes' and
            @group_student='yes' and @intensity_basic='yes' and
            @medium_screen='yes'][1]">
          <xsl:call-template name="succ"/>
        </xsl:for-each>
      </xsl:if>
      <xsl:if test="@medium_paper='yes'">
      ...
<xsl:for-each select="descendant::ref | descendant::related">
  <xsl:if test="@href">
    <xsl:if test="contains(@href,'#')">
      <xsl:variable name="href" select="substring-after(@href,'#')"/>
      <xsl:element name="successor_state">
        <xsl:attribute name="idref">
          <xsl:value-of select="/descendant::*[@id=$href]/@pu_id"/>
        </xsl:attribute>
        ...
```

Here is the above fragment, after it has been XSL transformed:

```
<section form_script="yes" form_slide="no" group_student="yes"
    group_teacher="yes" intensity_advanced="yes" intensity_basic="yes"
    intensity_expert="yes" medium_paper="yes" medium_screen="yes"
    id="L1_grundlegende_erklaerungen" style_type="pu"
    title="Grundlegende Erklärungen">
  <successor_state idref="L1_aufbau_von_fuzzysystemen" type="successor"/>
  <successor_state idref="L1_zwei_beispiele" type="ref"/>
  <definition id="1.1.1.4. Fuzzy-Logik"
      pu_id="L1_grundlegende_erklaerungen" title="Fuzzy-Logik">
    <text>
      <defined>Fuzzy-Logik</defined>
      erweitert die klassische zweiwertige Logik auf ...
    </text>
  </definition>
  <description id="N66742" pu_id="L1_grundlegende_erklaerungen">
    <text>
      Bei den beiden
      <ref href="#L1_zwei_beispiele" id="N66746">vorangegangenen
      Beispielen</ref> handelt es sich um Fuzzy-Systeme.
    </text>
  </description>
</section>
```

Now it's the `LmmlDocumentAdapter`'s turn. It traverses the resulting XML tree and extracts states and assertions from it, including scaling information and things like definitions or examples:

```
private Content traverseTree(Node tree, Content c) {
  if (tree instanceof Element) {
    Element e = (Element)tree;
    if (e.getTagName().equals("section") && e.getAttribute("style_type")
        .equals("pu")) {
      String id = e.getAttribute("id");
      ContentState cstate = c.getState(id);
      if (cstate == null)
        cstate = new ContentState(id);
      // this adds the scaling information 'hasScaling' and 'scaleTo'
      // to the ABox of c
      c = getScaling(e, id, id, c);
      NodeList successors = e.getElementsByTagName("successor_state");
      for (int i=0;i<successors.getLength();i++) {
        String idref = successors.item(i).getAttribute("idref");
        // no duplicate successors
        if (!cstate.successors.contains(idref) && !idref.equals(""))
          cstate.successors.add(idref);
      }
      //
      if (cstate.successors.size() == 0) {
        cstate.successors.add(id);
        cstate.reftypes.put(id, "successor");
      }
      c.setState(cstate);
    } else {
      ...
        } else if (e.getTagName().equals("algorithm")) {
          c = getScaling(e, id, stateid, c);
          // add standard assertions like ‘‘Fragment’’ or ‘‘topicOf’’,
          // as well as one called ‘‘Algorithm’’ (depend. on the param.)
          assertions = updateAssertions(e, id, "Algorithm", assertions);
        } else if (e.getTagName().equals("definition")) {
          c = getScaling(e, id, stateid, c);
          assertions = updateAssertions(e, id, "Definition", assertions);
          // find all title strings for an element
          String[] titles = getTitle(e);
          for (int i=0;i<titles.length;i++)
            if (!titles[i].equals("")) {
              assertions.add(new ContentABoxAssertion("definedTopic",
                titles[i]));
              assertions.add(new ContentABoxAssertion("definedAt", id,
                titles[i]));
            }
        } else if (e.getTagName().equals("demonstration")) {
          c = getScaling(e, id, stateid, c);
          assertions = updateAssertions(e, id, "Demonstration",
            assertions);
          ...
```

This tree traversal returns semantic content in the form of a `Content` class. This content is now converted into an $\mathcal{ALC}$CTL model by the `ALCCTLContentAdapter`:

```
List<GeneralModelState> mstates = new ArrayList<GeneralModelState>();
// create state list
for (int j=0;j<c.getStates().size();j++) {
  ContentState cstate = c.getStates().get(j);
  GeneralModelState mstate = new ModelState(cstate.getName());
  // convert assertions to interpretations and roles
  List<ContentABoxAssertion> assertions = c.getABox().getAssertions(
      cstate.getName());
  for (int i=0;i<assertions.size();i++) {
    if (assertions.get(i).isDualValued())
      mstate.addRole(assertions.get(i).getName(), assertions.get(i)
        .getValue(), assertions.get(i).getValue2());
    else
      mstate.addInterpretation(assertions.get(i).getName(), assertions
        .get(i).getValue());
  }
  mstates.add(mstate);
}
m.setStates(mstates);
// set successor states
...
m.markStartingStates();
// set deltaI
List<String> deltaI = new ArrayList<String>();
// extract them from the ABox, to make sure to get exactly those that
// are used; no more, no less
...
    String value = assertion.get(i).getValue();
    if (!deltaI.contains(value))
      deltaI.add(value);
    if (assertion.get(i).isDualValued()) {
      value = assertion.get(i).getValue2();
      if (!deltaI.contains(value))
        deltaI.add(value);
    }
...
m.setDeltaI(deltaI);
```

Finally, the model can be saved as an XML file itself. Here is a fragment of this file:

```
<state name="L1_grundlegende_erklaerungen" startingState="no">
  <successor name="L1_aufbau_von_fuzzysystemen" type="successor"/>
  <successor name="L1_zwei_beispiele" type="ref"/>
  <interpretation name="Definition">
    <i_item value="1.1.1.4. Fuzzy-Logik"/>
    <i_item value="1.1.1.4. N66721"/>
  </interpretation>
  <interpretation name="Description">
```

```
    <i_item value="1.1.1.4. N66742"/>
  </interpretation>
  <interpretation name="definedTopic">
    <i_item value="Fuzzy-Logik"/>
    ...
  <role name="hasScaling">
    <r_item concept1="1.1.1.4. Fuzzy-Logik" concept2="GroupStudent"/>
    <r_item concept1="1.1.1.4. Fuzzy-Logik" concept2="FormScript"/>
    <r_item concept1="1.1.1.4. Fuzzy-Logik" concept2="IntensityBasic"/>
    <r_item concept1="L1_grundlegende_erkl." concept2="GroupStudent"/>
    <r_item concept1="L1_grundlegende_erkl." concept2="FormScript"/>
    <r_item concept1="L1_grundlegende_erkl." concept2="IntensityBasic"/>
    ...
  </role>
  <role name="topicOf">
    <r_item concept1="Fuzzy-System" concept2="1.1.1.4. N66721"/>
    ...
</state>
...
<deltaI>
  <d_item value="FormSlide"/>
  <d_item value="Fuzzy-Systeme für Klassifikation: Iris"/>
  ...
```

Extraction of $<ML^3>$ models works analogous, even though other stylesheets and an adjusted XML tree traversal are required. Figure 4.11 shows a sequence diagram of the model extraction process.

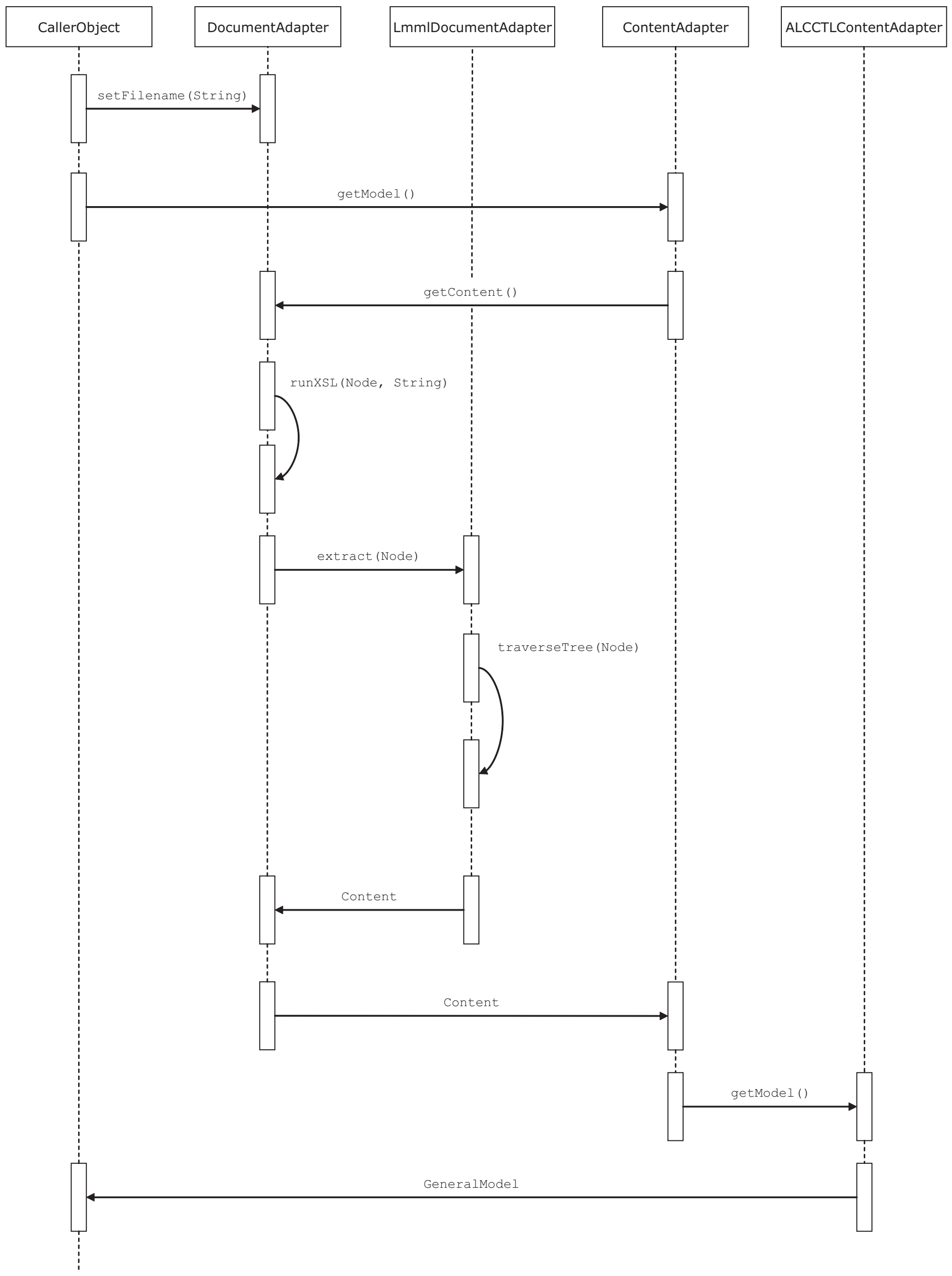


Figure 4.11: Sequence diagram of the model extraction process.

Chapter 5

Approach I: Reduction

The first possible approach is to reduce $\mathcal{ALC}\mathsf{CTL}$ formulas and models to CTL, and then use the existing CTL model checking tools.

5.1 Description

The reduction process itself is – thanks to the "translation templates" from [WF04a] – rather straight forward. The formula tree is traversed from top to bottom, each operator reduced in turn, and the process continues recursively for its argument(s).
The first of the following two definitions has been taken almost literally from [WF04a] (note that since CTL is itself a temporal logic, the temporal operators need no explicit translation):

Definition 11 (CTL reduction of an $\mathcal{ALC}\mathsf{CTL}$ Formula) *Given is an $\mathcal{ALC}\mathsf{CTL}$ formula ϕ and an $\mathcal{ALC}\mathsf{CTL}$ model $M = (S, \longrightarrow, I, \Delta^I)$. The CTL reduction ϕ_{Δ^I} is inductively obtained from ϕ as follows:*

$$\begin{aligned}
\left(\psi_1 \dot{\sqsubseteq} \psi_2\right)_{\Delta^I} &=_{def} \textstyle\bigwedge_{c\in\Delta^I} (\psi_1\,[c] \;\rightarrow\; \psi_2\,[c]), \textit{ for concepts } \psi_1 \textit{ and } \psi_2\\
(\neg\phi)_{\Delta^I} &=_{def} \neg\,(\phi_{\Delta^I})\\
(\phi_1 \,\wedge\, \phi_2)_{\Delta^I} &=_{def} \phi_{1\,\Delta^I} \,\wedge\, \phi_{2\,\Delta^I}
\end{aligned}$$

where $\cdot\,[c]$ is for A being an atomic concept and ψ an $\mathcal{ALC}\mathsf{CTL}$ state concept recursively defined as

$$\begin{aligned}
A\,[c] &=_{def} A(c)\\
(\psi_1 \,\sqcap\, \psi_2)\,[c] &=_{def} \psi_1\,[c] \,\wedge\, \psi_2\,[c]\\
(\psi_1 \,\sqcup\, \psi_2)\,[c] &=_{def} \psi_1\,[c] \,\vee\, \psi_2\,[c]\\
(\neg\psi)\,[c] &=_{def} \neg\,(\psi\,[c])\\
(\exists R.\psi)\,[c] &=_{def} \textstyle\bigvee_{d\in\Delta^I} (R(c,d) \,\wedge\, \psi\,[d])\\
(\forall R.\psi)\,[c] &=_{def} \textstyle\bigwedge_{d\in\Delta^I} (R(c,d) \,\wedge\, \psi\,[d])
\end{aligned}$$

Definition 12 (CTL reduction of an $\mathcal{ALC}$CTL Model) *Given is an $\mathcal{ALC}$CTL model $M = (S, \longrightarrow, I, \Delta^I)$. The CTL reduction $M' = (S, \longrightarrow, L)$ is obtained by reducing all formulas for this model to CTL after definition 11 using Δ^I and by replacing the interpretation function I with a labelling function L. The latter is defined as $L(s) =_{def} \{A(c) \in \mathbf{A} \mid c \in A^{I(s)}\} \cup \{R(a,b) \in \mathbf{R} \mid (a,b) \in R^{I(s)}\}$, where $\mathbf{A}$ is the set of atomic concepts and $\mathbf{R}$ is the set of atomic roles [WF04a].*

5.2 Specification

5.2.1 Formula

As mentioned above, the method `toCTL` does the actual CTL reduction of the formula. The implementation of an $\mathcal{ALC}$CTL formula allows for all temporal operators, including AB and EB. Those are not supported by most CTL tools, however, so that the implementation of the `toCTL` method for the B operators includes a base reduction similar to that of the `translate` method, thus substituting other operators instead that can then be reduced to CTL without problem.

5.2.2 CTL Model Export

The class `InputOutput` (see chapter 4.3.4) also provides two methods `saveToCTLFile` that expect a model or a list of formulas and converts them to a CTL model. The first version of the method creates the actual CTL model, while the second version reads the previously created model and adds the reduced formulas to it, saving the entire model to a new file. This file can now be used as input for CTL model checking tools. Finally, the method `runCTLModelChecking` can start an external CTL model checker. It takes a CTL file created with `saveToCTLFile` as a parameter, as well as the path to the external tool.

5.3 Implementation

The structure of the implementation for the reduction approach can be seen in figure 5.1.

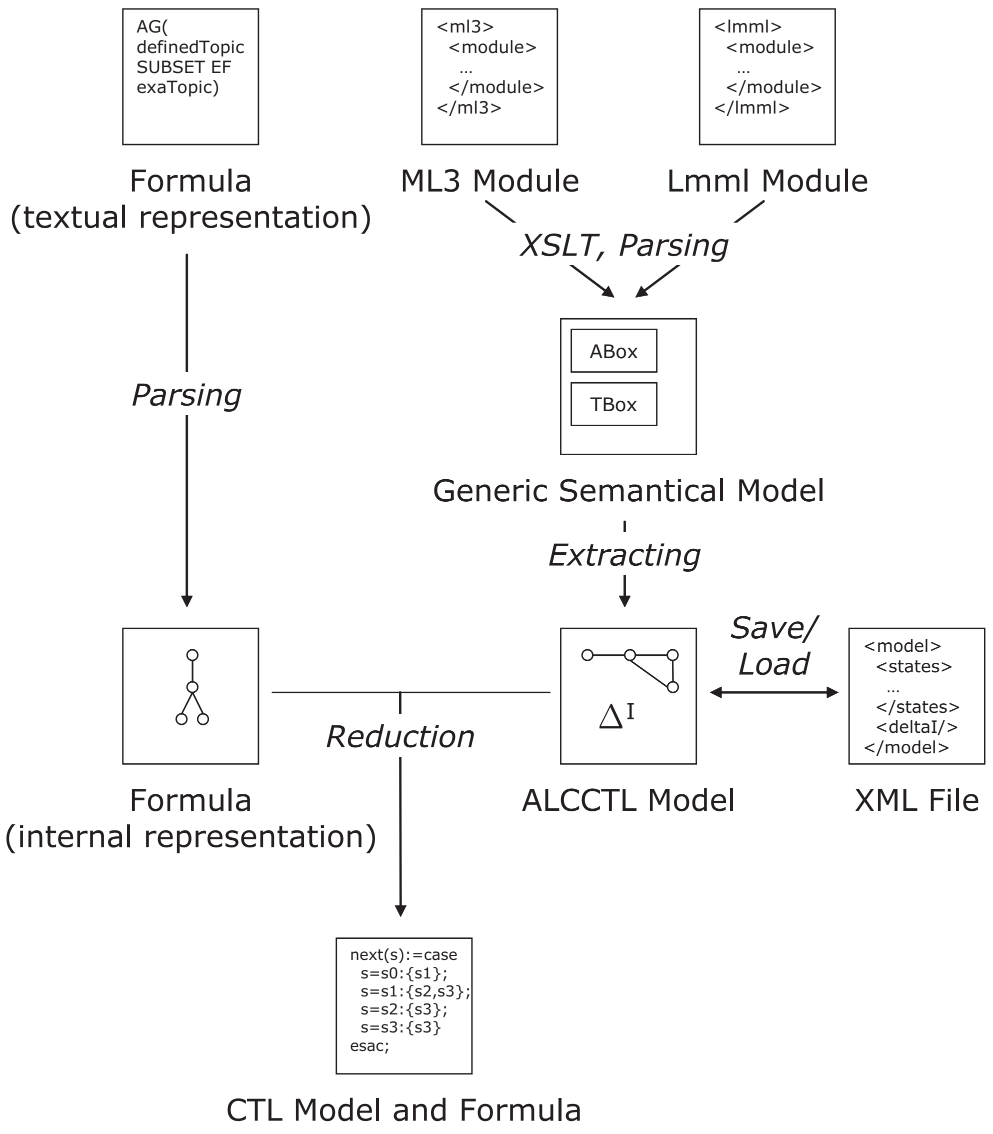


Figure 5.1: Diagram of the reduction process.

```
public String toCTL(GeneralModel model) {
  return "EX("+arg.toCTL(model)+")";
}
```

Table 5.1: `FormulaEX`

```
public String toCTL(GeneralModel model) {
  String result = "";
  String arg1CTL = "";
  String arg2CTL = "";
  for (int i=0;i<model.getDeltaI().size();i++) {
    arg1CTL = arg1.toCTL(model.getDeltaI().get(i), model);
    arg2CTL = arg2.toCTL(model.getDeltaI().get(i), model);
    if (i<model.getDeltaI().size()-1)
      result += "("+arg1CTL+" -> "+arg2CTL+") & ";
    else
      result += "("+arg1CTL+" -> "+arg2CTL+")";
  }
  return result;
}
```

Table 5.2: `FormulaSubset`

5.3.1 Formula

The reduction process is done along the lines of definition 11. Each `Formula` subclass implements its own `toCTL(GeneralModel)` method that does the local conversion and makes the recursive call(s). Each `Concept` subclass similarly implements a `toCTL(String, GeneralModel)` method. The `String` is the element of Δ^I that is currently evaluated (corresponding to the $\cdot\,[c]$ syntax of definition 11).
See tables 5.1 through 5.5 for implementation examples.

5.3.2 Model

Reducing an $\mathcal{ALC}$CTL model to a CTL model that can be read by CTL model checking tools, namely NuSMV, is done in three stages. First, all required variables are declared. Second, all those variables are initialised. Third, for each state transition the new values for all variables are listed.
Formally, a NuSMV file looks like this. A module declaration

```
protected String toCTL(String element, GeneralModel model) {
  String result = "E[";
  result += arg1.toCTL(element, model);
  result += " U ";
  result += arg2.toCTL(element, model);
  return result+"]";
}
```

Table 5.3: ConceptEU

```
protected String toCTL(String element, GeneralModel model) {
  String result = "";
  String argCTL = arg.toCTL(element, model);
  if (model.getDeltaI().size() > 0)
    result += "("+name+"("+element+", "+model.getDeltaI().get(0)
           + ") & "+argCTL+")";
  for (int i=1;i<model.getDeltaI().size();i++) {
    result += " | ("+name+"("+element+", "+model.getDeltaI().get(i)
           + ") & "+argCTL+")";
  }
  return result;
}
```

Table 5.4: ConceptExists

```
protected String toCTL(String element, GeneralModel model) {
  return Utils.formatCTL(element+"_is_"+name);
}
```

Table 5.5: ConceptAtomic

```
MODULE main
```

is followed by the declaration of variables.

```
VAR
DFA_is_Definition: boolean;
DFA_is_Example: boolean;
DFA_is_Task: boolean;
DFA_is_Fragment: boolean;
...
state : {Intro, Def, Exa, Conc};
```

These declarations are succeeded by a list of initial assignments:

```
ASSIGN
init(state) := Intro;
init(DFA_is_Definition) := 0;
init(DFA_is_Example) := 0;
...
```

After that comes the value transition list.

```
next(state) := case
  state = Intro : {Def};
  state = Def : {Exa, Conc};
  state = Exa : {Conc};
  state = Conc : {Conc};
esac;
next(DFA_is_Definition) := case
  next(state) = Def : 1;
  1: 0;
esac;
next(DFA_is_Task) := case
  1: 0;
esac;
...
```

Finally, the file is concluded with a list of formulas.

```
SPEC (DFA_is_definedTopic -> EF(DFA_is_exemplifiedTopic)) & ...
```

Note that this file is still valid for SMV, not only NuSMV.

To actually create such a file from a `GeneralModel`, it is necessary to repeatedly iterate over all states to find all interpretations, roles and predicates, and again to find their respective values when converting them to single

```
w.write("VAR\n");
for (int i=0;i<model.getStates().size();i++) {
  keys = model.getStates().get(i).getInterpretations().keySet().iterator();
  while (keys.hasNext()) {
    String key = keys.next();
    for (int j=0;j<model.getDeltaI().size();j++) {
      String tmp = Utils.formatCTL(model.getDeltaI().get(j)+"_is_"+key);
      if (!interpretations.contains(tmp))
        interpretations.add(tmp);
    }
  }
}
for (int i=0;i<interpretations.size();i++)
  w.write(interpretations.get(i)+" : boolean;\n");
```

Table 5.6: Extract of the CTL export method `saveToCTLFile`.

Boolean expressions. Table 5.6 shows an extract of the code. Since names (for states, concepts, roles etc.) in `GeneralModel` can contain virtually any character, while NuSMV limits the range to letters, numbers and the underscore character, the conversion method `Utils.formatCTL` is needed to trim an $\mathcal{ALC}$CTL name into CTL shape. Particularly umlauts have proven to be hard to filter out. Since this conversion is done a lot, it slows the entire extraction process down. A possible optimisation would be to create a copy of the `GeneralModel` and convert all names there – that way, each conversion would only have to be done once, and not every time the variable is used.
In table 5.7 an extract of the `runCTLModelChecking` method that runs the external model checking tool can be seen.

Figure 5.2 shows a sequence diagram of the CTL reduction process.

A major problem for the CTL reduction are roles, inherited from the description logics $\mathcal{ALC}$. Roles are completely unknown in CTL, with the result that they have to be expanded to their maximum possible size: $\Delta^I \times \Delta^I$! Each role has to be split into $|\Delta^I| \cdot |\Delta^I|$ Boolean variables of the form $rolename\_concept_1\_is\_concept_2$, e.g. $topicOf\_Definition\_is\_DFA$. This inflates the CTL model size notably, not to mention extraction time or processing time! To shorten the extraction/reduction time, I have separated the two reduction processes: The model reduction has to be done only once, and the CTL model is written to a file. When reducing formulas, this model is simply copied before the formula declarations, to create a processable CTL model. Since the model reduction is by far the more time consuming task

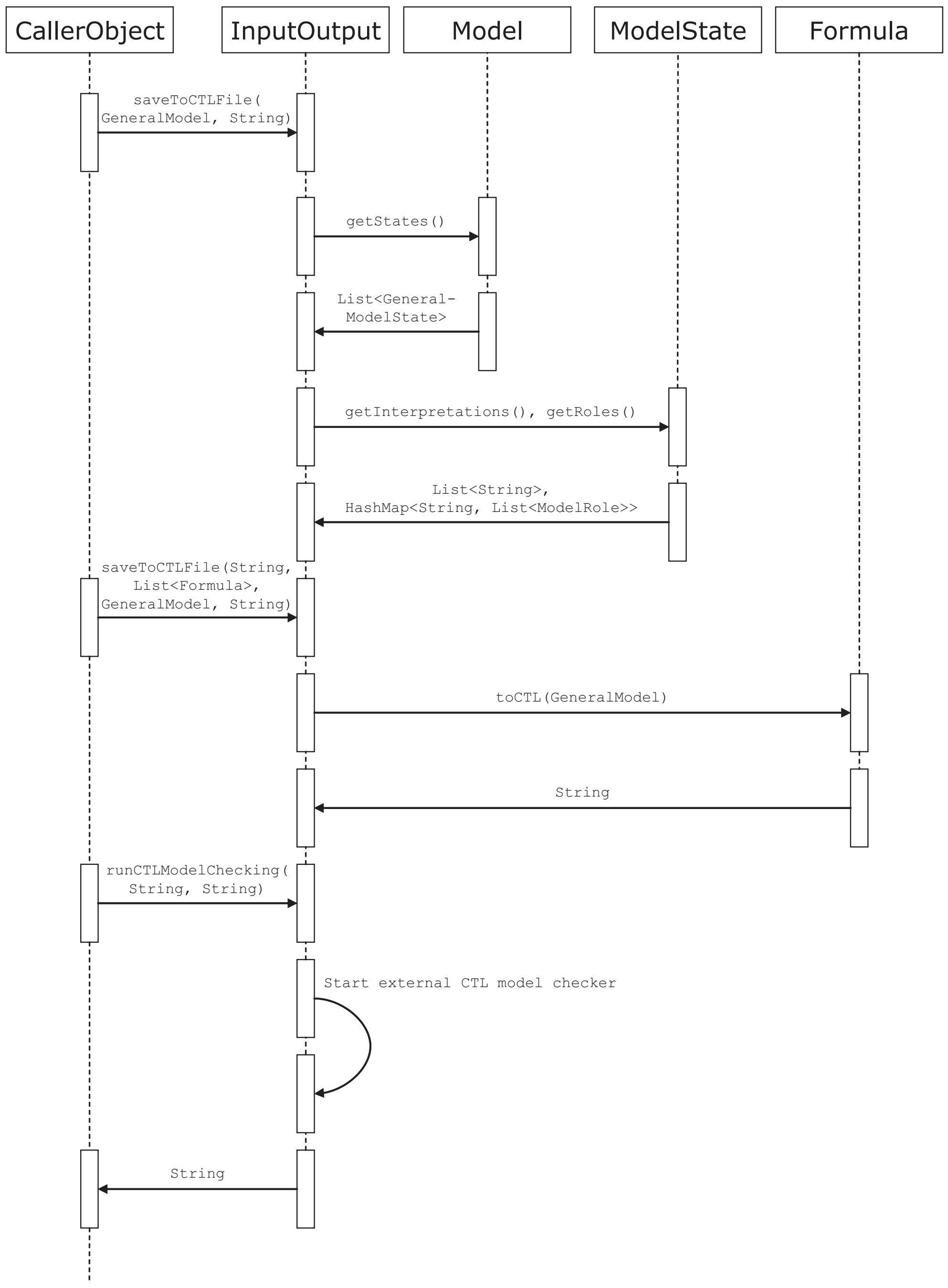

CallerObject
InputOutput
Model
ModelState
Formula
saveToCTLFile(GeneralModel, String)
getStates()
List<General- ModelState>
getInterpretations(), getRoles()
List<String>, HashMap<String, List<ModelRole>>
saveToCTLFile(String, List<Formula>, GeneralModel, String)
toCTL(GeneralModel)
String
runCTLModelChecking(String, String)
Start external CTL model checker
String

Figure 5.2: Sequence diagram of the CTL reduction process.

```
public static String runCTLModelChecking(String command,
                                         String CTLFile) {
  Runtime run = Runtime.getRuntime();
  ShutdownThread hook = new ShutdownThread();
  Process mc = run.exec(command + " " + CTLFile);

  // create a shutdownhook that will terminate
  // the external application
  // if the java program is terminated.
  hook.process = mc;
  run.addShutdownHook(hook);

  // read the output of the external application
  BufferedReader output = new BufferedReader(
    new InputStreamReader(mc.getInputStream()));
  String result = ""; String line = output.readLine();
  while (line != null) {
    result += line + "\n"; line = output.readLine();
  }
  output = new BufferedReader(
    new InputStreamReader(mc.getErrorStream()));
  ...

  // remove the shutdownhook when the
  // external application has finished.
  run.removeShutdownHook(hook);
  return result;
}
```

Table 5.7: Extract of method `runCTLModelChecking` that runs the external CTL tool.

of the two, this cuts down the total run time considerably. Unfortunately, at the same time, it disallows the use of an optimisation that was previously possible. If no roles are used in the formulas, they could be omitted in the CTL model reduction entirely. Tests have shown that this shortens the time requirements decidedly. However, in the current implementation, the formulas are not known at the time of the model reduction. In theory, it would have even been possible to refine that optimisation, by generally including only those roles that are required (possibly none). This concept could also have been extended to the interpretation of concepts: Concepts not mentioned in a formula could be omitted as well. However, the trade-off is worthwhile.
Pending the outcome of the analysis of the reduction approach, I have not yet tried to combine both refinements (the separation and the role/interpretation screening). I have, though, included an optional parameter for the model reduction that indicates whether roles should be included in the CTL model or not (the default being "yes", of course).
One theoretical optimisation is to include only those elements of Δ^I in the role reduction that are actually part of the role relation (e.g. "Section_1_Introduction" would probably never be in the "scaleTo" relation). That is made quite difficult by the role semantics, however. Recall the semantics of the $\forall$ quantor from chapter 2.4, definition 9: $(\forall R.\psi)^{I(s)} =_{def} \left\{a \in \Delta^I \mid \forall b.(a,b) \in R^{I(s)} \rightarrow b \in \psi^{I(s)}\right\}$. It includes the entirety of Δ^I, not just the elements of the role relation. But more important is definition 11 from chapter 5.1: $(\forall R.\psi)[c] =_{def} \bigwedge_{d \in \Delta^I} (R(c,d) \wedge \psi[d])$. Since c can be any element of Δ^I, and d iterates over Δ^I, the current semantics clearly suggest a scaling of $\Delta^I \times \Delta^I$. Nonetheless, it would be possible to reduce the second Δ^I (d) to the scope of the role. Given the dismal performance of CTL role reduction (see below), this would most likely be a just a drop in a bucket.

Another problem concerning the CTL reduction is that CTL models, as recognised by most tools, can only have a single starting state. Since that constraint is not necessarily valid for $\mathcal{ALC}$CTL models (see chapter 6.3.2), it may occur that not the entire model is checked. A possible solution is to extract multiple CTL models from a single $\mathcal{ALC}$CTL model, each with a distinct starting state. Again, I will await the outcome of the analysis before implementing this solution.

5.4 Complexity

Let $M = (S, \longrightarrow, I, \Delta^I)$ be an $\mathcal{ALC}$CTL model, C the set of atomic concepts and R the set of atomic roles. Exporting M to a CTL model M' is done in several steps. The first is to create lists of interpretations of concepts and roles and to declare them as variables, with complexity classes $O\left(|S| \cdot |C|^2 \cdot |\Delta^I|\right)$ and $O\left(|S| \cdot |R|^2 \cdot |\Delta^I|^2\right)$, respectively.

```
// listing concepts:
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  keys = model.getStates().get(i).getInterpretations().keySet()
    .iterator();
  // iterate over (possibly) all concepts: |C|
  while (keys.hasNext()) {
    String key = keys.next();
    // iterate over DeltaI: |DeltaI|
    for (int j=0;j<model.getDeltaI().size();j++) {
      String tmp = Utils.formatCTL(model.getDeltaI().get(j)+"_is_"+key);
      // filter duplicates
      // ArrayList takes linear time for 'contains': |C|
      if (!interpretations.contains(tmp))
        interpretations.add(tmp);
    }
  }
}

// listing roles:
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  keys = model.getStates().get(i).getRoles().keySet().iterator();
  // iterate over (possibly) all roles: |R|
  while (keys.hasNext()) {
    String key = keys.next();
    // iterate over DeltaI: |DeltaI|
    for (int j=0;j<model.getDeltaI().size();j++)
      // iterate over DeltaI: |DeltaI|
      for (int k=0;k<model.getDeltaI().size();k++) {
        String tmp = Utils.formatCTL(key)+"_"
          + Utils.formatCTL(model.getDeltaI().get(j))+"_is_"
          + Utils.formatCTL(model.getDeltaI().get(k));
        // filter duplicates
        // ArrayList takes linear time for 'contains': |R|
        if (!roles.contains(tmp))
          roles.add(tmp);
      }
  }
```

The next step is to declare all states as variables. This is in $O\left(|S|\right)$.

```
// declare states as variables:
w.write("state : {");
// iterate of all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  w.write(Utils.formatCTL(model.getStates().get(i).getName()));
  if (i<model.getStates().size()-1)
    w.write(", ");
}
w.write("};\n");
```

Then comes the initialisation phase, which is in $O\left(|C|^2 \cdot |\Delta^I|\right)$ for the concepts, and in $O\left(|R|^2 \cdot |\Delta^I|^2\right)$ for the roles.

```
// initialisation phase (concepts):
// iterate over (possibly) all concepts: |C|
for (int i=0;i<interpretations.size();i++) {
  w.write("init("+interpretations.get(i)+") := ");
  boolean init = false;
  keys = model.getStates().get(startingState).getInterpretations()
    .keySet().iterator();
  // iterate over (possibly) all concepts: |C|
  while (keys.hasNext()) {
    String key = keys.next();
    interpretation = model.getStates().get(startingState).interpret(key);
    // iterate over interpretations (can be DeltaI): |DeltaI|
    for (int j=0;j<interpretation.size();j++) {
      String tmp = Utils.formatCTL(interpretation.get(j)+"_is_"+key);
      // ArrayList access is in constant time
      if (interpretations.get(i).equals(tmp))
        init = true;
      if (init) break;
    }
    if (init) break;
  }
  if (init) w.write("1"); else w.write("0");
  w.write(";\n");
}

// initialisation phase (roles):
// iterate over (possibly) all roles: |R|
for (int i=0;i<roles.size();i++) {
  w.write("init("+roles.get(i)+") := ");
  boolean init = false;
  keys = model.getStates().get(startingState).getRoles()
    .keySet().iterator();
  // iterate over (possibly) all roles: |R|
  while (keys.hasNext()) {
    String key = keys.next();
    role = model.getStates().get(startingState).getRoles().get(key);
    Iterator<String> rkeys = role.getConcepts();
    // iterate over left-hand-side of role (can be DeltaI): |DeltaI|
    while (rkeys.hasNext()) {
      String rkey = rkeys.next();
      Iterator<String> elements = role.get(rkey).iterator();
      // iterate over right-hand-side of role (can be DeltaI): |DeltaI|
      while (elements.hasNext()) {
        String tmp = Utils.formatCTL(key)+"_"+Utils.formatCTL(rkey)
          + "_is_"+Utils.formatCTL(elements.next());
        // ArrayList access is in constant time
        if (roles.get(i).equals(tmp))
          init = true;
        ...
```

The last step is the assignment of new values on each state transition. The complexity is in $O\left(|S|^2\right)$ for the states themselves (the actual state transitions), and $O\left(|S| \cdot |C|^2 \cdot |\Delta^I|\right)$ and $O\left(|S| \cdot |R|^2 \cdot |\Delta^I|^2\right)$ for concepts and roles, respectively.

```
// state transitions:
w.write("next(state) := case\n");
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  w.write("\tstate = "
    + Utils.formatCTL(model.getStates().get(i).getName())+" : {");
  // iterate over (possibly) all states: |S|
  for (int j=0;j<model.getStates().get(i).getSuccessors().size();j++) {
    w.write(Utils.formatCTL(model.getStates().get(i).getSuccessors()
      .get(j).getName()));
    if (j<model.getStates().get(i).getSuccessors().size()-1)
      w.write(", ");
  }
  w.write("};\n");
}
w.write("esac;\n");

// changing values for concepts:
// iterate over (possibly) all concepts: |C|
for (int i=0;i<interpretations.size();i++) {
  w.write("next("+interpretations.get(i)+") := case\n");
  // iterate over all states: |S|
  for (int k=0;k<model.getStates().size();k++) {
    keys = model.getStates().get(k).getInterpretations()
      .keySet().iterator();
    // iterate over (possibly) all concepts: |C|
    while (keys.hasNext()) {
      String key = keys.next();
      interpretation = model.getStates().get(k).interpret(key);
      // iterate over (possibly) entire DeltaI: |DeltaI|
      for (int j=0;j<interpretation.size();j++) {
        String tmp = Utils.formatCTL(interpretation.get(j)+"_is_"+key);
        // ArrayList access is in constant time
        if (interpretations.get(i).equals(tmp))
          w.write("\tnext(state) = "
            + Utils.formatCTL(model.getStates().get(k).getName())
            + " : 1;\n");
      } } }
  w.write("\t1: 0;\n");
  w.write("esac;\n");
}

// changing values for roles:
// iterate over (possibly) all roles: |R|
for (int i=0;i<roles.size();i++) {
  w.write("next("+roles.get(i)+") := case\n");
  // iterate over all states: |S|
  for (int k=0;k<model.getStates().size();k++) {
    keys = model.getStates().get(k).getRoles().keySet().iterator();
    // iterate over (possibly) all roles: |R|
    while (keys.hasNext()) {
      String key = keys.next();
      role = model.getStates().get(k).getRoles().get(key);
      Iterator<String> rkeys = role.getConcepts();
      // iterate over left-hand-side of role (can be DeltaI): |DeltaI|
      while (rkeys.hasNext()) {
        String rkey = rkeys.next();
        Iterator<String> elements = role.get(rkey).iterator();
        // iterate over right-hand-side of role (can be DeltaI): |DeltaI|
        while (elements.hasNext()) {
          String tmp = Utils.formatCTL(key)+"_"+Utils.formatCTL(rkey)
            + "_is_"+Utils.formatCTL(elements.next());
          // ArrayList access is in constant time
          if (roles.get(i).equals(tmp))
            w.write("\tnext(state) = "
              + Utils.formatCTL(model.getStates().get(k).getName())
              + " : 1;\n");
        } } } }
  w.write("\t1: 0;\n");
  w.write("esac;\n");
}
```

This adds up to a complexity class of $|S| \cdot |C|^2 \cdot |\Delta^I| + |S| \cdot |R|^2 \cdot |\Delta^I|^2 + |S| + |C|^2 \cdot |\Delta^I| + |R|^2 \cdot |\Delta^I|^2 + |S|^2 + |S| \cdot |C|^2 \cdot |\Delta^I| + |S| \cdot |R|^2 \cdot |\Delta^I|^2$. By the reduction rules of O, this can be reduced to the following complexity class:

Proposition 1 (Complexity of the CTL Model Export Algorithm) *Exporting a* CTL *model M' from M (denoted as $M \triangleright M'$) has the following complexity: $M \triangleright M' \in O\left(\max\left(|S| \cdot |C|^2 \cdot |\Delta^I|, |S| \cdot |R|^2 \cdot |\Delta^I|^2\right)\right)$.*

As there are usually only small numbers of atomic concepts and roles, the deciding factors are $|S|$ and – even more importantly – $|\Delta^I|$. Thus, the complexity can be further reduced to $M \triangleright M' \in O\left(|S| \cdot |R|^2 \cdot |\Delta^I|^2\right)$.

In the current implementation, however, another very expensive factor is the `formatCTL` method: It scales linearly to the length of the input string, but with a high constant factor! Since the input string is arbitrary in length (only confined by the available memory), this factor is not even present in

the above complexity class.
The complexity is a worst-case estimate: The actual values can be smaller. As long as roles are included, they cannot be much smaller, though. Only the number of roles in each state can be less than $|R|$. $|\Delta^I|^2$ remains unchanged (see chapter 5.3.2, the section about roles, for more details). If roles are excluded, however, the complexity shrinks to a more manageable $O\left(|S| \cdot |C|^2 \cdot |\Delta^I|\right)$. Due to the relatively small size of C and the reduction from $|\Delta^I|^2$ to $|\Delta^I|$, the advantage is considerable.

Proposition 2 (Complexity of the CTL Formula Reduction)
Reducing an $\mathcal{ALC}$CTL formula ϕ to CTL is in $O\left(|\Delta^I| \cdot |\phi|\right)$, as can be easily seen from the reduction definition of $\forall$ (see definition 11 and below), which has the most complex reduction algorithm. $|\phi|$ denotes the number of subformulas of ϕ: The number of "components" of the formula.

```
String argCTL = arg.toCTL(element, model);
if (model.getDeltaI().size() > 0)
  result += "("+Utils.formatCTL(name)+"_"+Utils.formatCTL(element)
    + "_is_"+Utils.formatCTL(model.getDeltaI().get(0))+" -> "+argCTL+")";
for (int i=1;i<model.getDeltaI().size();i++) {
  result += " & ("+Utils.formatCTL(name)+"_"+Utils.formatCTL(element)
    + "_is_"+Utils.formatCTL(model.getDeltaI().get(i))+" -> "+argCTL+")";
}
```

Proposition 3 (Complexity of the CTL Model Checking) *Model checking CTL is P-complete with an upper bound of $O(|S| \cdot |\phi|)$ [Sch02], so the complete model checking process using the reduction approach has the following complexity: $M, s \models_{CTL\,Reduction} \phi \in O\left(|S| \cdot |R|^2 \cdot |\Delta^I|^2 + |\Delta^I| \cdot |\phi| + |S| \cdot |\phi|\right) = O\left(|S| \cdot |R|^2 \cdot |\Delta^I|^2\right)$ for a formula ϕ (the complexity of checking ϕ against M in state s). However, the upper bound for model checking CTL does not consider the number of variables, which is in this case among the deciding factors.*

5.5 Tests

To test the implementation and the performance of the reduction approach, I will use five different models and up to 29 different formulas. Table 5.9 holds an overview over the formulas. The models are the following:

- TinyTest – This is the model introduced as an example in chapter 2.2.
- Test – This is a small (yet not tiny) test model, shown in figure 4.9.

- EZKom – This is a real E-Learning module: The WWR EZKom module, extracted from *Lmml*.
- Fuzzy – This is another real module: The WWR Fuzzy module, extracted from *Lmml*. See figure 4.10.
- BIS – This also is an E-Learning module: The WWR BIS module, extracted from <ML^3>.

A statistical overview of each model is shown in table 5.8.

All tests have been run on a computer with the following configuration:

Processor	Intel® Pentium™ 4, 2.4 Ghz, Hyperthreading
RAM (physical)	1 GB
Disk Capacity	60 GB Raid 1
Operating System	Microsoft® Windows® XP, SP2
Java	Version 1.5.0
JLex	Version 1.2.6
CUP	Version 11a
NuSMV	Version 2.3.1

A representative subset of the tests has been repeated on a computer in the graduates' room[1] to prove reproducibility. All values shown here are from the initial tests with the above configuration.

The following tests have been performed:

1. All five models have been extracted from *Lmml* or <ML^3> and saved as an XML file.
2. Then a `CTL` model has been exported from each of them, once without roles, once including roles.
3. Now every model was checked against up to 27 formulas with NuSMV, creating a formula/model `CTL` file for each formula in the process.

Time measurement has been done with the methods `startTiming` and `stopTiming` from the package `Utils`. Both use the Java method `System.currentTimeMillis`. Memory usage has been measured using the Windows® Task-Manager. The statistics in table 5.8 were created with the command-line tool ALCCTLStatistics, while all tests were done using the command-line tool ALCCTLModelChecking.

The model extraction times (for *Lmml* or <ML^3>, respectively) are listed in table 5.10. The time it takes to export each model to `CTL` as well

[1]Prof. Dr. B. Freitag, Chair for Information Management, Faculty for Mathematics and Informatics, University of Passau

	TinyTest	Test	EZKom	Fuzzy	BIS
Size of Δ^I	15	35	983	1183	69
Number of states	4	12	161	255	39
Total number of successors	5	15	203	385	70
Mean number of succ./state	1.25	1.25	1.26	1.51	1.79
Total number of interpret.	6	18	526	683	26
Mean number of int./state	1.5	1.5	3.27	2.68	0.7
Total number of roles	14	38	612	920	96
Mean number of roles/state	3.5	3.17	3.8	3.6	2.46
Memory usage	485kb	774kb	10.289mb	13.925mb	1.144mb

Table 5.8: Model Complexity.

as the resulting file size can be seen in table 5.11. The time it takes the CTL model checking tool (NuSMV) to run the model is listed in table 5.13, while the memory usage during the processing is shown in table 5.12. Table 5.14 shows the actual results of the run, that is, whether or not the formula is true.
A graph that shows how the reduction approach scales with respect to the size of the model is shown in figure 7.4, chapter 7. It compares the scaling of the two model checking approaches.

Trying to export a CTL model *including* roles for both the EZKom and the Fuzzy module failed with an `OutOfMemoryError`: The Java heap space was insufficient. Considering that both models did not produce results for any simple formula, testing them with the complicated formulas 26 and 27 (both include roles) seemed like a waste of time. Therefore, I have made no effort to procure a CTL model with roles for EZKom and Fuzzy, neither by trying to further optimise the model export process, nor by simply increasing the Java heap space.
Model checking of formula 16 against the Test model was aborted after more than 12 hours – as were several other tests (see table 5.14). The EZKom model completely withstood any attempts at model checking – even formula 8 (*true*) did not terminate within a day! Tests on the Fuzzy module terminated rather quickly, but did not produce any result, not even an error. Since the Fuzzy model is by far the largest, it stands to reason that it exceeds some internal limit of NuSMV, especially considering the aborted test of EZKom.

Number	Formula
Formula 1	$\bot \mathrel{\dot\sqsubseteq} \top$
Formula 2	$EG(\bot \mathrel{\dot\sqsubseteq} \top)$
Formula 3	$EX(\bot \mathrel{\dot\sqsubseteq} \top)$
Formula 4	$E\left[true\,U\,\bot \mathrel{\dot\sqsubseteq} \top\right]$
Formula 5	$\bot \mathrel{\dot\sqsubseteq} EG\bot$
Formula 6	$\bot \mathrel{\dot\sqsubseteq} EX\top$
Formula 7	$\bot \mathrel{\dot\sqsubseteq} E\,[\top\,U\,\top]$
Formula 8	$true$
Formula 9	$\neg true$
Formula 10	$true\ \wedge\ true$
Formula 11	$true\ \wedge\ false$
Formula 12	$true\ \vee\ false$
Formula 13	$false\ \vee\ false$
Formula 14	$\top\ \doteq\ \top$
Formula 15	$\top\ \doteq\ \bot$
Formula 16	$definedTopic \mathrel{\dot\sqsubseteq} EF\,exemplifiedTopic\ \sqcap\ \top$
Formula 17	$definedTopic \mathrel{\dot\sqsubseteq} EF\,exemplifiedTopic\ \sqcup\ \neg\top$
Formula 18	$definedTopic \mathrel{\dot\sqsubseteq} EX\,exemplifiedTopic$
Formula 19	$definedTopic \mathrel{\dot\sqsubseteq} EF\,exemplifiedTopic$
Formula 20	$definedTopic \mathrel{\dot\sqsubseteq} E\,[definedTopic\,U\,exemplifiedTopic]$
Formula 21	$definedTopic \mathrel{\dot\sqsubseteq} AG\,exemplifiedTopic$
Formula 22	$AG(definedTopic \mathrel{\dot\sqsubseteq} EX\,exemplifiedTopic)$
Formula 23	$AG(definedTopic \mathrel{\dot\sqsubseteq} EF\,exemplifiedTopic)$
Formula 24	$AG(definedTopic \mathrel{\dot\sqsubseteq} E\,[definedTopic\,U\,exemplifiedTopic])$
Formula 25	$AG(definedTopic \mathrel{\dot\sqsubseteq} AG\,exemplifiedTopic)$
Formula 26	$AG(definedTopic \mathrel{\dot\sqsubseteq} EF\exists topicOf.Example)$
Formula 27	$AG(\forall topicOf.Definition \mathrel{\dot\sqsubseteq} EF\exists topicOf.Fragment)$

Table 5.9: Test Formulas.

TinyTest	Test	EZKom	Fuzzy	BIS
1s 735ms	1s 875ms	10s 719ms	13s 781ms	4s 422ms

Table 5.10: Model Extraction Test Results.

	TinyTest	Test	EZKom	Fuzzy	BIS
Time (no roles)	93ms	469ms	16m 51s	26m 41s	672ms
Time (roles)	4s 266ms	1m 31s	–$^{(aborted)}$	–$^{(aborted)}$	26m 2s
File size (no roles)	9.79kb	40.6kb	2.79mb	3.7mb	48.5kb
File size (roles)	222kb	1.25mb	–$^{(aborted)}$	–$^{(aborted)}$	7.86mb

Table 5.11: CTL Export Test Results.

TinyTest	Test	EZKom	Fuzzy	BIS
8.5mb	15.4mb	–	–	9.3mb

Table 5.12: CTL Model Checking Memory Usage (usage measured for formula 1, no roles).

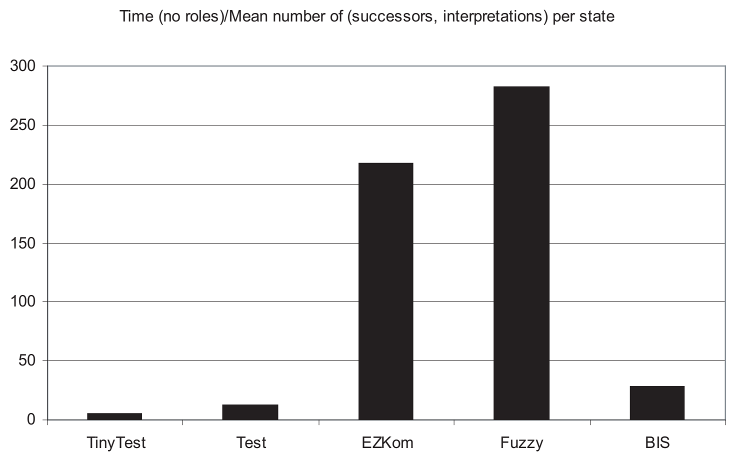


Figure 5.3: CTL Model Export: Relation of $time\,(no\,roles)$ to $\sum_{\{succ,\,interp\}} \varnothing$.

	TinyTest	Test	EZKom	Fuzzy	BIS
Formula 1	391ms	3s 250ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	2s 312ms
Formula 2	375ms	3s 313ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	719ms
Formula 3	375ms	3s 266ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	734ms
Formula 4	406ms	3s 359ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	734ms
Formula 5	297ms	3s 422ms	–$^{\text{(aborted)}}$	–$^{\text{(no result)}}$	735ms
Formula 6	391ms	3s 375ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	719ms
Formula 7	297ms	3s 375ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	750ms
Formula 8	391ms	3s 234ms	–$^{\text{(aborted)}}$	–$^{\text{(no result)}}$	703ms
Formula 9	313ms	3s 469ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	735ms
Formula 10	391ms	3s 375ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	735ms
Formula 11	297ms	3s 484ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	750ms
Formula 12	375ms	3s 391ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	719ms
Formula 13	391ms	3s 485ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	766ms
Formula 14	391ms	3s 375ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	719ms
Formula 15	391ms	3s 500ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	750ms
Formula 16	500ms	–$^{\text{(aborted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	266ms
Formula 17	500ms	–$^{\text{(not attempted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	250ms
Formula 18	297ms	3s 485ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	250ms
Formula 19	547ms	–$^{\text{(aborted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	250ms
Formula 20	469ms	–$^{\text{(aborted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	266ms
Formula 21	344ms	3s 500ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	265ms
Formula 22	375ms	3s 454ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	250ms
Formula 23	578ms	–$^{\text{(not attempted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	266ms
Formula 24	594ms	–$^{\text{(aborted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	250ms
Formula 25	391ms	4s 62ms	–$^{\text{(not attempted)}}$	–$^{\text{(no result)}}$	265ms
Formula 26	3m 56s	–$^{\text{(aborted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(not attempted)}}$
Formula 27	3m 50s	–$^{\text{(aborted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(not attempted)}}$	–$^{\text{(not attempted)}}$

Table 5.13: CTL Model Checking Test Results.

	TinyTest	Test	EZKom	Fuzzy	BIS
Formula 1	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 2	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 3	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 4	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 5	*true*	*true*	_(aborted)	_(no result)	*true*
Formula 6	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 7	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 8	*true*	*true*	_(aborted)	_(no result)	*true*
Formula 9	*false*	*false*	_(not attempted)	_(no result)	*false*
Formula 10	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 11	*false*	*false*	_(not attempted)	_(no result)	*false*
Formula 12	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 13	*false*	*false*	_(not attempted)	_(no result)	*false*
Formula 14	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 15	*false*	*false*	_(not attempted)	_(no result)	*false*
Formula 16	*true*	_(aborted)	_(not attempted)	_(no result)	*true*
Formula 17	*true*	_(not attempted)	_(not attempted)	_(no result)	*true*
Formula 18	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 19	*true*	_(aborted)	_(not attempted)	_(no result)	*true*
Formula 20	*true*	_(aborted)	_(not attempted)	_(no result)	*true*
Formula 21	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 22	*true*	*true*	_(not attempted)	_(no result)	*true*
Formula 23	*true*	_(not attempted)	_(not attempted)	_(no result)	*true*
Formula 24	*true*	_(aborted)	_(not attempted)	_(no result)	*true*
Formula 25	*false*	*false*	_(not attempted)	_(no result)	*true*
Formula 26	*false*	_(aborted)	_(not attempted)	_(not attempted)	_(not attempted)
Formula 27	*false*	_(aborted)	_(not attempted)	_(not attempted)	_(not attempted)

Table 5.14: CTL Model Checking Results.

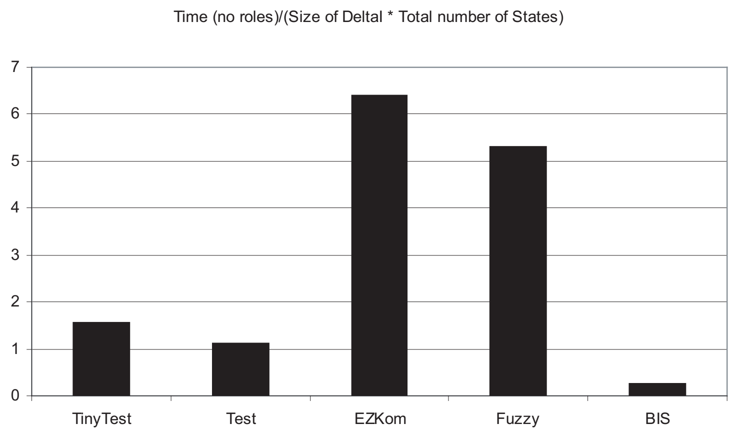


Figure 5.4: CTL Model Export: Relation of $time\,(no\,roles)$ to $|\Delta^I| \cdot |S|$.

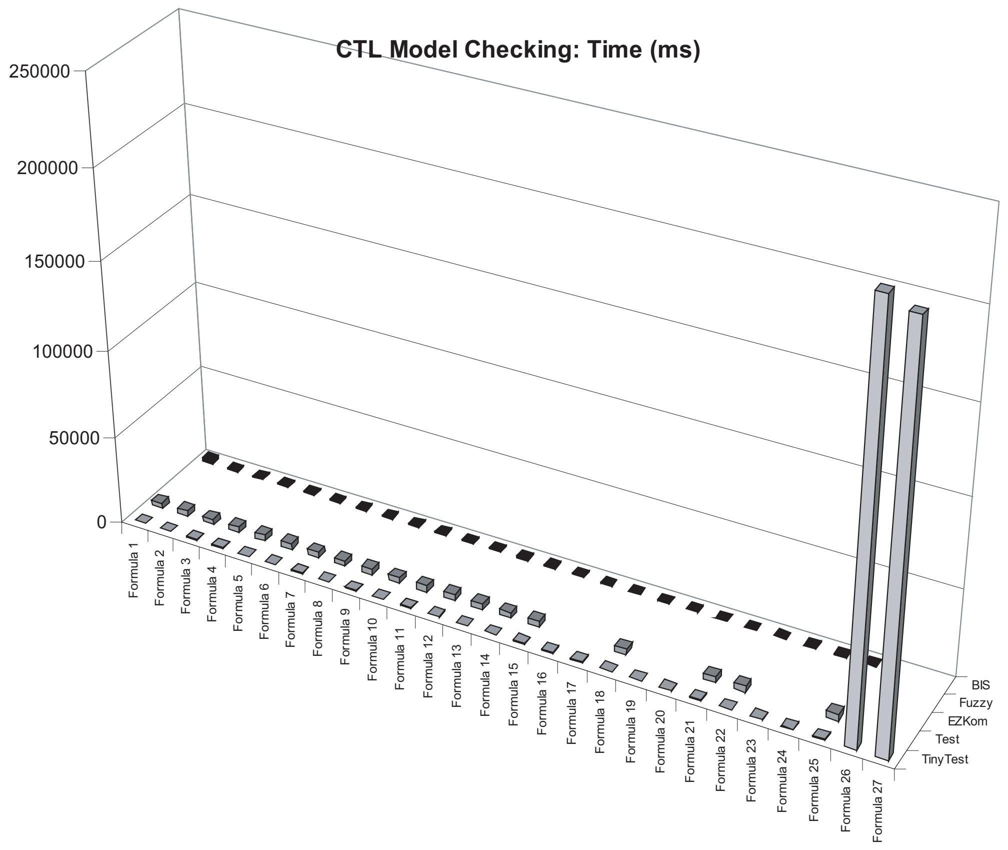


Figure 5.5: CTL Model Checking: *time*.

5.6 Evaluation

From figure 5.3 it is apparent that there is no linear relation between the complexity of a model, and the time it takes to export it to CTL. This is hardly surprising: A more exponential relation was to be expected (see below). This can be clearly seen in table 5.11. Comparing figures 5.3 and 5.4 shows that the number of successors or interpretations has tremendous impact on the export time: Relating the export time to those numbers clearly reflects the actual proportions, while relating the time only to the number of states and the size of Δ^I distorts the picture.

Figure 5.5 shows impressively how fast CTL model checking a reduced $\mathcal{ALC}$CTL model can actually be – provided it is possible at all and there are no roles in the formula!

Regarding the test results for model checking, there are a few oddities that spring to mind. Many of the times measured are exactly the same, for example the amount of 391ms appears eight times in table 5.13. I believe there are several reasons for that: Time measurement with Java always is somewhat imprecise – the implementation of the timing mechanism might have something to do with the likeness of the numbers. Then, the operating system caches memory access: Once something has been read, it takes little time to read it over and over again – like the CTL model! But most importantly, it would seem that the major time factor with NuSMV is reading and processing the model, while the actual formula has only minor impact. This is supported by the general similarity of the model checking time *within the same* model, and by the differences in performance *between* models.
The general performance decrease for similar formulas containing the EF or EU operators suggests that those are especially troublesome. This is emphasised by the abortive attempts with the Test module on such formulas.
A further noteworthy oddity is the fact that the BIS model as a whole performs better than the – far smaller – Test model. It seems likely that this is due to the lower number of interpretations per state with the BIS module: This greatly reduces the amount of work that is to be done on each state transition.
There is another strange effect concerning the BIS model: All formulas that consist solely of generics like $\top$, $true$ or $false$ (formulas 1 through 15) perform worse than the – more complex – formulas 16 through 24! I can only assume that this has something to do with the internal optimisation process of NuSMV – a process that very probably focuses on the evaluation of predicates, disregarding constants and such entirely.

One thing, however, seems to be obvious: The sheer complexity of the models, resulting in a huge number of variables, is a principal problem for NuSMV, which scales exponentially to the number of variables [McM93, CCGR00].

Chapter 6

Approach II: Algorithmic

The second possible approach to model checking $\mathcal{ALC}$CTL is to re-implement the standard CTL model checking algorithm for $\mathcal{ALC}$CTL. The implementation can re-use most of the components defined in chapter 4.

6.1 Description

Implementing the model checking algorithm defined in definition 10 and described in chapter 2.3 for CTL requires a representation of both an $\mathcal{ALC}$CTL formula and an $\mathcal{ALC}$CTL model. Those, including methods for reading, writing and extracting them, have already been described. What remains is the description of the actual algorithm itself, as well as any additional objects or concepts that might be required for it.

6.1.1 Temporal Base

There are ten possible combinations of temporal quantors for both formulas and concepts, plus several logical and set operators, as well as basic concepts and truth values. To implement model checking in the algorithmic approach for all of them would require an unreasonable amount of time and effort. However, it is possible to reduce the ten temporal quantors (AB, AF, AG, AU, AX, EB, EF, EG, EU and EX) to a base of three, and to express the remaining seven as combinations of these base quantors. There are several different bases possible; I have decided to use EG, EU and EX, because the absence of A-quantors allows for a fairly efficient implementation.

Definition 13 (Semantic Equivalence) *Two formulas ϕ and ψ are semantically equivalent ($\phi \equiv \psi$), if any state in any model that satisfies one formula, also satisfies the other [HR00].*

The equivalences are as follows:

$$\begin{array}{rcl}
A\,[\phi_1\,B\,\phi_2] & \equiv & \neg E\,[\neg\phi_1\,U\,\phi_2] \\
AF\phi & \equiv & \neg EG\neg\phi \\
AG\phi & \equiv & \neg E\,[\top\,U\,\neg\phi] \\
A\,[\phi_1\,U\,\phi_2] & \equiv & \neg\,(E\,[\neg\phi_2\,U\,\neg\phi_1\,\wedge\,\neg\phi_2]\,\vee\,EG\neg\phi_2) \\
AX\phi & \equiv & \neg EX\neg\phi \\
E\,[\phi_1\,B\,\phi_2] & \equiv & E\,[\neg\phi_2\,U\,\phi_1\,\wedge\,\neg\phi_2]\,\vee\,EG\neg\phi_2 \\
EF\phi & \equiv & E\,[\top\,U\,\phi]
\end{array}$$

These equivalences are proven in [HR00], except three: $AF\phi \equiv \neg EG\neg\phi$, $AG\phi \equiv \neg E\,[\top\,U\,\neg\phi]$ and $E\,[\phi_1\,B\,\phi_2] \equiv E\,[\neg\phi_2\,U\,\phi_1\,\wedge\,\neg\phi_2]\;\vee\;EG\neg\phi_2$. However, the equivalences $AF\phi \equiv A\,[\top\,U\,\phi]$, $AG\phi \equiv \neg EF\neg\phi$ and $E\,[\phi_1\,B\,\phi_2] \equiv \neg A\,[\neg\phi_1\,U\,\phi_2]$ are proven, from which follows:

$$\begin{array}{rcl}
AF\phi & \equiv & A\,[true\,U\,\phi] \Leftrightarrow^{(A[\phi_1\,U\,\phi_2]\equiv\neg(E[\neg\phi_2\,U\,\neg\phi_1\,\wedge\,\neg\phi_2]\,\vee\,EG\neg\phi_2))} \\
AF\phi & \equiv & \neg\,(E\,[\neg\phi\,U\,\neg true\,\wedge\,\neg\phi]\,\vee\,EG\neg\phi) \Leftrightarrow \\
AF\phi & \equiv & \neg\,(E\,[\neg\phi\,U\,false\,\wedge\,\neg\phi]\,\vee\,EG\neg\phi) \Leftrightarrow \\
AF\phi & \equiv & \neg\,(E\,[\neg\phi\,U\,false]\,\vee\,EG\neg\phi) \Leftrightarrow \\
AF\phi & \equiv & \neg\,(false\,\vee\,EG\neg\phi) \Leftrightarrow \\
AF\phi & \equiv & \neg EG\neg\phi
\end{array}$$

□

$$\begin{array}{rcl}
AG\phi & \equiv & \neg EF\neg\phi \Leftrightarrow^{(EF\phi\equiv E[\top\,U\,\phi])} \\
AG\phi & \equiv & \neg E\,[\top\,U\,\neg\phi]
\end{array}$$

□

$$\begin{array}{rcl}
E\,[\phi_1\,B\,\phi_2] & \equiv & \neg A\,[\neg\phi_1\,U\,\phi_2] \Leftrightarrow^{(A[\phi_1\,U\,\phi_2]\equiv\neg(E[\neg\phi_2\,U\,\neg\phi_1\,\wedge\,\neg\phi_2]\,\vee\,EG\neg\phi_2))} \\
E\,[\phi_1\,B\,\phi_2] & \equiv & \neg\neg\,(E\,[\neg\phi_2\,U\,\neg\neg\phi_1\,\wedge\,\neg\phi_2]\,\vee\,EG\neg\phi_2) \Leftrightarrow \\
E\,[\phi_1\,B\,\phi_2] & \equiv & E\,[\neg\phi_2\,U\,\phi_1\,\wedge\,\neg\phi_2]\,\vee\,EG\neg\phi_2
\end{array}$$

□

The above definitions are given for predicates, the appropriate definitions and the proof for concepts are analogous.

6.2 Specification

Since there will be no major new components, but rather added functionality for old ones, there will be no new packages. The model checking algorithm will make use of the interfaces `GeneralFormula` and `GeneralModel`, both of which reside in the package `ALCCTL`.

6.2.1 Formula

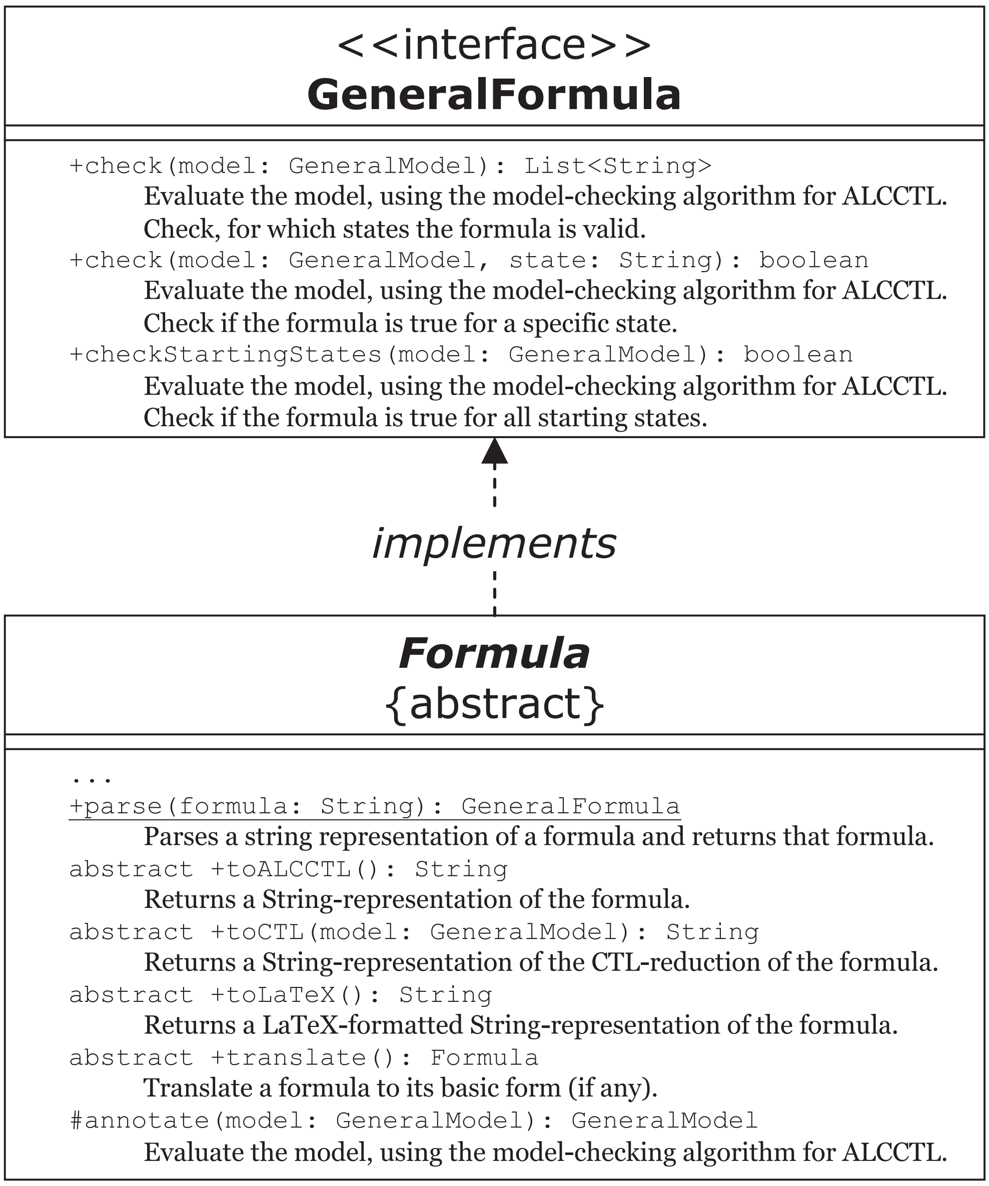
The interface `GeneralFormula` has already been mentioned in chapter 4.3.1. Model checking in general can mean to check whether $M, s \models \phi$ for all $s \in S$, or just for some specific s, usually the starting states ($M = (S, \longrightarrow, I, \Delta^I)$). The interface `GeneralFormula` reflects these alternatives. It defines three methods: One for checking the formula against all states of a supplied `GeneralModel`, one for checking one specific state of the model, and one method for checking all designated starting states of the model.
Figure 6.1 shows an overview of the formula implementation.

In principle, the `check` method that does the actual model checking could have been moved to the `GeneralModel` interface. Why hasn't it been? The answer is simple, because model checking is inductively defined over the structure of the formula, so the algorithm primarily has to traverse the formula tree, not the model.

The process of model checking works like this: One of a formula's `check` methods is called. This method initialises the model checking, for example by discarding old counter example information (see below) from a previous check, and by reducing the entire formula against the base operators. It then calls its own `annotate` method, which in turn calls the `annotate` method of its parameter(s) before doing anything else. This continues down the entire formula tree to the leafs (atomic concepts and such). From there, the annotation returns step by step upwards, annotating each subformula to the states of the model in each step. The last step annotates the entire formula, thus those states where it is annotated are those where the formula holds.

6.3 Implementation

The structure of the implementation for the algorithmic approach can be seen in figure 6.2.



```
+check(model: GeneralModel): List<String>
        Evaluate the model, using the model-checking algorithm for ALCCTL.
        Check, for which states the formula is valid.
+check(model: GeneralModel, state: String): boolean
        Evaluate the model, using the model-checking algorithm for ALCCTL.
        Check if the formula is true for a specific state.
+checkStartingStates(model: GeneralModel): boolean
        Evaluate the model, using the model-checking algorithm for ALCCTL.
        Check if the formula is true for all starting states.
```

```
...
+parse(formula: String): GeneralFormula
        Parses a string representation of a formula and returns that formula.
abstract +toALCCTL(): String
        Returns a String-representation of the formula.
abstract +toCTL(model: GeneralModel): String
        Returns a String-representation of the CTL-reduction of the formula.
abstract +toLaTeX(): String
        Returns a LaTeX-formatted String-representation of the formula.
abstract +translate(): Formula
        Translate a formula to its basic form (if any).
#annotate(model: GeneralModel): GeneralModel
        Evaluate the model, using the model-checking algorithm for ALCCTL.
```

Figure 6.1: Overview of the Formula Implementation.

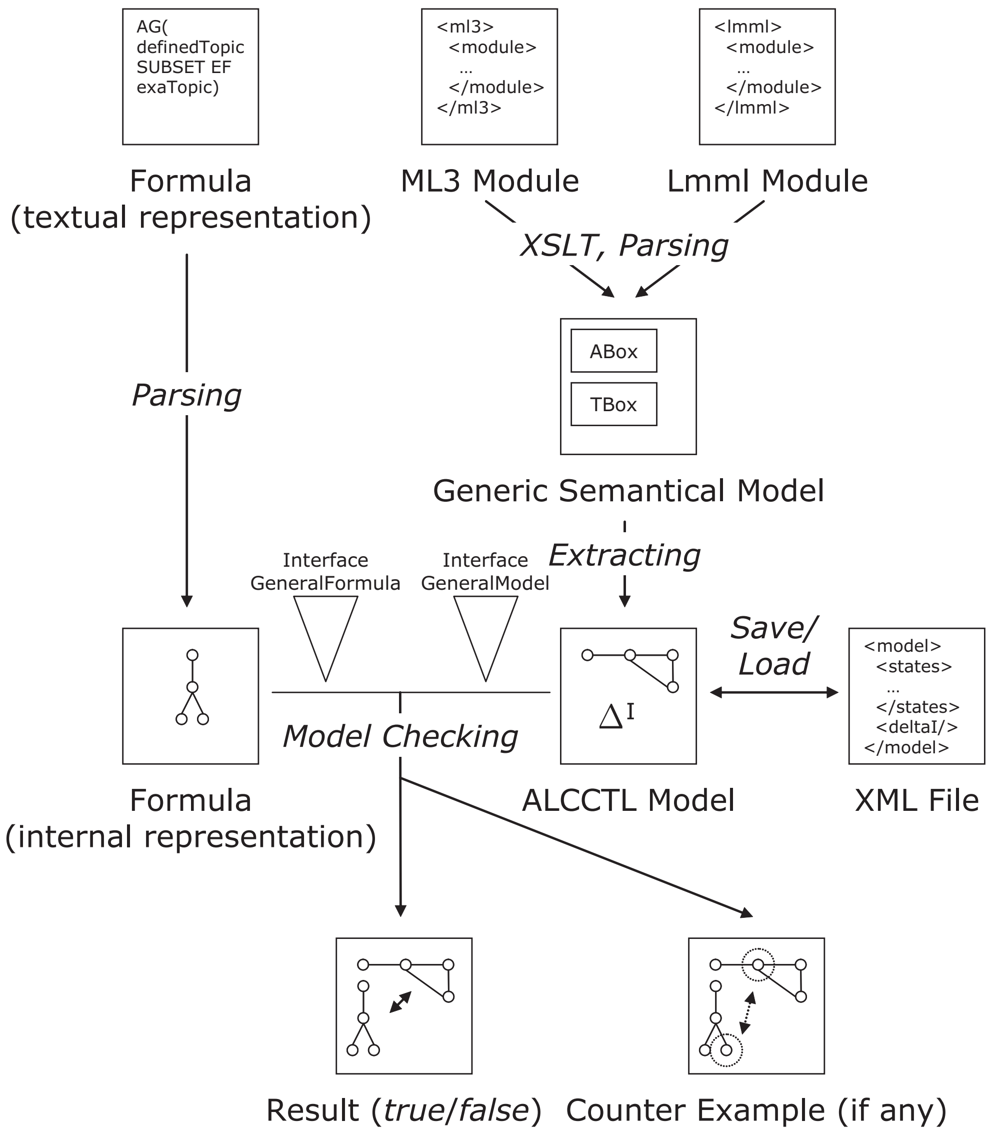


Figure 6.2: Diagram of the algorithmic model checking process.

6.3.1 Formula

In implementing the formula, there was one decision of principle to make concerning the temporal operators. It was the decision of implementing the set $\{AB, AF, AG, AU, AX, EB, EF, EG, EU, EX\}$ of operators, versus implementing the set $\{A, E, B, F, G, U, X\}$ of operators and combining them to the full list above. I implemented a few operators in both concepts (namely, AU, EU, and A, E, U) for direct comparison. In the end, the decision turned towards implementing the operators as doubles (like EU or EG) instead of combining them (from e.g. E, U or G) almost by default – the second approach would have been too inefficient. There are several reasons for that. For one, using the base reduction described in chapter 6.1.1 there are in fact far fewer operators to implement: Three for the first approach, but four for the second. Another reason is that splitting the operators would have made the base reduction (specifically the `transform` method) more complex: There would have had to be a long list of case differentiation to find out exactly how to reduce a certain operator, because it would always depend on its argument. But finally, and most importantly, the implementation would have been far less efficient. For example, splitting EU into E and U will result in two unnecessary loops that could easily have been avoided. They are caused by redundancy between E and U – both need to look at future states to determine their respective result.

It is noteworthy that, owing to the base reduction of the temporal operators, it was possible to provide the user with the full set of them – especially the AB and EB operators are otherwise often left out and have to be manually emulated by rephrasing the formula.

One useful optimisation in the model checking process is that the algorithm keeps track of which subformulas where already annotated to the model, so that duplicate subformulas can be ignored.

Table 6.1 exemplifies the implementations of `annotate` (the method that does the actual model checking) for the EG concept.

6.3.2 Model

Usually, a model has a single starting state. This is true not out of necessity, but more often out of habit. Multiple starting states can make structural or algorithmic definitions more complex, and most of the time there simply is no need for more than one starting state. E-Learning modules, however, can very well have multiple starting states. The WWR Fuzzy module, for instance, has a slide variant that is completely separate from the online variant, thus resulting in two distinct starting states: One for each variant. Adding an

```
private List<String> checkFuture(String c, GeneralModelState s) {
  List<String> result = s.interpret(c);
  // termination case: if s has already been visited, do not recurse
  // further but return the result that was found the last time around
  if (s.getMarked())
    return Utils.cloneStringList(s.getFormerResultSet());
  // mark as visited
  s.setMarked(true);
  // save the current result
  s.setFormerResultSet(Utils.cloneStringList(result));
  // if the current result is not empty, recurse further
  // if it is empty, EG (namely, G) is does not hold anyway
  if (result.size() > 0)
    for (int j=0;j<s.getSuccessors().size();j++)
      result.addAll(checkFuture(c, s.getSuccessors().get(j)));
  // save the current result again: if we come across this state again,
  // we do not have to run the entire recursion again!
  // (note that s can be the successor of more than one state)
  s.setFormerResultSet(Utils.cloneStringList(result));
  return result;
}

protected GeneralModel annotate(GeneralModel model) {
  model = arg.annotate(model);
  String curALCCTL = toALCCTL();
  if (model.isAnnotated(curALCCTL))
    return model;
  model.addAnnotation(curALCCTL);
  String argALCCTL = arg.toALCCTL();
  for (int i=0;i<model.getStates().size();i++) {
    for (int j=0;j<model.getStates().size();j++)
      model.getStates().get(j).setMarked(false);
    model.getStates().get(i).getInterpretations().put(curALCCTL,
      checkFuture(argALCCTL, model.getStates().get(i)));
  }
  Debug.debug(curALCCTL, Debug.LV_EVAL);
  Debug.debug(model, Debug.LV_EVAL);
  model.stepCounterExample(curALCCTL);
  return model;
}
```

Table 6.1: `ConceptEG.annotate`

artificial starting state that branches off to those two is not a good option: There could be no meaningful interpretations of concepts, roles or predicates at that state, thus wrecking havoc with the verification of formulas! On the other hand, allowing more than one starting state, and defining the model checking problem accordingly, creates no big problems. Every state now needs to have a flag that indicates whether it is a starting state or not (rather than holding a reference to a single starting state in the model). The algorithm itself does not have to be changed, only the method that checks all starting states needs to take this into account.

As stated in chapter 5.3.2, the list of states is implemented as an `ArrayList`. The list of concept interpretations, however, is implemented as a `HashMap<String, List<String>>`, where the `String` is the interpreted concept, and the `List<String>` is the set of objects from Δ^I. Since the lookup operation is the one most often used on the interpretations, a hashmap is an efficient way to implement it. This argumentation is also valid for predicates (`HashSet<String>`) and roles (`HashMap<String, GeneralModelRole>`, where the `String` is the name of the role). `ModelRole` itself uses a `HashMap` as well to represent the relations between concepts.

Figure 6.3 shows a sequence diagram of the $\mathcal{ALC}$CTL model checking process.

6.4 Counter Example

When actually using the algorithm, a major problem quickly arises: When a given formula – unexpectedly – turns out not to hold for a given model, it is rather tricky to find out why! The debugging mechanism already mentioned in chapter 4.3.4 in combination with log files can be used to trace the problem, but for long formulas and/or large models that is a long and tedious task indeed. It would be nice if the system would simply present a counter example of where the likely crunch is.
Let us regard an example of how something like that might work. Recall the model from figure 2.5, chapter 2.4. The formula $defTopic \dot{\sqsubseteq} AX\, exaTopic$ would not hold against that model, because the AX would require that there be an example in *each* successor state of s_{Def}, not just in one. Which is exactly what the counter example should show! It might actually be a message like "The formula does not hold because $defTopic^{I(s_{Def})} = \{DFA\}$ and $(AX\, exaTopic)^{I(s_{Def})} = \emptyset$, so $\{DFA\}$ is not a subset of $\emptyset$ in state s_{Def}".

Most of the time it is very hard to find a useful counter example. Regard, for example, the formula $defTopic \dot{\sqsubseteq} refTopic$. It will probably never hold, because it is unlikely that there will be a reference to a definition in the very

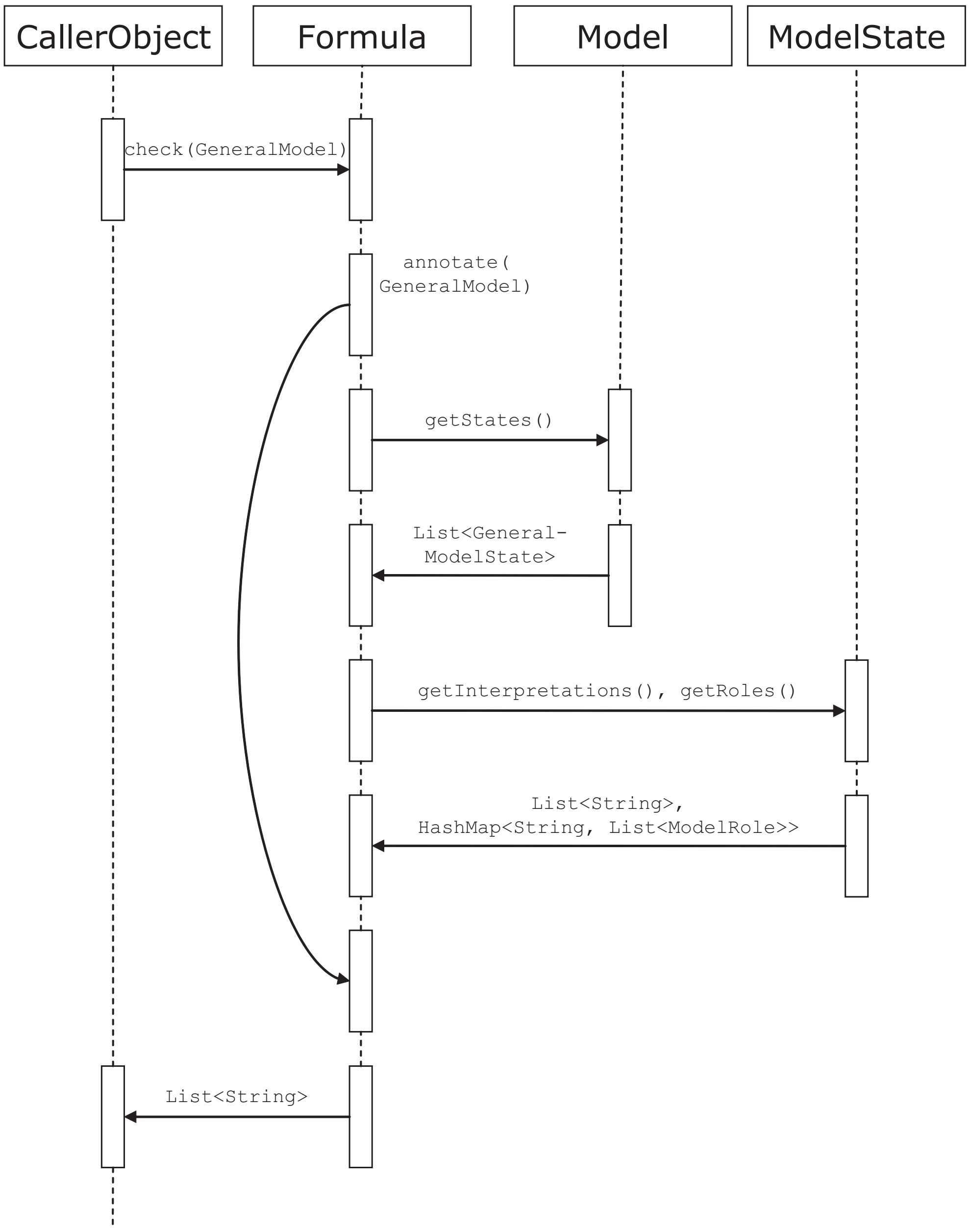


Figure 6.3: Sequence diagram of the $\mathcal{ALC}$CTL model checking process.

paragraph *holding* the definition. Thus, a suitable counter example would be the set of *all* states, which, of course, could be rather unwieldy. Simply reducing the counter example to provide a single state would be equally unhelpful in other cases, however.
Another problem is *finding* the point of failure in the first place. A formula can have almost infinite length and complexity, and the actual reason why it does not hold could be at any part of it. Simply returning all subformulas for all states where they interpret to $\emptyset$ or to $false$ clearly is of no use whatsoever. Negating the entire formula, that is, checking where the formula does *not* hold, is a simple approach to that problem. CTL model checkers do just that [CCGR00, McM93]. But with $\mathcal{ALC}$CTL most of the time the relevant point of failure is where the $\dot{\sqsubseteq}$ or $\doteq$ relation fails: In the current implementation that is what the counter example checks for, with encouraging results.
But even if the relevant point of failure is found, it is a point in the base reduced formula which does not necessarily have any resemblance to the original formula left: Therefore, a user might find himself with a counter example to a formula he has never seen before! A possible solution would be to attempt some kind of re-mapping, extending the base reduced formula back to its original form (or at least as close as possible). Another option is to forgo the base reduction entirely.

In the current implementation, the model checking process stores each step of the way, for future use of the class `CounterExample`. If the model check fails, the counter example is displayed by calling this class's `toString()` method. Table 6.2 lists parts of that method. It is divided in three sections. The first section tries to find the point of failure – that is, the step of the $\dot{\sqsubseteq}$ or $\doteq$ operator. The second step lists all states where this operation failed. In the final step, all these states are displayed, along with the local interpretations of the subformulas of the failed operation.

The counter example issue has been addressed by several authors, [Ki96, CGMZ94] among others. However, they all refer to symbolic model checking based on ordered binary decision diagrams (OBDDs). In symbolic model checking, sets of states are symbolised by the OBDD of a Boolean function.

An ordered binary decision *diagram* can be used as a compact representation of a Boolean formula. An ordered binary decision *tree* is a structure that introduces a new variable on each level of the tree. Each left subtree assumes that the variable is $false$, while each right subtree assumes that the variable is *true*. The leafs hold the values of the Boolean formula for the variable assignments of each respective path. An OBDD is an OBDT where all isomorphic subtrees are merged and duplicate nodes are left out. Logical operators can be applied directly to an OBDD.
The state transition relation of a model can be represented as a Boolean

```
public String toString() {
  String result = "";
  int stepNumber = -1;
  // find the first annotation for SUBSET or EQUALS (globally)
  ...
  if (stepNumber == -1)
    result = "No SUBSET or EQUALS found! No counterexample available.";
  else {
    List<String> states = new ArrayList<String>();
    // add all states where the SUBSET or EQUALS relation is false
    ...
    if (states.size() == 0)
      result = "Everything seems to be in order. No CE found.";
    else {
      String step = model.getCounterExampleSteps().get(stepNumber);
      // find out which operator is relevant
      String operator = "EQUALS";
      if (step.contains("SUBSET"))
        operator = "SUBSET";
      result = "The "+operator+" predicate was [false] in ...\n";
      for (int i=0;i<states.size();i++) {
        // display each state where the releation is false,
        // and the relevant sets (interpretations)
        ...
      }
    }
  }
  return result;
}
```

Table 6.2: `CounterExample.toString()` – Extraction of the Counter Example.

function $\longrightarrow (v_{s_0}, v_{s_1})$, where v_s is the complete set of variables with values as in state s, and thus as an OBDD.
Formulas can now be solved as combinations of such diagrams [Cla03].

NuSMV employs, as does its predecessor SMV, ODBBs for model checking. A counter example with this method can usually be found by negating the last step, that is, negating the lowest subtree of the complete OBDD. But even though [BCCZ99] proposes alternate methods of symbolic model checking, such as Stålmarck's Method or the Davis & Putnam Procedure, all of these approaches cannot be used to find counter examples for $\mathcal{ALC}$CTL model checking: The procedure simply is too different [CCGR00, CCGR+02, McM93, BCM+92].

6.5 Complexity

Let $M = (S, \longrightarrow, I, \Delta^I)$ be an $\mathcal{ALC}$CTL model, and let ϕ be an $\mathcal{ALC}$CTL formula. $|\phi|$ denotes the "size" of the formula, that is, the number of independent subformulas of ϕ. When calling `GeneralFormula.check` for an $\mathcal{ALC}$CTL model M and a state s, the check is done for the entire model, and then s is tried to be located in the result set. Since the result set is an `ArrayList`, and its `contains` method scales in a linear fashion, the complexity is $O(|S|)$.

```
// Formula.check(GeneralModel, String state)
List<String> states = check(model);
// ArrayList.contains is in linear time: |S|
return states.contains(state);
```

Collecting this result set required iterating over all states and checking if the formula is annotated to them. The iteration costs $|S|$, obviously, leaving the complexity unchanged with $O(|S| \cdot 2) = O(|S|)$. Since the predicates are implemented as a `HashSet`, the `contains` method of which only takes constant time, looking for the annotation does not increase the complexity.

```
// Formula.check(GeneralModel)
// base reduction
Formula tmp = translate();
// ALCCTL string of the current formula
String curALCCTL = tmp.toALCCTL();
...
model = tmp.annotate(model);
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  // HashSet.contains is in constant time
  if (model.getStates().get(i).getPredicates().contains(curALCCTL))
    result.add(model.getStates().get(i).getName());
}
return result;
```

Now, all that is left to do is to annotate every subformula of ϕ to the states of M. Thus, $M, s \models \phi \in O\left(|S| + |\phi| \cdot op\right)$ for a formula ϕ, where op is the maximum complexity class of all $\mathcal{ALC}$CTL operators: $op =_{def} O\left(\wedge\right) + O\left(\neg\right) + O\left(\sqcup\right) + O\left(\dot{\sqsubseteq}\right) + O\left(EG\right) + O\left(EU\right) + O\left(\exists\right) + O\left(\forall\right) + \ldots = O\left(\max\left(\wedge, \neg, EG, \exists, \ldots\right)\right)$.

To narrow down op, let's look at some of those operators more closely. The Boolean operators like $\wedge$ or $\vee$ only require constant time for each state, so we can safely disregard them when looking for the maximum. The operators $\dot{\sqsubseteq}$ and $\doteq$, as well as $\sqcap$ and $\sqcup$ are all in $|S| \cdot |\Delta^I|^2$. The complexities of the three base temporal operators EX, EG and EU are all in $O(|S|^2 \cdot |\Delta^I|)$ for concepts.
The other temporal operators are also within that complexity class, even though they are translated to a combination of several operators. I will show this for the operator AF, the proof for the rest is analogous: Since $AF\psi = \neg EG\neg\psi$ for a concept ψ, $O(AF) = O(\neg EG\neg)$. But $O(\neg EG\neg) = O(\neg) + O(EG) + O(\neg)$, and $(O(\neg) + O(EG) + O(\neg)) \in O(EG)$! Therefore, $O(AF) \in O(EG)$.
The last two operators that require attention are the role operators $\exists$ and $\forall$. They have a complexity of $O(|S| \cdot |\Delta^I|^3)$ and as such the maximum complexity of all operators, for $|\Delta^I|$ is usually greater than $|S|$ by a factor of two to six. This leads to a total complexity of $|S| + |\phi| \cdot |S| \cdot |\Delta^I|^3 = |\phi| \cdot |S| \cdot |\Delta^I|^3$.

```
// FormulaAnd:
String arg1ALCCTL = arg1.toALCCTL();
String arg2ALCCTL = arg2.toALCCTL();
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++)
  // HashSet methods 'contains' and 'add' require only constant time
  if (model.getStates().get(i).getPredicates().contains(arg1ALCCTL)
    && model.getStates().get(i).getPredicates().contains(arg2ALCCTL))
    model.getStates().get(i).getPredicates().add(curALCCTL);

// ConceptEU:
// aux. method:
```

```
private boolean checkUntil(String a, String b, String concept,
                           GeneralModelState s) {
  if (s.getMarked()) return false; // termination case
  s.setMarked(true);
  // ModelState.interpret only requires constant time,
  // as does HashSet.contains
  if (s.interpret(b).contains(concept))
    return true;
  boolean tmp = false;
  if (s.interpret(a).contains(concept))
    // continue with the recursion; but |S| has already been counted
    for (int j=0;j<s.getSuccessors().size();j++)
      tmp = tmp || checkUntil(a, b, concept, s.getSuccessors().get(j));
  return tmp;
}

protected GeneralModel annotate(GeneralModel model) {
  String curALCCTL = toALCCTL();
  String arg1ALCCTL = arg1.toALCCTL();
  String arg2ALCCTL = arg2.toALCCTL();
  // iterate over all states: |S|
  for (int i=0;i<model.getStates().size();i++) {
    List<String> tmp = new ArrayList<String>();
    // iterate over DeltaI: |DeltaI|
    for (int j=0;j<model.getDeltaI().size();j++) {
      // iterate over all states: |S|
      // this is at most as complex as the recursive call, however,
      // so it does not influence the overall complexity
      for (int k=0;k<model.getStates().size();k++)
        model.getStates().get(k).setMarked(false);
      String concept = model.getDeltaI().get(j);
      // the recursion runs over all states, but terminates if
      // one state is visited twice: |S|
      if (checkUntil(arg1ALCCTL, arg2ALCCTL, concept,
          model.getStates().get(i)))
        // ArrayList.add ist constant
        tmp.add(concept);
    }
    // HashMap.put ist constant
    model.getStates().get(i).getInterpretations().put(curALCCTL, tmp);
  }
  return model;
}

// FormulaEU:
// aux. method:
private boolean checkUntil(String a, String b, GeneralModelState s) {
  if (s.getMarked()) return false;
  s.setMarked(true);
  // constant time
  if (s.getPredicates().contains(b))
    return true;
  boolean tmp = false;
  if (s.getPredicates().contains(a)) {
    // continue with the recursion; but |S| has already been counted
    for (int j=0;j<s.getSuccessors().size();j++)
      tmp = tmp || checkUntil(a, b, s.getSuccessors().get(j));
  }
  return tmp;
}

protected GeneralModel annotate(GeneralModel model) {
  String curALCCTL = toALCCTL();
  String arg1ALCCTL = arg1.toALCCTL();
  String arg2ALCCTL = arg2.toALCCTL();
  // iterate over all states: |S|
  for (int i=0;i<model.getStates().size();i++) {
    // iterate over all states: |S|
    // this is at most as complex as the recursive call, however,
    // so it does not influence the overall complexity
    for (int j=0;j<model.getStates().size();j++)
      model.getStates().get(j).setMarked(false);
    // the recursion runs over all states, but terminates if
    // one state is visited twice: |S|
    if (checkUntil(arg1ALCCTL, arg2ALCCTL, model.getStates().get(i)))
      // HashSet.add ist constant
      model.getStates().get(i).getPredicates().add(curALCCTL);
  }
  return model;
}

// FormulaSubset:
```

```
String arg1ALCCTL = arg1.toALCCTL();
String arg2ALCCTL = arg2.toALCCTL();
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  boolean all = true;
  // ModelState.interpret only requires constant time:
  // It calls HashMap.get, which requires constant time.
  List<String> a = model.getStates().get(i).interpret(arg1ALCCTL);
  List<String> b = model.getStates().get(i).interpret(arg2ALCCTL);
  // iterate over (possibly all of) DeltaI: |DeltaI|
  for (int j=0;j<a.size();j++)
    // ArrayList.contains is linear: |DeltaI|
    if (!b.contains(a.get(j)))
      all = false;
  // HashSet.add ist constant
  if (all)
    model.getStates().get(i).getPredicates().add(curALCCTL);
}

// ConceptAnd:
String arg1ALCCTL = arg1.toALCCTL();
String arg2ALCCTL = arg2.toALCCTL();
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  // ModelState.interpret only requires constant time:
  // It calls HashMap.get, which requires constant time.
  List<String> a = model.getStates().get(i).interpret(arg1ALCCTL);
  List<String> b = model.getStates().get(i).interpret(arg2ALCCTL);
  // ArrayList.retainAll requires n^2: |DeltaI|^2
  a.retainAll(b);
  model.getStates().get(i).getInterpretations().put(curALCCTL, a);
}

// ConceptForall:
// Semantic: FORALL R.C^I(s) =_def {a ELEMENT DeltaI |
//   FORALL b.(a,b) ELEMENT R^I(s) -> b ELEMENT C^I(s)}
String argALCCTL = arg.toALCCTL();
// iterate over all states: |S|
for (int i=0;i<model.getStates().size();i++) {
  List<String> result = new ArrayList<String>();
  // ModelState.interpret only requires constant time:
```

```
// It calls HashMap.get, which requires constant time.
List<String> C = model.getStates().get(i).interpret(argALCCTL);
// HashMap.get requires only constant time
GeneralModelRole role = model.getStates().get(i).getRoles().get(name);
// The role might be unknown in this state - but since it can be known
// in other states, this is not an error! So simply check for it.
if ((role != null) && (C.size() > 0)) {
  boolean all = true;
  // interate over DeltaI: |DeltaI|
  for (int j=0;j<model.getDeltaI().size();j++) {
    String a = model.getDeltaI().get(j);
    Set<String> B = role.get(a);
    // left-hand-side of the implication is false:
    // implication itself is true
    if (B == null) {
      result.add(a);
    } else {
      Iterator<String> bs = B.iterator();
      // iterate over (possibly all of) DeltaI: |DeltaI|
      while (bs.hasNext()) {
        String b = bs.next();
        // ArrayList.contains requires linear time: |DeltaI|
        if (!C.contains(b)) {
          all = false;
          break;
        } } if (all) result.add(a);
    } if (!all) break;
  }
  if (all)
    model.getStates().get(i).getInterpretations().put(
      curALCCTL, result);
} }
```

Proposition 4 (Complexity of the $\mathcal{ALC}$CTL Model Checking Algorithm) *$M, s \models \phi \in O\left(|\phi| \cdot |S| \cdot |\Delta^I|^3\right)$ (the complexity of checking ϕ against M in state s).*

By changing the result type of the `ModelState.interpret` method from `ArrayList` to `HashSet`, the complexity could be decreased to $O\left(|\phi| \cdot |S| \cdot |\Delta^I|^2\right)$. However, this would make iterating over the result more expensive (even though still constant). And since it is highly unrealistic that the iteration would actually require $\Delta^I \times \Delta^I \times \Delta^I$ in its entirety, the real-world performance would decrease, not increase. Thus, I have decided to accept the higher complexity in exchange for better run-time results.

Since model checking CTL is P-complete with an upper bound of $O(|S| \cdot |\phi|)$ [Sch02], and the context satisfiability problem in $\mathcal{ALC}$ is ExpTime-complete [Sch91, Don02] with an upper bound exponential time in the size of ϕ and the TBox ($\equiv |\Delta^I|$) [DM00], it seems unlikely that the complexity of the $\mathcal{ALC}$CTL model checking algorithm could be decreased significantly. Even though the context satisfiability problem is more complex than the model checking problem, it is clear that model checking $\mathcal{ALC}$CTL has to depend on the size of Δ^I at some point.
$\mathcal{ALC}$CTL can be reduced to CTL, so the complexity of model checking there is a lower bound for $\mathcal{ALC}$CTL. $\mathcal{ALC}$CTL also includes roles and set relations with a domain of $\Delta^I \times \Delta^I$, which suggests a complexity close to that of proposition 4.

6.6 Tests

The test configuration and testing methods are the same as described in chapter 5.5. The test performed was to check every model against 27 formulas, using the model checking algorithm for $\mathcal{ALC}$CTL and the command-line tool ALCCTLModelChecking.

Table 6.3 lists the base reduced formulas into which the formulas that where actually checked were transformed in the first step. The model checking time for each model can be seen in table 6.5, while the memory requirements are displayed in table 6.4. Table 6.6 shows the model checking results. In tables 6.7 to 6.11 the counter examples are listed.
Graphs that show how the algorithmic approach scales with respect to the size of the model or the number of concepts are shown in figures 7.4 and 7.5, chapter 7. They compare the scaling of the two model checking approaches.

In contrast to the reduction approach (see chapter 5), the $\mathcal{ALC}$CTL model checking algorithm did produce a result for every test formula. However, additional testing with formulas containing the operator AF lead to a Java exception `OutOfMemoryError`: Java heap space. Unfortunately, even increasing the maximum heap size to 1.5 GB (!) did not solve this problem. Reformulating the formula, or, better yet, implementing the AF operator directly without resorting to the base reduction, would probably prove more effective.

Many of the counter examples listed are shortened: Since the entire result sets of the failed relation are produced, they can get quite lengthy. The

Number	Base reduced formula
Formula 1	$\bot \mathrel{\dot\sqsubseteq} \top$
Formula 2	$EG(\bot \mathrel{\dot\sqsubseteq} \top)$
Formula 3	$EX(\bot \mathrel{\dot\sqsubseteq} \top)$
Formula 4	$E\left[true\, U\, \bot \mathrel{\dot\sqsubseteq} \top\right]$
Formula 5	$\bot \mathrel{\dot\sqsubseteq} EG\bot$
Formula 6	$\bot \mathrel{\dot\sqsubseteq} EX\top$
Formula 7	$\bot \mathrel{\dot\sqsubseteq} E\,[\top\, U\, \top]$
Formula 8	$true$
Formula 9	$\neg true$
Formula 10	$true\ \wedge\ true$
Formula 11	$true\ \wedge\ false$
Formula 12	$true\ \vee\ false$
Formula 13	$false\ \vee\ false$
Formula 14	$\top\ \doteq\ \top$
Formula 15	$\top\ \doteq\ \bot$
Formula 16	$definedTopic \mathrel{\dot\sqsubseteq} E\,[\top\, U\, (exemplifiedTopic\ \sqcap\ \top)]$
Formula 17	$definedTopic \mathrel{\dot\sqsubseteq} E\,[\top\, U\, (exemplifiedTopic\ \sqcup\ \neg\top)]$
Formula 18	$definedTopic \mathrel{\dot\sqsubseteq} EX\, exemplifiedTopic$
Formula 19	$definedTopic \mathrel{\dot\sqsubseteq} E\,[\top\, U\, exemplifiedTopic]$
Formula 20	$definedTopic \mathrel{\dot\sqsubseteq} E\,[definedTopic\, U\, exemplifiedTopic]$
Formula 21	$definedTopic \mathrel{\dot\sqsubseteq} \neg E\,[\top\, U\, \neg exemplifiedTopic]$
Formula 22	$\neg E\left[true\, U\, \neg\left(definedTopic \mathrel{\dot\sqsubseteq} EX exemplifiedTopic\right)\right]$
Formula 23	$\neg E\left[true\, U\, \neg\left(definedTopic \mathrel{\dot\sqsubseteq} E\,[\top\, U\, exemplifiedTopic]\right)\right]$
Formula 24	$\neg E\left[true\, U\, \neg\left(definedTopic \mathrel{\dot\sqsubseteq} E\,[definedTopic\, U\, exemplifiedTopic]\right)\right]$
Formula 25	$\neg E\left[true\, U\, \neg\left(definedTopic \mathrel{\dot\sqsubseteq} \neg E\,[\top\, U\, \neg exemplifiedTopic]\right)\right]$
Formula 26	$\neg E\left[true\, U\, \neg\left(definedTopic \mathrel{\dot\sqsubseteq} E\,[\top\, U\, \exists topicOf.Example]\right)\right]$
Formula 27	$\neg E\left[true\, U\, \neg\left(\forall topicOf.Definition \mathrel{\dot\sqsubseteq} E\,[\top\, U\, \exists topicOf.Fragment]\right)\right]$

Table 6.3: Base Reduced Test Formulas.

TinyTest	Test	EZKom	Fuzzy	BIS
7.3mb	8.7mb	22.1mb	28.0mb	8.7mb

Table 6.4: $\mathcal{ALC}$CTL Model Checking Memory Usage (usage measured for formula 1).

	TinyTest	Test	EZKom	Fuzzy	BIS
Formula 1	47ms	47ms	1s 250ms	1s 828ms	62ms
Formula 2	47ms	63ms	1s 500ms	2s 234ms	78ms
Formula 3	16ms	46ms	1s 516ms	2s 219ms	78ms
Formula 4	47ms	62ms	1s 781ms	2s 625ms	94ms
Formula 5	31ms	46ms	1s 234ms	1s 703ms	63ms
Formula 6	15ms	62ms	1s 531ms	2s 281ms	78ms
Formula 7	31ms	47ms	22s 578ms	52s 656ms	125ms
Formula 8	15ms	32ms	610ms	953ms	47ms
Formula 9	0ms	46ms	922ms	1s 328ms	46ms
Formula 10	15ms	47ms	969ms	1s 360ms	62ms
Formula 11	31ms	46ms	1s 344ms	1s 750ms	62ms
Formula 12	15ms	46ms	1s 282ms	1s 750ms	62ms
Formula 13	16ms	46ms	984ms	1s 344ms	63ms
Formula 14	16ms	47ms	4s 907ms	10s 16ms	62ms
Formula 15	31ms	46ms	1s 219ms	1s 781ms	63ms
Formula 16	47ms	78ms	26m 6s 797ms	54m 25s 516ms	859ms
Formula 17	47ms	78ms	26m 9s 297ms	54m 36s 750ms	985ms
Formula 18	32ms	47ms	1s 453ms	2s 140ms	78ms
Formula 19	31ms	78ms	25m 55s 578ms	54m 16s 203ms	843ms
Formula 20	16ms	63ms	2s 672ms	5s 875ms	93ms
Formula 21	47ms	78ms	26s 344ms	59s 390ms	172ms
Formula 22	31ms	63ms	2s 468ms	3s 844ms	125ms
Formula 23	47ms	109ms	26m 17s 171ms	55m 6s 250ms	969ms
Formula 24	31ms	78ms	3s 907ms	7s 875ms	141ms
Formula 25	47ms	94ms	25s 875ms	58s 937ms	187ms
Formula 26	31ms	109ms	26m 9s 750ms	55m 10s 94ms	906ms
Formula 27	32ms	125ms	24m 5s 157ms	52m 27s 47ms	953ms

Table 6.5: $\mathcal{ALC}\mathsf{CTL}$ Model Checking Test Results.

	TinyTest	Test	EZKom	Fuzzy	BIS
Formula 1	$true$	$true$	$true$	$true$	$true$
Formula 2	$true$	$true$	$true$	$true$	$true$
Formula 3	$true$	$true$	$true$	$true$	$true$
Formula 4	$true$	$true$	$true$	$true$	$true$
Formula 5	$true$	$true$	$true$	$true$	$true$
Formula 6	$true$	$true$	$true$	$true$	$true$
Formula 7	$true$	$true$	$true$	$true$	$true$
Formula 8	$true$	$true$	$true$	$true$	$true$
Formula 9	$false$	$false$	$false$	$false$	$false$
Formula 10	$true$	$true$	$true$	$true$	$true$
Formula 11	$false$	$false$	$false$	$false$	$false$
Formula 12	$true$	$true$	$true$	$true$	$true$
Formula 13	$false$	$false$	$false$	$false$	$false$
Formula 14	$true$	$true$	$true$	$true$	$true$
Formula 15	$false$	$false$	$false$	$false$	$false$
Formula 16	$true$	$true$	$true$	$true$	$true$
Formula 17	$true$	$true$	$true$	$true$	$true$
Formula 18	$true$	$true$	$true$	$true$	$true$
Formula 19	$true$	$true$	$true$	$true$	$true$
Formula 20	$true$	$true$	$true$	$true$	$true$
Formula 21	$true$	$true$	$true$	$true$	$true$
Formula 22	$true$	$true$	$false$	$false$	$true$
Formula 23	$true$	$true$	$false$	$false$	$true$
Formula 24	$true$	$true$	$false$	$false$	$true$
Formula 25	$false$	$false$	$false$	$false$	$true$
Formula 26	$true$	$true$	$false$	$false$	$true$
Formula 27	$false$	$false$	$true$	$false$	$true$

Table 6.6: $\mathcal{ALC}$CTL Model Checking Results.

	Counter Example
Formula 15	The $\doteq$ predicate was [false] in the following states ($\neg \top \doteq \bot$): Intro: $\neg$ {FormScript, Def, GroupStudent, ...} $\doteq \emptyset$ Def: $\neg$ {FormScript, Def, GroupStudent, ...} $\doteq \emptyset$ Exa: $\neg$ {FormScript, Def, GroupStudent, ...} $\doteq \emptyset$ Conc: $\neg$ {FormScript, Def, GroupStudent, ...} $\doteq \emptyset$
Formula 25	The $\dot\sqsubseteq$ predicate was [false] in the following states ($\neg$ definedTopic $\dot\sqsubseteq$ $\neg$ E[$\top$ U $\neg$ exemplifiedTopic]): Def: $\neg$ {DFA} $\dot\sqsubseteq \emptyset$

Formula 27	The $\dot\sqsubseteq$ predicate was [false] in the following states
	($\neg \forall$ topicOf.Definition $\dot\sqsubseteq$ E[$\top$ U $\exists$ topicOf.Fragment]):
	Def: $\neg$ {FormScript, Def, GroupStudent, ...} $\dot\sqsubseteq$ {DFA}

Table 6.7: $\mathcal{ALC}$CTL Model Checking Counter Examples: TinyTest.

counter examples for formulas 9, 11 and 13 were omitted, they all read "No SUBSET or EQUALS found! No counterexample available.", since the current implementation tries to find the point of failure at the $\dot\sqsubseteq$ or $\doteq$ operator. It would be possible to adopt a behaviour where the counter example negates the entire formula in such a case.

6.7 Evaluation

Quite similar to the findings in chapter 5.6, figure 6.4 shows the relation between average model checking time (mean over all formulas) and the number of successors, interpretations and roles. This relation obviously not only holds for CTL model export, but for $\mathcal{ALC}$CTL model checking as well: Since both have similar dependencies, this is not unexpected.

The actual speed of $\mathcal{ALC}$CTL model checking is summed up in figure 6.5. Six formulas stand out immediately: Formulas 16, 17, 19, 23, 26 and 27. The latter two are hardly surprising: Checking roles is the computationally most expensive operation of all formulas. The other four all have one thing in common: When reduced to their base, each of them contains $E\,[\top\, U\, \phi]$. The difference to the similar formulas 21 and 25 is that ϕ is not negated! Checking $\top\, U\, \phi$ is an expensive operation because $\top$ is a very large set, which has to be checked for every single state until ϕ holds – however, in formulas 21 and 25 this check has only to be done until a certain case is *not* true (specifically there should be no example). Since it is very likely that this case is indeed

	Counter Example
Formula 15	The $\doteq$ predicate was [false] in the following states ($\neg \top \doteq \bot$): L1P1: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L1P2: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L1P3: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L1P1s: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L1P2s: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L1P3s: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L2P1: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L2P1E1: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L2P1E2: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L2P1E3: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L2P2: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$ L2P3: $\neg$ {FormSlide, L1P1s, GroupStudent, ...} $\doteq \emptyset$
Formula 25	The $\dot\sqsubseteq$ predicate was [false] in the following states ($\neg$ definedTopic $\dot\sqsubseteq$ $\neg$ E[$\top$ U $\neg$ exemplifiedTopic]): L2P1E1: $\neg$ {DFA} $\dot\sqsubseteq \emptyset$ L2P2: $\neg$ {NFA} $\dot\sqsubseteq \emptyset$

Formula 27	The $\dot\sqsubseteq$ predicate was [false] in the following states
	($\neg$ $\forall$ topicOf.Definition $\dot\sqsubseteq$ E[$\top$ U $\exists$ topicOf.Fragment]):
	L2P1E1: $\neg$ {FormSlide, L1P1s, ...} $\dot\sqsubseteq$ {DFA, NFA}
	L2P2: $\neg$ {FormSlide, L1P1s, ...} $\dot\sqsubseteq$ {NFA}

Table 6.8: $\mathcal{ALC}$CTL Model Checking Counter Examples: Test.

	Counter Example
Formula 15	The $\doteq$ predicate was [false] in the following states ($\neg \top \doteq \bot$): 1.1.1.1. Überblick N65662: $\neg$ {FormScript, ...} $\doteq \emptyset$ 1.1.1.4.1. Feldbusse N70680: $\neg$ {FormScript, ...} $\doteq \emptyset$ randbedingungen: $\neg$ {FormScript, ...} $\doteq \emptyset$ ...
Formula 22	The $\dot{\sqsubseteq}$ predicate was [false] in the following states ($\neg$ definedTopic $\dot{\sqsubseteq}$ EX exemplifiedTopic): Definition von Verbindungsnetzen: $\neg$ {Verbindungsnetz, Station, Paket, Nachricht, Sender, Empfänger} $\dot{\sqsubseteq}$ {Regelkreis} Klassifik. von Sendern: $\neg$ {konstante Raten, variable Raten} $\dot{\sqsubseteq} \emptyset$ ...

Formula 23	The $\dot{\sqsubseteq}$ predicate was [false] in the following states
	($\neg$ definedTopic $\dot{\sqsubseteq}$ E[$\top$ U exemplifiedTopic]):
	Definition von Verbindungsnetzen: $\neg$ {Verbindungsnetz, Station, Paket, Nachricht, Sender, Empfänger} $\dot{\sqsubseteq}$ {Ein Switch mit drei Verbindungen, Sendeanteil, Verspätung, Sendedauer, Frühe Tokenankunft, ...}
	Klassifikation von Sendern: $\neg$ {konstante Raten, variable Raten} $\dot{\sqsubseteq}$ {Ein Switch mit drei Verbindungen, Sendeanteil, ...}
	...

Formula 24	The $\dot{\sqsubseteq}$ predicate was [false] in the following states ($\neg$ definedTopic $\dot{\sqsubseteq}$ E[definedTopic U exemplifiedTopic]): Definition von Verbindungsnetzen: $\neg$ {Verbindungsnetz, Station, Paket, Nachricht, Sender, Empfänger} $\dot{\sqsubseteq} \emptyset$ Klassifikation von Sendern: $\neg$ {konstante Raten, variable Raten} $\dot{\sqsubseteq}$ {Variable Raten bei Audio- und Videodaten} ...
Formula 25	The $\dot{\sqsubseteq}$ predicate was [false] in the following states ($\neg$ definedTopic $\dot{\sqsubseteq}$ $\neg$ E[$\top$ U $\neg$ exemplifiedTopic]): Definition von Verbindungsnetzen: $\neg$ {Verbindungsnetz, Station, Paket, Nachricht, Sender, Empfänger} $\dot{\sqsubseteq} \emptyset$ Klassifikation von Sendern: $\neg$ {konst. Raten, var. Raten} $\dot{\sqsubseteq} \emptyset$ ...

Formula 26	The $\dot{\sqsubseteq}$ predicate was [false] in the following states
	($\neg$ definedTopic $\dot{\sqsubseteq}$ E[$\top$ U EXISTS topicOf.Example]):
	Definition von Verbindungsnetzen: $\neg$ {Verbindungsnetz, Station, Paket, ...} $\dot{\sqsubseteq}$ {Sendeanteil, ...}
	Klassifikation von Sendern: $\neg$ {konstante Raten, variable Raten} $\dot{\sqsubseteq}$ {Sendeanteil, Verspätung, ...}
	..

Table 6.9: $\mathcal{ALC}$CTL Model Checking Counter Examples: EZKom.

	Counter Example
Formula 15	The $\doteq$ predicate was [false] in the following states ($\neg$ $\top$ $\doteq$ $\bot$): L1_einleitung: $\neg$ {FormSlide, Irisklassifikation, ...} $\doteq$ $\emptyset$ L1_zwei_beispiele: $\neg$ {FormSlide, Irisklassifikation, ...} $\doteq$ $\emptyset$ ...

Formula 22	The $\dot\sqsubseteq$ predicate was [false] in the following states
	($\neg$ definedTopic $\dot\sqsubseteq$ EX exemplifiedTopic):
	L1_linguistische_terme: $\neg$ {linguistischer Term} $\dot\sqsubseteq$
	{Einparken eines Autos, Qualitätsbestimmung}
	L1_grundlegende_erklaerungen: $\neg$ {Fuzzy-Logik, Erweiterung der zweiwertigen Logik, ...} $\dot\sqsubseteq$ {Einparken, Qualitätsbestimmung}
	...

Formula 23	The $\dot\sqsubseteq$ predicate was [false] in the following states
	($\neg$ definedTopic $\dot\sqsubseteq$ E[$\top$ U exemplifiedTopic]):
	L1_linguistische_terme: $\neg$ {linguistischer Term} $\dot\sqsubseteq$
	{System mit einfacher Regel, charakteristische Funktion, ...}
	L1_grundlegende_erklaerungen: $\neg$ {Fuzzy-Logik, Erweiterung der zweiwertigen Logik, ...} $\dot\sqsubseteq$ {System mit einfacher Regel, ...}
	...

Formula 24	The $\dot\sqsubseteq$ predicate was [false] in the following states ($\neg$ definedTopic $\dot\sqsubseteq$ E[definedTopic U exemplifiedTopic]): L1_linguistische_terme: $\neg$ {linguistischer Term} $\dot\sqsubseteq$ $\emptyset$ L1_grundlegende_erklaerungen: $\neg$ {Fuzzy-Logik, ...} $\dot\sqsubseteq$ $\emptyset$ ...
Formula 25	The $\dot\sqsubseteq$ predicate was [false] in the following states ($\neg$ definedTopic $\dot\sqsubseteq$ $\neg$ E[$\top$ U $\neg$ exemplifiedTopic]): L1_linguistische_terme: $\neg$ {linguistischer Term} $\dot\sqsubseteq$ $\emptyset$ L1_grundlegende_erklaerungen: $\neg$ {Fuzzy-Logik, ...} $\dot\sqsubseteq$ $\emptyset$ ...
Formula 26	The $\dot\sqsubseteq$ predicate was [false] in the following states ($\neg$ definedTopic $\dot\sqsubseteq$ E[$\top$ U $\exists$ topicOf.Example]): L1_linguistische_terme: $\neg$ {linguistischer Term} $\dot\sqsubseteq$ {System mit einfacher Regel, charakteristische Funktion, ...} L1_grundlegende_erklaerungen: $\neg$ {Fuzzy-Logik, ...} $\dot\sqsubseteq$ {System mit einfacher Regel, charakteristische Funktion, ...} ...
Formula 27	The $\dot\sqsubseteq$ predicate was [false] in the following states ($\neg$ $\forall$ topicOf.Definition $\dot\sqsubseteq$ E[$\top$ U $\exists$ topicOf.Fragment]): L1_grundlegende_erklaerungen: $\neg$ {Irisklassifikation, ...} $\dot\sqsubseteq$ {Aufgabenstellung Iris-Klassifikation, Mamdani-Regler, ...} Grundlegende Erklärungen: $\neg$ {Irisklassifikation, ...} $\dot\sqsubseteq$ {Aufgabenstellung Iris-Klassifikation, ...} ...

Table 6.10: $\mathcal{ALC}$CTL Model Checking Counter Examples: Fuzzy.

	Counter Example
Formula 15	The $\doteq$ predicate was [false] in the following states ($\neg\top \doteq \bot$): pu_online_script_name_services_directory_services_intro_01: $\neg$ {DeviceOnline, Weiterleitung von Anfragen, ...} $\doteq \emptyset$ pu_online_script_name_services_directory_services_dns_01: $\neg$ {DeviceOnline, Weiterleitung von Anfragen, ...} $\doteq \emptyset$ ...

Table 6.11: $\mathcal{ALC}$CTL Model Checking Counter Examples: BIS.

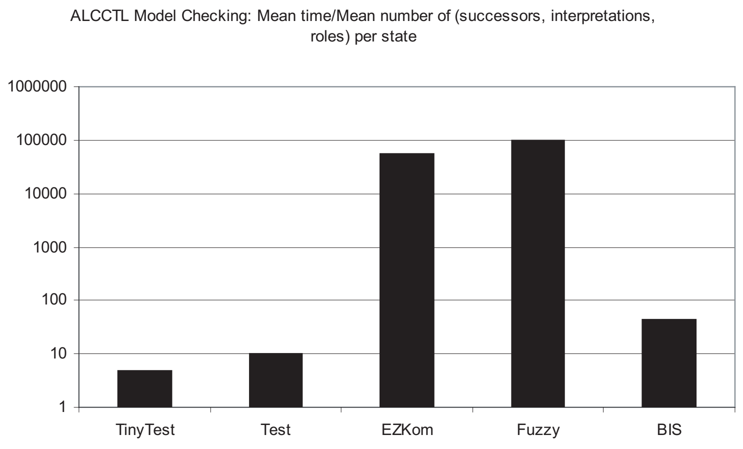


Figure 6.4: $\mathcal{ALC}$CTL Model Checking: Relation of *avg. time* to $\sum_{\{succ,\, interp,\, roles\}} \varnothing$ (log. scale).

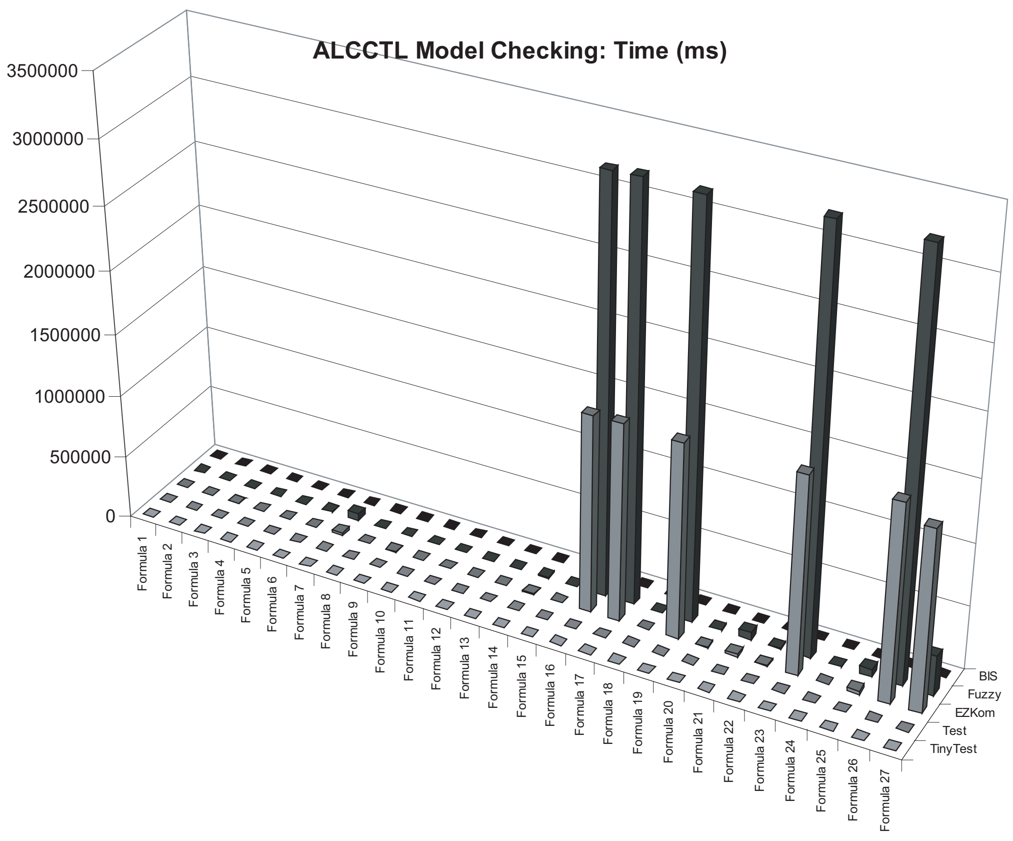


Figure 6.5: $\mathcal{ALC}$CTL Model Checking: *time*.

not true, the $\top\, U\, \phi$ check terminates quickly. In the four formulas mentioned above, ϕ is far less likely, so the U check has to be done for quite some time – resulting in the obvious poor performance.
A simple way to avoid these $\top\, U\, \phi$ checks would be to eliminate the base reduction and check for whatever operator (EF, in this case) was reduced to $E\,[\top\, U\, \phi]$ itself.

The counter examples extracted for the failed model checks all appear to be quite helpful in tracking down the problem. This illustrates that the current approach, where the point of failure is looked for at the $\dot{\sqsubseteq}$ or $\doteq$ operator, can be rather successful, more so if it were adapted for formulas that do not contain this operator. However, there are many possible formulas where this approach would fail, for example for the negation of any of the given formulas!

Chapter 7

Comparison and Analysis

Figure 7.3 shows an effect already observed in chapter 6.7: Four formulas take above-average time with the algorithmic approach. This is due to the disadvantageous base reduction.

Figure 7.4 shows how the two approaches scale. The tests were run on nine models with different numbers of states (between 10 and 5000), and with one definition and one example (of the same concept) in each 10 states. Checked was formula 18 ($\bot \mathrel{\dot\sqsubseteq} \top$). Since performance in relation to the length of the formula is completely dependant on the operators used, and the performance of different operators has already been tested, the scaling was not done along the size of the formula.
Note that the y-axis scale is logarithmic, showing an exponential performance decrease for the CTL model checker, and that the CTL approach did not produce results for the last two models. Instead, it produced the following error:

```
Assertion failed: h->array, file heap.c, line 122
```

The total time values for CTL range from 250ms to an impressive 5 hours (or about 8 hours before announcing the error for the model with 2000 states), while those for $\mathcal{ALC}$CTL range from 31ms to a little less than 5 seconds. Model extraction, export and parsing are not included in those numbers.

TinyTest	Test	EZKom	Fuzzy	BIS
1.2mb	6.7mb	–	–	0.6mb

Table 7.1: Model Checking Memory Usage Comparison (CTL usage − $\mathcal{ALC}$CTL usage: Memory usage advantage of the algorithmic approach).

	TinyTest	Test	EZKom	Fuzzy	BIS
Formula 1	344ms	3s 203ms	–	–	2s 250ms
Formula 2	328ms	3s 250ms	–	–	641ms
Formula 3	359ms	3s 180ms	–	–	656ms
Formula 4	359ms	3s 297ms	–	–	640ms
Formula 5	266ms	3s 276ms	–	–	672ms
Formula 6	376ms	3s 313ms	–	–	641ms
Formula 7	266ms	3s 328ms	–	–	625ms
Formula 8	376ms	3s 205ms	–	–	656ms
Formula 9	313ms	3s 423ms	–	–	689ms
Formula 10	376ms	3s 328ms	–	–	673ms
Formula 11	266ms	3s 438ms	–	–	688ms
Formula 12	360ms	3s 345ms	–	–	657ms
Formula 13	375ms	3s 439ms	–	–	703ms
Formula 14	375ms	3s 328ms	–	–	657ms
Formula 15	360ms	3s 454ms	–	–	687ms
Formula 16	453ms	–	–	–	-593ms
Formula 17	453ms	–	–	–	-735ms
Formula 18	265ms	3s 438ms	–	–	172ms
Formula 19	516ms	–	–	–	-593ms
Formula 20	453ms	–	–	–	173ms
Formula 21	297ms	3s 422ms	–	–	93ms
Formula 22	344ms	3s 391ms	–	–	125ms
Formula 23	531ms	–	–	–	-703ms
Formula 24	563ms	–	–	–	109ms
Formula 25	344ms	3s 968ms	–	–	78ms
Formula 26	3m 56s 156ms	–	–	–	–
Formula 27	3m 50s 405ms	–	–	–	–

Table 7.2: Model Checking Test Comparison (CTL time − $\mathcal{ALC}$CTL time: Speed advantage of the algorithmic approach).

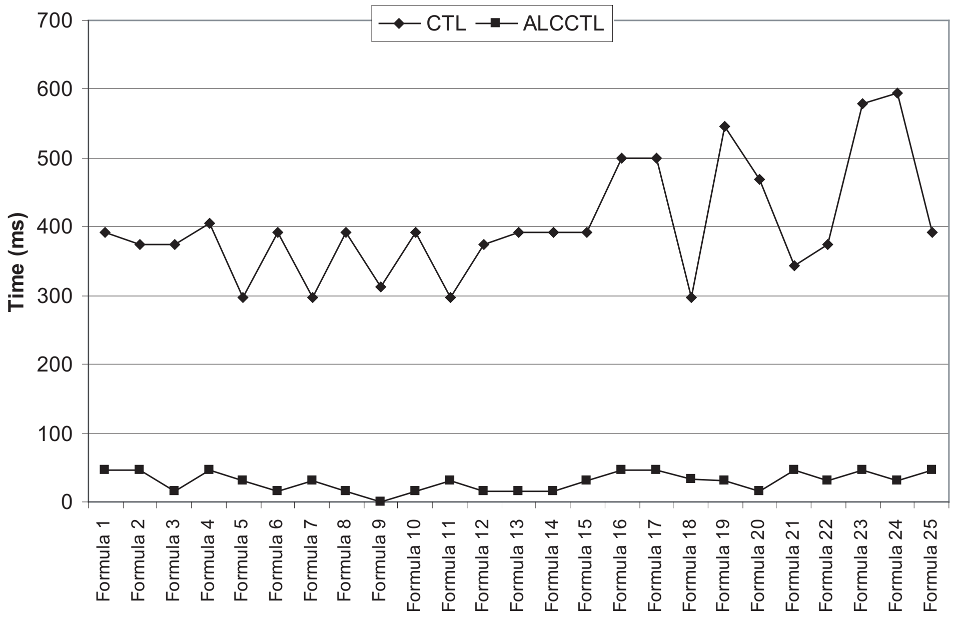


Figure 7.1: Model Checking Comparison: TinyTest.

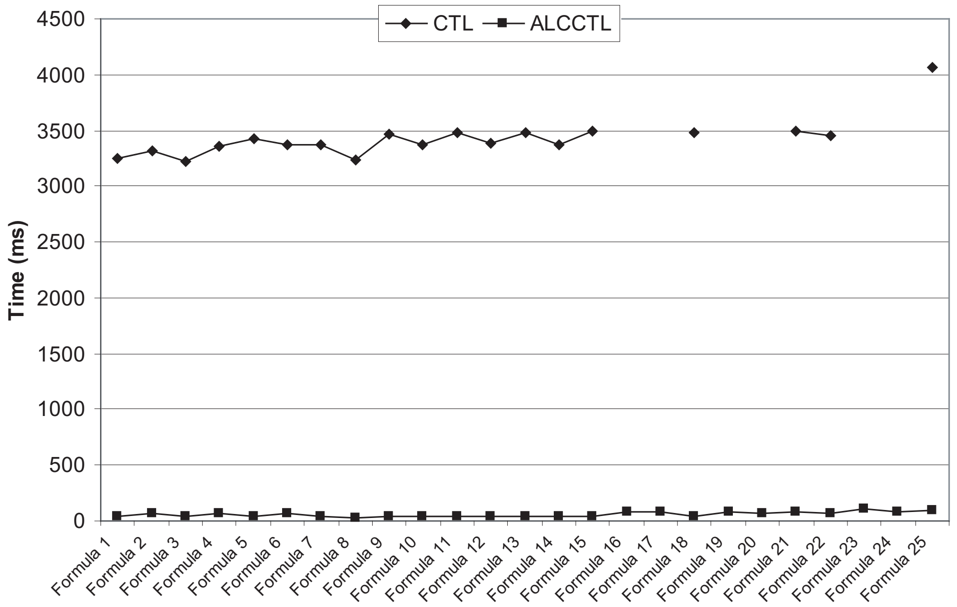


Figure 7.2: Model Checking Comparison: Test.

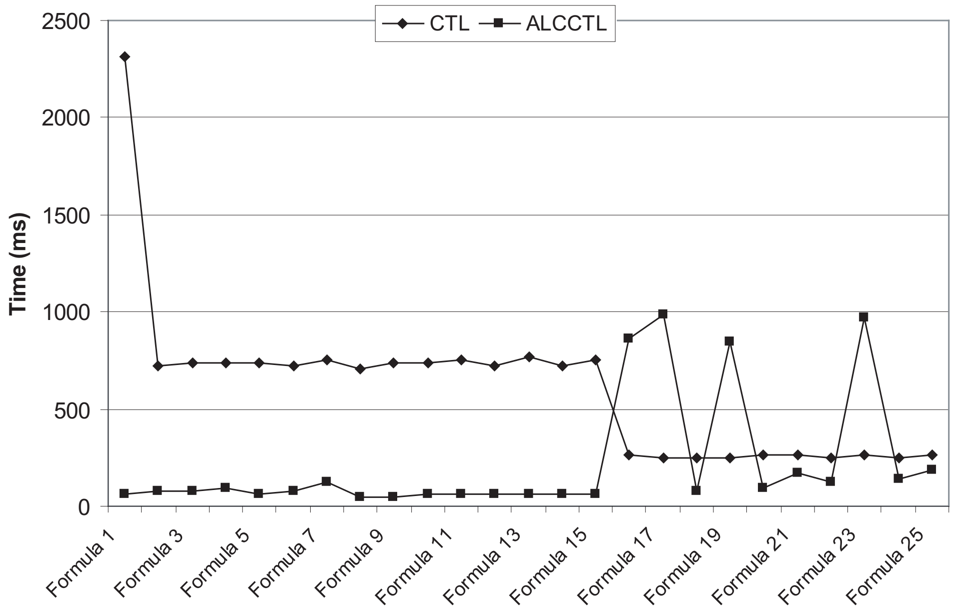


Figure 7.3: Model Checking Comparison: BIS.

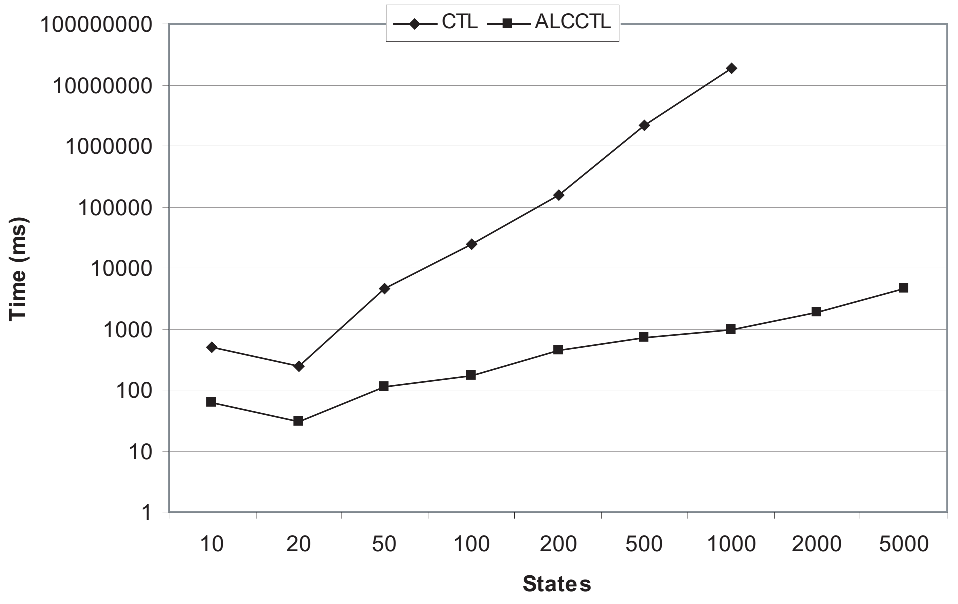


Figure 7.4: Model Checking Comparison: Scaling States.

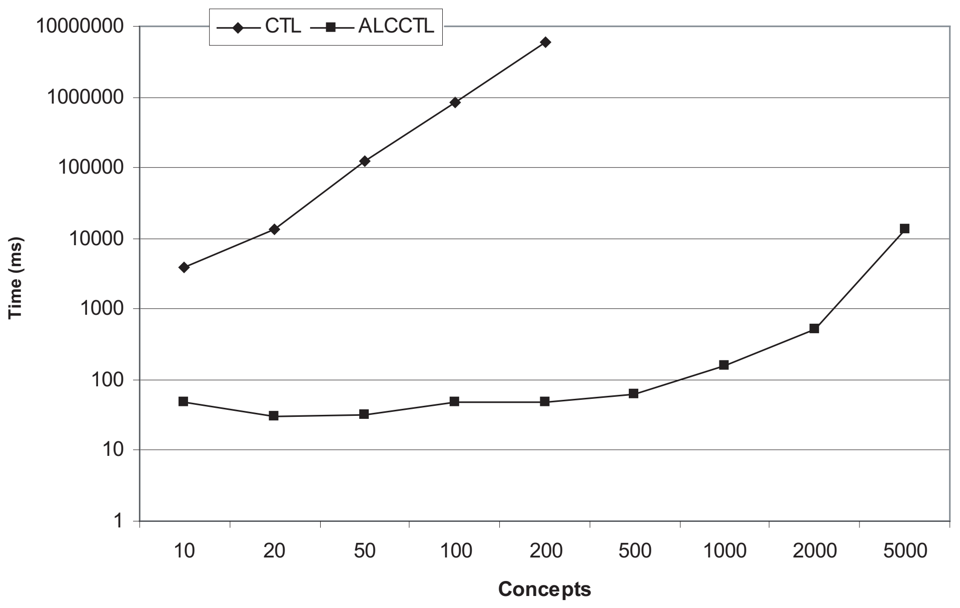


Figure 7.5: Model Checking Comparison: Scaling Concepts.

The figure shows that the CTL reduction approach does not scale well with an increasing number of states. Adding more concepts to a model makes this even worse, since it increases the number of variables. Figure 7.5 shows the same comparison as figure 7.4, but for a number of 10 to 5000 different concepts, with 10 states. This time, the CTL model checker refused to cooperate from 500 concepts upwards (with the same error as above after about 8 hours).

When asked to summarise the test results of the two $\mathcal{ALC}$CTL model checking approaches in a few words, I would be forced to say about the reduction approach: "unsuitable", and about the algorithmic approach: "usable, but in need of enhancement".

The fact alone that the reduction approach did not work in all test cases puts it "out of the game", so to speak. But the algorithmic approach, too, is far from being perfect. Several test cases took a horrendous amount of time, and two additional tests with AF failed completely. And then there is the whole counter example issue: There are still many cases where a useful counter example is not available.

Most of the performance issues could be corrected by abandoning the base reduction and implementing all temporal operators (or at least most of them). But even then for large models there will still be formulas that require several seconds up to a few minutes to check – too long for a truly interactive environment. Some of that time might be eaten up by the slow Java string processing, which is another avenue for optimisation. But nonetheless, for an application that provides direct user interaction, a few seconds are the utmost possible delay – the average should be measured in milliseconds. Such an application might graphically show the user in which states of a model a formula holds, and if the user makes some changes to the formula, the resulting validity changes are highlighted. An application like this might in fact even be feasible without too many optimisations: Small changes to a formula imply that after each change not the entire formula has to be re-checked, but only a partial subformula. In the meantime, for more demanding uses, at the very least some progress information should be provided for lengthy operations.

Performance optimisations aside, I believe that the true usability problem is the counter examples: If a user has no convenient way of finding the faults in a formula, he is not going to use the system. Error recovery is one of the most important features in modern computer programs – doubly so if the error is the user's. But even if failing the model checking is the desired effect of a formula, the user will probably still want to check if the point of failure is the correct one. All that hinges on the quality of the counter example.

There is one more small issue that slightly impedes usability: The formula syntax is somewhat cumbersome. It would be far more convenient for a user if he could (at least as an alternate form of input) arrange his formula by dragging components or even entire subformulas from a component palette into place. That is, of course, a prerogative of a graphical user interface.

A small feature that might be nice to have is some kind of "mostly" operator. Especially with large models, complicated formulas tend to be true most of the time (that is, if they are true at all), but fail in very few special cases. The model checking algorithm correctly marks such formulas as false – however, it is quite possible that one or two exceptions to the formula would be desired, e.g. for special cases in an introduction or something like that. Of course, these cases could be coded explicitly into the formula, but that is both tedious and complex, greatly increasing time and effort requirements for the user *and* the model checker. An operator that allows for some deliberate error factor, maybe for one or two percent of the states, would be of use here. To illustrate that behaviour, imagine a model where for every definition follows an example directly afterwards, *except* in the introduction: Here, there are a small definition and its example both in the same state. A simple formula to check for the example-after-definition constraint is $AG\left(definedTopic \mathrel{\dot{\sqsubseteq}} EX\, exemplifiedTopic\right)$. Allowing the exception on a general level would be

$AG\left(definedTopic \mathrel{\dot{\sqsubseteq}} (exemplifiedTopic \sqcup EX\, exemplifiedTopic)\right)$, but allowing it specifically for the introduction would require something along the lines of $AG\left(\left(definedTopic \mathrel{\dot{\sqsubseteq}} EX\, exemplifiedTopic\right) \vee \left((definedTopic \wedge \exists hasTopic.Introduction) \mathrel{\dot{\sqsubseteq}} exemplifiedTopic\right)\right)$. Using a "mostly" operator $\odot$, it could be written (even though not with the same precision) as $\odot\left(definedTopic \mathrel{\dot{\sqsubseteq}} EX\, exemplifiedTopic\right)$.

Chapter 8

Conclusion

In the course of this Diploma Thesis I have described, implemented, tested and analysed two different approaches to the model checking problem for $\mathcal{ALC}$CTL. The analysis has shown the first approach, reduction to CTL, to be of only limited use. The second approach, reimplementation of the model checking algorithm for $\mathcal{ALC}$CTL, has proven to be more promising. Both methods allow for several avenues of optimisation, however, so a final verdict cannot be reached at this time. Model checking $\mathcal{ALC}$CTL directly has the advantage of working on the "natural structure" of $\mathcal{ALC}$CTL models, while reducing such a model to CTL enlarges the structure exponentially. On the other hand, logical optimisations in CTL might upset this efficiency edge. Nonetheless, the current implementation clearly favours the algorithmic approach.

8.1 Possible Extensions and Future Work

Several issues remain open for the time being. An optimised version of the CTL reduction is the primary concern for that approach: If it were possible to reduce the number of variables significantly, or to represent them more efficiently, the performance gain would probably be considerable. It is as of yet unknown whether that possibility exists or not.

The algorithmic approach has no such central issue concerning performance, yet one concerning usability: The extraction of a useful counter example is the main problem here. Aspects of this problem are

- Finding the relevant point of failure
- Base reduction can change a formula considerably
- Selecting a suitable error representation (single example vs. all errors)

- Java String related performance loss

(see chapter 6.4). Without a working counter example, the usability is severely limited.

It can be seen that abandoning or at least restricting the base reduction would at the same time solve two different problems. It would increase performance (especially for the AF case described in chapter 6.7), and it would improve recognisability of the counter example formula. It would, unfortunately, require the implementation of no less than 10, and up to 14 operators (depending on whether the B operator would still be represented by other operators).

Among other options for enhancement is memory management: A manual release of obsolete temporary objects would decrease the total memory usage. Since Δ^I, concepts, role and state names are all string based, avoiding the slow Java String implementation and using a more efficient representation, e.g. a numbered index-based one, would certainly prove beneficial as well.

In addition to the optimisations, there are some possible extensions to the model checking process. One possibility is to combine the two model checking approaches into a hybrid algorithm: The part that is expensive to reduce to CTL is checked directly in $\mathcal{ALC}$CTL, and the result along with the rest of the formula is then reduced and checked in CTL. Two points of separation appear feasible: Either just after the $\dot{\sqsubseteq}$ or $\doteq$ operator, thus putting the concept part on $\mathcal{ALC}$CTL turf and the formula part in CTL territory. Or after the last role operator, which is usually the main reason for the exponential growth of the CTL formula. The first options would be easier to implement because on formula level there would have to be no translating of sets to Boolean variables.

As mentioned in 7, it is sometimes hard to make complicated formulas work for large models: There simply is no room for exceptions. One possible cure is the introduction of a “mostly” operator that allows for a certain number of exceptions to the formula, e.g. 1%. Another possibility is to annotate the result with a statistical probability, such as “the formula is valid in 97% of all cases”. Theoretically probability information could be annotated to any initial concept (coming from the ABox assertions of the semantic model), and thus an overall probability for the entire formula could be derived – maybe in combination with the aforementioned statistical probability. For example, if an EF operation is valid for one out of three successors at a given state, the probability that the user will actually select this path is $\frac{1}{3}$. If one of these paths is e.g. an excursion, it could be initialised with a lower base probability like $\frac{1}{|successors| \cdot 2}$, giving the EF operation an overall probability of $\frac{5}{12}$.

There are still many other possible extensions, optimisations and options to boost performance and usability of $\mathcal{ALC}\mathsf{CTL}$ model checking.

Bibliography

[BCCZ99] A. Biere, A. Cimatti, E. Clarke, Y. Zhu: *Symbolic model checking without BDDs*, In TACAS'99, March 1999

[BCM+92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, L.J. Hwang: *Symbolic model checking* 10^20 *states and beyond*, Information and Computation, 98(2):142–170, June 1992

[BHWZ04] S. Bauer, I. Hodkinson, F. Wolter, M. Zakharyaschev: *On non-local propositional and weak monodic quantified CTL**, Journal of Logic and Computation, 14:3-22, 2004

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri: *NuSMV: a new symbolic model checker*, International Journal on Software Tools for Technology Transfer, 2000

[CCGR+02] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella: *NuSMV 2: An open-source tool for symbolic model checking*, in Proceeding of the 14th International Conference on Computer-Aided Verification (CAV'2002)

[CGMZ94] E. Clarke, O. Grumberg, K. McMillan, X. Zhao: *Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking*, Technical Report CMU-CS-94-204, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, October 1994

[Cla03] E. Clarke: *Theorem Proving and Model Checking in PVS, Lecture 1: Symbolic Model Checking with BDDs*, Lecture script, Carnegie Mellon University, Pittsburgh, 2003

[DM00] F. Donini, F. Massacci: *EXPTIME tableaux for ALC*, Artificial Intelligence, 124(1):87-138, 2000

[Don02] F. Donini: *Complexity of Reasoning*, in the Description Logic Handbook, edited by F. Baader, D. Calvanese, D.L. McGuinness, D.

Nardi, P.F. Patel-Schneider, Cambridge University Press, 2002, pages 101-141

[Fre03] B. Freitag: *Description Logics und Onthologien*, Lecture script, Universität Passau, 2003/2004

[Hol91] G. Holzmann: *Design and Validation of Computer Protocols*, Prentice Hall, New Jersey, 1991

[HR00] M. Huth, M. Ryan: *Logic in Computer Science - Modeling and Reasoning about Systems*, Cambridge University Press, 2000

[Ki96] A. Kick: *Generation of Counterexamples and Witnesses for Model Checking*, PhD thesis, Fakultät für Informatik, Universität Karlsruhe, Germany, July 1996

[KLT+04] L. Kornelsen, U. Lucke, D. Tavangarian, D. Voigt, M. Waldhauer: *Mehrdimensionale Lehr- und Lerninhalte und ihre Beschreibung in der Sprache* $<ML^3>$, Workshop Structured eLearning – Wissenswerkstatt Rechensysteme, Rostock, 30. März 2004

[McM93] K. L. McMillan: *Symbolic Model Checking*, Kluwer Academic Publ., 1993

[Neu04] L. Neumann: *Temporal Logic*, in Seminar Wissensrepräsentation, Universität Passau, 2004/2005

[Rad04] S. Radde: *Ein System zur Verifikation von Webdokumenten mit temporaler Beschreibungslogik*, Diploma Thesis, Universität Passau, 2005

[Sch91] K. Schild: *A correspondence theory for terminological logics: Preliminary report*, in Proc. of the 12th Int. Joint Conf. on Artificial Intelligence (IJCAI'91), pp.466-471, 1991

[Sch02] Ph. Schnoebelen: *The complexity of temporal logic model checking*, in Advances in Modal Logic, papers from 4th Int. Workshop on Advances in Modal Logic (AiML'2002), Sep.-Oct. 2002, Toulouse, France. World Scientific, 2003

[SFC98] P. D. Stotts, R. Furuta, C. R. Cabarrus: *Hyperdocuments as automata: Verification of trace-based browsing properties by model checking*, Information Systems, 16(1):1-30, 1998

[Sue05] C. Süß: *Eine Architektur für die Wiederverwendung und Adaptation von eLearning-Inhalten*, Dissertation zur Erreichung des Doktorgrades, Universität Passau, 2005

[WF04a] F. Weitl, B. Freitag: *Decidability of the Satisfiability and Model Checking Problem of the Temporal Description Logic ALCCTL*, IFIS-Report 2004/01

[WF04b] F. Weitl, B. Freitag: *Checking Semantic Integrity Constraints on Integrated Web Documents*, in Proceedings CoMWIM International Workshop on Conceptual Model-directed Web Information Integration and Mining, Shanghai, 2004, Springer Verlag

[WF04c] F. Weitl, B. Freitag: *Checking Content Consistency of Integrated Web Documents* ER 2004 Conference workshop

[WF05] F. Weitl, B. Freitag: *Towards Verifying the Semantic Consistency of Integrated Web Documents*, in Proceedings WWW2005, Chiba, Japan, 2005

[WFG+04] F. Weitl, B. Freitag, W. Grass, B. Sick, R. Kammerl, M. Göstl, A. Wiesner: *Mediendidaktische Aufbereitung von Vorlesungsinhalten für das Online-Lernen*, Workshop Structured eLearning – Wissenswerkstatt Rechensysteme, Rostock, 30. März 2004

List of Figures

List of Tables

Appendix A

Manual for Model Checking $\mathcal{ALC}$CTL

A.1 Installation requirements

To be able to use the model checking tools (namely the command-line tool ALCCTLModelChecking and the graphical user interfaces MCGUI_CTL and MCGUI_ALCCTL), an installed version of Java 1.5 is required, including a correctly set `CLASSPATH`. To be able to use the included batch-files (Windows®), the `PATH` needs to include the directory where java.exe and javac.exe are located. If the batch-files are not used, the `CLASSPATH` needs to include a reference to the CUP v.11 runtime jar file (located in /Application/Program/ALCCTL/Parser). Model extraction of *Lmml* models also requires a copy of the *Lmml* DTD files in the directory /wwrpub/schema. No other tools or resources are needed, and apart from the Java Runtime Environment (or Java Software Development Kit) no software needs to be installed.

A.2 Command-line Tool ALCCTLModelChecking

The tool ALCCTLModelChecking has the following syntax:

```
 ALCCTLModelChecking  [formulafile] [-xml|-lmml|-ml3 file]
                      [-output file] [-savemodel file] [-savesvg file]
                      [-noce] [-allstates] [-debug] [-notimer]
                      [-savectl file] [-noroles]
                      [-ctl toolpath ctlmodel ctlfile]
```

`[formulafile]`	A file containing one or more ALCCTL formulas.
`-xml [modelfile]`	An XML file containing the model.
`-lmml [module.xml]`	A main *Lmml* module file. Alternative to -xml.
`-ml3 [cmain.xml]`	An $<ML^3>$ cmain file. Alternative to -lmml.
`-output [filename]` (opt.)	Write the output to a file. Disables time measurement.
`-savemodel [xmlfile]` (opt.)	Save the model to an XML file.
`-savesvg [svgfile]` (opt.)	Save an SVG view of the model.
`-noce` (opt.)	Do not collect counter example information.
`-allstates` (opt.)	Check all states, not just the starting states.
`-debug` (opt.)	Display more detailed status information.
`-notimer` (opt.)	Do not measure the time requirements.
`-savectl [ctlfile]` (opt.)	Save a CTL reduction of the model.
`-noroles` (opt.)	Do not include roles in a CTL model. Used in combination with -savectl.
`-ctl [toolpath] [ctlmodel] [ctlfile]` (opt.)	Use CTL reduction and a CTL tool instead of the $\mathcal{ALC}$CTL model checking algorithm. [ctlfile] is generated automatically from the base model [ctlmodel].

A.3 Graphical User Interfaces MCGUI_CTL and MCGUI_ALCCTL

Figure A.1 shows the model checking graphical user interface for the CTL reduction. In the Formula box, the user can either enter the formula directly, or load a text file with one or more formulas. In the Model box, the user can load a model from a previously saved XML file. In theory, he could also extract a model from an *Lmml* or $<ML^3>$ source. However, there is an unfixed Java bug (as of Java 1.5.0_06) that produces an error during the XSL transformation when a Java Swing window is open. Until that bug is fixed (or a workaround is found), the model extraction is only available via the command-line interface ALCCTLModelChecking (see above). In the Model Checking box, the user can select a filename for the CTL model, and for the CTL model that includes the formula. He also has to select an external CTL model checking tool such as NuSMV. If the formulas he wants to check do not contain roles, the user can also check the "No Roles in Formula(s)" option, which will exclude any roles from the CTL model export.
The button "Run Model Checking" does just what it says – and displays the results in the Result box.

Figure A.2 shows the model checking graphical user interface for the $\mathcal{ALC}$CTL algorithmic model checking. The Formula and Model boxes are

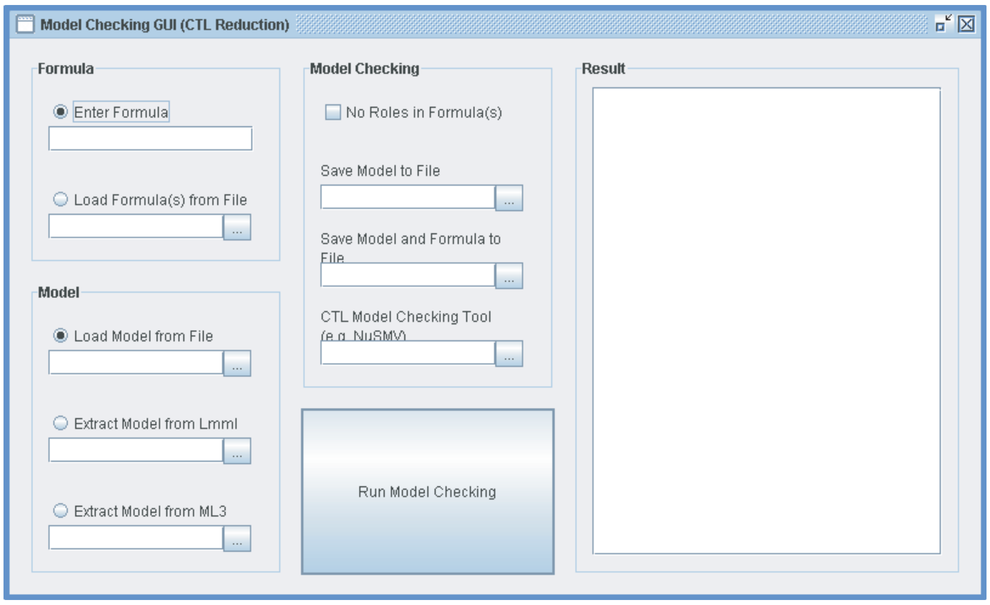


Figure A.1: Screenshot of the CTL reduction GUI.

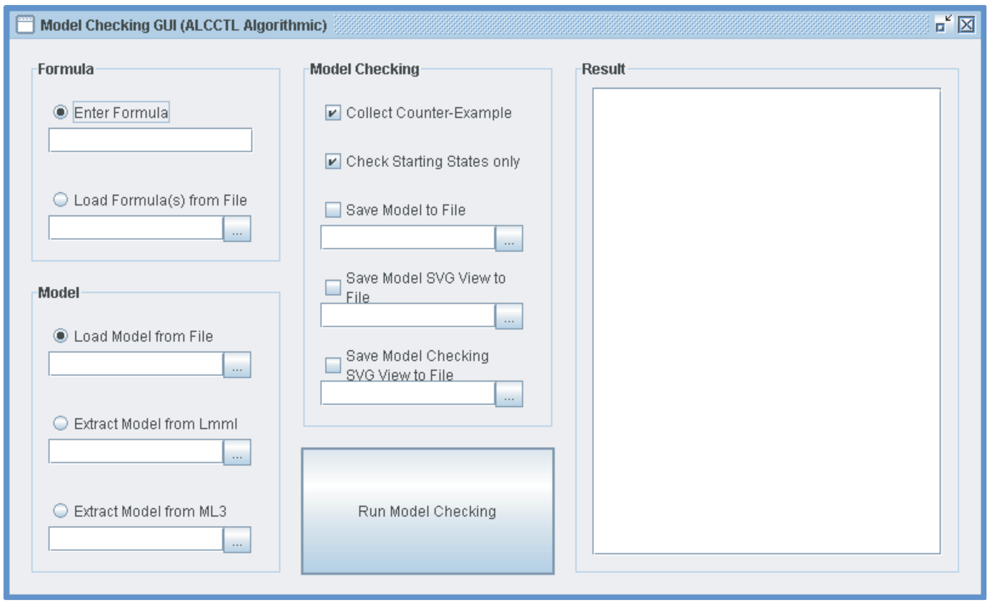


Figure A.2: Screenshot of the ALCCTL algorithmic GUI.

the same as above (including the Java bug!). The Model Checking box offers different options, however. The user can select whether or not he would like to have a counter example available, and whether or not he wants only the initial states (starting states) to be checked. He can choose the save the model to an XML file (this options is of little use as long as the model extraction does not work with the GUI). He can also choose to save an SVG view of the model, and an SVG view of the model checking process itself. Again, the appropriate button starts the model checking process, and the result is displayed in the Result box.

A.4 Formula Syntax

An $\mathcal{ALC}$CTL formula follows the syntax rules specified in chapter 2.4. However, a few things should be noted:

- Constants are denoted in capital letters: TRUE, FALSE, TOP, BOTTOM.
- The same is true for operators: SUBSET, EQUALS, AND, OR, NOT, IMPLIES.
- Temporal operators are denoted as usual: AF, EX, E[U], ...
- Role quantors are also capitalised: FORALL, EXISTS.
- AND, OR and NOT can be used both for formulas, where they are interpreted as Boolean operators, and for concepts, where they are interpreted as the set intersect, union and complement.
- The names of atomic concepts and roles must start with either an underscore or a letter, followed by underscores, dashes, letters or numbers.
- White space in a formula is generally ignored, except when reading multiple formulas from a file: Then a linebreak separates two formulas.
- The operator NOT has the strongest binding, followed by the role quantors. AND and OR are weaker, followed by the temporal operators. IMPLIES is still weaker, while SUBSET and EQUALS have the weakest binding priority.
- Parenthesis () can be used to circumvent default binding priorities.

A.5 Predefined Concepts and Roles

Concepts		Roles		
Name	Value	Name	Concept 1	Concept 2
definedTopic	title	scaleTo	scaling	id
exemplifiedTopic	title	hasScaling	id	scaling
Fragment	id	topicOf	title	id
Hint	id	hasTopic	id	title
Algorithm	id	definedAt	id	title
Analogy	id	exemplifiedAt	id	title
Declaration	id			
Definition	id			
Demonstration	id			
Question	id			
Answer	id			
Description	id			
Example	id			
Explanation	id			
Illustration	id			
Proof	id			
Proposition	id			
Reflection	id			
Remark	id			
Task	id			
Simulation	id			
Theorem	id			

A.6 *Lmml* and <ML^3> Modules

Lmml modules consist of one or more XML source files, formula graphics and other resources. For the purposes of model checking, only the XML sources are relevant: Namely the main module file called module.xml. It is this file that must be referred to when extracting model information from an *Lmml* module.

Similar to *Lmml* modules, <ML^3> modules include a collection of resources that can be discarded for model checking. There are, however, two main xml files, one for the content information (called cmain.xml), and one for the didactic information (called dmain.xml). Model extraction makes use of the first one, cmain.xml.

```
<!DOCTYPE model [
  <!ELEMENT model (states, deltaI)>
  <!ATTLIST model source CDATA #IMPLIED>
  <!ELEMENT states (state*)>
  <!ELEMENT deltaI (d_item*)>
  <!ELEMENT state (successor*, interpretation*, predicate*, role*)>
  <!ATTLIST state name CDATA #REQUIRED
                  startingState (yes|no) #IMPLIED>
  <!ELEMENT d_item EMPTY>
  <!ATTLIST d_item value CDATA #REQUIRED>
  <!ELEMENT role (r_item*)>
  <!ATTLIST role name CDATA #REQUIRED>
  <!ELEMENT successor EMPTY>
  <!ATTLIST successor name CDATA #REQUIRED>
  <!ELEMENT interpretation (i_item*)>
  <!ATTLIST interpretation name CDATA #REQUIRED>
  <!ELEMENT predicate EMPTY>
  <!ATTLIST predicate name CDATA #REQUIRED>
  <!ELEMENT r_item EMPTY>
  <!ATTLIST r_item concept1 CDATA #REQUIRED
                   concept2 CDATA #REQUIRED>
  <!ELEMENT i_item EMPTY>
  <!ATTLIST i_item value CDATA #REQUIRED>
]>
```

Table A.1: DTD of the Model XML Structure.

A.7 Module XML Format

Table A.1 lists the DTD of the module XML format. Both `r_items` and `i_items` refer to the elements of Δ^I: The `d_items`.

A.8 Interfaces, Extensions and Options

There are several points, where the current implementation can be extended by additional features or alternate functionality.

Alternate model implementations can be created by implementing the `GeneralModel` and `GeneralModelState` interfaces; alternate formula implementations can be created by implementing the `GeneralFormula` interface or subclassing `Formula` or `Concept`. Since all inter-class and inter-package exchange uses these interfaces and abstract classes, there will be no compatibility problems.

To add new operators, new descendants of `Formula` or `Concept` have to be created. However, to be able to use them, the parser needs to be updated, too. This can be done by updating the `.lex` and `.cup` files in the "Parser Specification" directory, and generating new Java files using JLex and CUP (included on the CD). The generated Java files have to be copied to "ALCCTL/Parser".

To change the behaviour of existing operators, their `annotate` methods need to be changed. To implement more operators that where – so far – reduced to their base, their `annotate` method needs to be implemented. To ensure that they are actually used, their `translate` method needs to be altered as well, to disallow base reduction for this operator (see any of the base operators for an example).

To be able to extract models from other sources than *Lmml* and $<ML^3>$, or to do the extraction differently (e.g. declaring other roles and concepts), the class `DocumentAdapter` needs to be extended, and its `extract` method implemented. For examples on how to proceed, see `LmmlDocumentAdapter` and `Ml3DocumentAdapter`.

Similarly, to extract other models than $\mathcal{ALC}$CTL models, the class `ContentAdapter` needs to be subclassed and its `getModel` method implemented. The class `ALCCTLContentAdapter` provides an example for that. This method also needs to be changed if an external reasoner (like Racer) should be called. The `getModel` method is the best place to call it. To create a truly different model, the interface `GeneralModel` probably needs to be implemented as well (see above, "alternate model implementations").

To extend the generic semantic model that lies between the `DocumentAdapter` and the `ContentAdapter`, the class `Content` needs to be edited or extended. It provides lists of ABox assertions, TBox terminology, and model states. The Javadoc documentation provides an overview of the available fields and methods.

To be able to use another external CTL model checking tool with the CTL reduction approach, the CTL model format needs to be updated. This can be done by altering (or overloading) the method `InputOutput.saveToCTLFile`. However, the current file format (NuSMV) supposedly is backwards compatible at least with SMV, so a change may not be necessary in that case. The alternate model checker can simply be used by specifying an alternate path at the command-line tool or in the GUI.

Appendix B

Contents of the CD

B.1 Directory Listing

```
- Application
    - Modules
        - Lmml
        - ML3
    - Program
        - ALCCTL
            - Parser
        - ELearning
        - Utils
        - xsl
    - Saves
        - models
        - svg
    - Tests
    - Tools
        - CUP
        - JLex
        - MSXSL
        - NuSMV
- Implementation
    - Javadoc
    - Prototype
        - PrePrototype
            - ALCCTL
            - Javadoc
            - Utils
        - Prototype
            - ALCCTL
            - ELearning
            - Scanner & Parser
            - Utils
            - xsl
    - Sequence Diagrams
    - Source Code
        - ALCCTL
            - Parser
        - ELearning
        - GUI
        - Parser Specification
        - Utils
        - xsl
- Presentations
    - 1 Initial Presentation
    - 2 Intermediary Presentation
    - 3 Final Presentation
- Thesis
    - LaTeX
        - resources
        - sources
- wwrpub
    - schema
```

B.2 Where to Find What?

- The tools ALCCTLModelChecking, ALCCTLStatistics, MCGUI_ALCCTL and MCGUI_CTL – These tools are all located in /Application/Program.

- *Lmml* modules – Four sample *Lmml* modules (two test modules, two real ones) are in /Application/Modules/Lmml.

- The WWR *Lmml* DTDs required for *Lmml* model extraction – They are located in /wwrpub/schema.

- <ML^3> modules – A sample <ML^3> module is located in /Application/Modules/ML3.

- XML model files – Extracted XML model files of those modules are in /Application/Saves/models.
- CTL model files – Exported CTL model files of those modules are in /Application/Saves/models.
- SVG views – SVG views of those models can be found in /Application/Saves/svg.
- Example formulas – Some example formulas are in /Application/Tests.
- The Javadoc documentation – The documentation is in /Implementation/Javadoc.
- The source code – The entire source code is in /Implementation/Source Code.
- NuSMV – The tool NuSMV is in /Application/Tools/NuSMV.
- The parser specification – The parser specification is in /Implementation/Source Code/Parser Specification.
- JLex and CUP – Both are in /Application/Tools in their respective subdirectories.
- Powerpoint^{TM} presentations about this Diploma Thesis – All presentations are in /Presentations in their appropriate subdirectories.
- This thesis document – A PDF version of this document resides in /Thesis.
- The LaTeX source files for this document – The LaTeX source files are in /Thesis/LaTeX.
- The figures for this document – All figures are in /Thesis/LaTeX/resources.
- The sources/papers referenced in this document – All sourced that are available for download are in /Thesis/LaTeX/sources.
- The first two versions of the implementation – They are located in /Implementation/Prototype.

Appendix C

Acknowledgements

This concludes the Diploma Thesis “Model Checking Temporal Description Logics” – thank you for reading. At this point I would like to thank everyone who made this project possible. First, there is my tutor, Franz Weitl, without whose advice and insights I would probably have spent some time banging my head against various walls – real and imaginate. I would also like to thank Professor Freitag, for his observations and helpful criticisms. I owe gratitude to my parents, Rita and Dr. Volker Schönberg, too, who helped creating this work by creating me in the first place, and then by supporting their creation in his. Additionally, I would like to thank all my friends, for their encouragement and because they reminded me that a Diploma Thesis is not a 24/7 job. Finally, my thanks belong to the Federal Republic of Germany, which provided the free, peaceful and prosperous environment that allowed for the writing of this thesis in the first place.

Passau, June 2006
Christian Schönberg

Statutory Declaration

I hereby declare that I have created this Diploma Thesis on my own and without using any resources other than the ones mentioned appropriately. All text that has been quoted either literally or analogously from other sources is marked accordingly. This Diploma Thesis has not been presented in this or in a similar form to any examination board before.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Diplomarbeit selbständig angefertigt habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind entsprechend gekennzeichnet. Die Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

....

Passau, 2006-06-19 (Christian Schönberg)