FernUniversität in Hagen
Faculty of Mathematics and Computer Science

Artificial Intelligence Group

Pinpointing Conflicts in Temporal Knowledge Bases via Model Checking

Bachelor's Thesis

in partial fulfillment of the requirements for
the degree of Bachelor of Science (B.Sc.)
in Computer Science

submitted by
Melinda Betz

First examiner:	Prof. Dr. Matthias Thimm Artificial Intelligence Group
Advisor:	Dr. Jandson Santos Ribeiro Santos Artificial Intelligence Group

Statement

Ich erkläre, dass ich die Bachelorarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Bachelorarbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Bachelorarbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

	Yes	No
I agree to have this thesis published in the library.	☑	☐
I agree to have this thesis published on the webpage of the artificial intelligence group.	☑	☐
The thesis text is available under a Creative Commons License (CC BY-SA 4.0).	☐	☑
The source code is available under a GNU General Public License (GPLv3).	☐	☑
The collected data is available under a Creative Commons License (CC BY-SA 4.0).	☐	☑

Althütte, 02.05.2023 ..... Melinda Betz

(Place, Date) (Signature)

Abstract

This bachelor's thesis proposes an algorithm which combines the 3-valued model checking of a system model with partial information, represented by a KMTS, against a CTL formula with the pinpointing of any eventually existing conflicts due to the partial information. The model checking is based on the Contraction Model Checking introduced by Ribeiro and Andrade [RA15]. The proposed procedures for determining conflicting parts by Guerra et al. [GAW13] are used for pinpointing conflicts within the model. This algorithm is also implemented as a Java program with a KMTS and a CTL formula as input. In case of a indefinite result, the program lists the responsible parts of the KMTS as Failure Witnesses. The formal definitions of CTL, Kripke structures and KMTS, plus all definitions necessary for the implementation of Contraction Model Checking and the Failure Witnesses logic are introduced. Based on these, the implementation approach for the algorithms is developed and some of the most important parts of the Java implementation are shown. The thesis is concluded with some experimental test runs of the Java program as well as an evaluation and presentation of the most important results and findings of these runs. Finally, a brief outlook is given on how future improvements, be it extensions to the definition of Failure Witnesses or even an automatic revision of the input model, could increase the use of the application.

Contents

List of Figures

List of Listings

List of Algorithms

List of Definitions

1 Introduction

Many applications of artificial intelligence (AI) have the need to make decisions about meaningful actions despite incomplete knowledge of the surroundings. In a multi-agent scenario, for example, an agent often only knows parts of his environment. Since these knowledge gaps may influence the agent's decisions, it is important to be able to explicitly show for a specific point in time which information an agent has and which not. If the agent later on receives new pieces of information, he will have to check whether these contradict with his previous knowledge. If they were contradictory, the agent should be able to identify the problematic parts of his body of knowledge and, if necessary, eliminate them. Thus, the maintenance of a body of knowledge over a certain period of time is made up of consistency checks and the identification of specific knowledge conflicts.

But not only lack of knowledge can be the reason for working with partial information system models. Another use case can be doing this entirely intentionally having one abstract system model that represents a whole product line of this system to minimise the cost of variations. Therefore, the model differentiates which features or components of a system are optional, alternative or mandatory. If you take a vending machine, for example, there are many different types of vending machines for various purposes. Vending machines for tea, coffee, cacao, soup and more in every possible combination. All these machines can be represented by a common model that satisfies all the same rules. Nevertheless, this model can have a variable part that represents those aspects that will be used to adjust the machine to the needs of different customers [FG08]. One dispenses coffee, another tea, one can be fed by credit cards or banknotes, another only by coins. The procedure is called product family engineering.

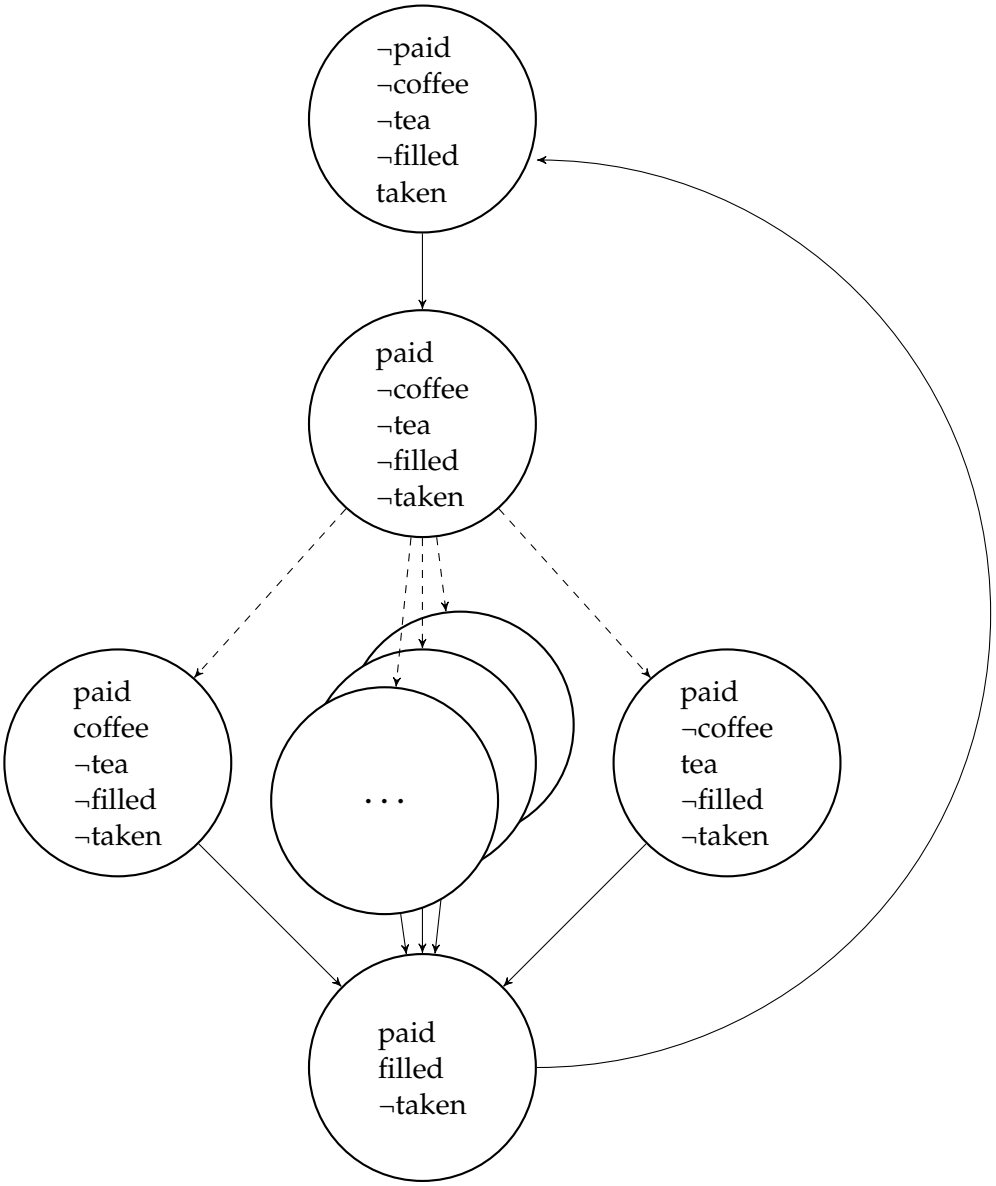

Figure 1.1: Highly simplified KMTS of a vending machine. Dashed arrows for genuine may-transitions, solid arrows for must-transitions.

A further example is the world of system development. In early phases generally not all information about the planned system is available which means there is only partial information to work with. In addition, a model often represents only a special view on a system. Sometimes it is wanted to analyse a system from different perspectives. For example, an end user of an application will have different requirements as more technically involved people. Therefore, collecting these requirements from these two groups independently will probably lead to several quite different

models but also cover a good amount of needs. Bringing together these views in one general system model will surely not come off without some incompatibilities. Not immediately trying to solve this problems, but to live some time with them to find better solutions, brings us back to the importance of representing partial information in a system model [BMPAFV04]. Even at such an early stage, the model should be able to be formally verified against the requirements of the system and, when a desired property is not satisfied, the affected part of the model should be clearly identifiable [GAW13].

1.1 Related Work

In general, a system model needs a formal structure to represent the underlying system. An example for that is the Kripke structure which we will use in this thesis. The Kripke structure is named after the American philosopher and logician Saul Kripke [Bun15] who proposed it in the early 1960s as a transition system [Kri63]. A Kripke Structure is basically made up by states, transitions and a labelling function for the states. A special form of Kripke structure, which is often used to represent indetermination within the model, is called a Kripke Modal Transition System (KMTS) first proposed by Huth et al. [HJS01]. A KMTS allows indetermination in state labels and in transformations. In contrast to a Kripke structure, it consists of two kinds of transitions, must- and may-transitions. A must-transition is a mandatory transition absolutely necessary for the structure, while may-transitions may or may not exist. Also, labels of states can be left empty in a KMTS to show the indetermination at this point. Figure 1.1 shows a simplified vending machine KMTS model that can be used as a blueprint of a coffee or a tea machine. Additional may-transitions for cacao or soup were implied and additional states and transitions for pouring sugar or milk can easily be thought of as extensions.

The logic for the formulas representing the system requirements in this thesis will be written in CTL (Computation Tree Logic). Given a Kripke structure K, an initial state s_0 and a CTL formula φ model checking solves the following problem: does K satisfy φ starting from s_0? Ribeiro and Andrade [RA15] have introduced a special variant of model checking called Contraction Model Checking to check, instead of only partial information available, whether a model represents the requirements of a system. We will use Contraction Model Checking to verify whether a given model satisfies a given CTL property. Contraction Model Checking can directly be applied to a KMTS representing partial information. Ribeiro and Andrade have used the work of Grumberg et al. [GLLS07] and Shoham and Grumberg [SG03] as a reference for their Contraction Model Checking. Shoham and Grumberg [SG03], on the other hand, exploit and extend Stirlings game-based framework [Sti01] for CTL model checking. The 3-valued semantics used are based on the work of Bruns and Godefroid [BG99]. Additionally, we use the Failure Witnesses detection as proposed by Guerra et al. [GAW13] to locate exactly the specific problems in a model.

1.2 Scope

Temporal knowledge bases as background of a system model are seldom complete and they often contain only partial information. This can cause conflicts and these conflicts should be exactly located, because only located conflicts can be solved. Whatever the reason is for the lack of definite information about a system, the parts of the model that cause the problems have to be pointed out. In this thesis, the model checking problem of such a system model will be connected with the task of exact pinpointing. The goal is to design and implement a complete algorithm for the combination of model checking with only partial information and pinpointing the conflicts which eventually occur because of the partial information. To achieve this goal, a Java program is designed and implemented. This program exactly captures the Contraction Model Checking of Ribeiro and Andrade [RA15] extended by the error analysis in Guerra et al. [GAW13]. It already contains some commonly used optimisations for model checking and will be evaluated against selected experimental constellations of models and formulas which we call use cases.

1.3 Overview

With regards to the organisation of this thesis, in Section 2 (Preliminaries) we present CTL, Kripke structures and introduce KMTS. Model checking, especially Contraction Model Checking, will be introduced in Section 3 (Contraction Model Checking). Section 4 (Pinpointing Conflicts) is about showing the conflicting areas using so-called Failure Witnesses. In Section 5 (Implementation Approach) we explain the approach to implement such algorithms including some ideas about starting points for eventually necessary performance optimisations, be it regarding time or space. Also the actual implementation will be presented there. Section 6 (Experiment and Evaluation) evaluates the implementation with the help of different use cases. Finally Section 7 (Discussion and Conclusion) completes the topic by discussing the implementation and some issues that occurred. Appendix A contains all the input files for the experimental use cases of Section 6 (Experiment and Evaluation) and some of the generated log files.

2 Preliminaries

In this section you will be guided through the basics needed for the algorithms mentioned above. It starts with some background information about CTL consistency checks and continues with model checking in general and the intended special version of Contraction Model Checking (CMC).

2.1 Computation Tree Logic

The base of our logic is temporal logic. It covers all reasoning approaches about time and temporal information as well as their formal representation within a logical framework [GR22].

"The idea of temporal logic is that a formula is not statically true or false in a model, as it is in propositional and predicate logic. Instead, the models of temporal logic contain several states and a formula can be true in some states and false in others". Since "Temporal logics have a dynamic aspect to them [...] they depend on the time point inside the model"[HR04].

Two well known temporal logics are Linear-time Temporal Logic (LTL) in which time is linear evolving along a path and Computation Tree Logic (CTL) in which the model of time is a tree structure with branching paths. In LTL the future is not determined as in CTL. However, LTL considers different possible futures, each with one possible path, any of which might be the actual future. But this is not exactly what we want for our model checking. We want one future with different paths, each of which could be the actual future path realised and not different futures. We want to be able to move in a different direction at any given moment. That is what CTL offers [HR04, Eme91].

2.1.1 CTL Syntax

To represent the requirements of a system, in this thesis, we will use temporal logic formulas written in CTL. There are different ranges of operators used for CTL formulas and there are different forms to define the CTL syntax. The following definitions specify the form and scope of how we exactly will employ CTL. The following Backus Naur form is sometimes also called the standard form.

Definition 2.1 *[HR04] A CTL formula φ in Backus Naur form, where p is an atomic proposition, is defined as follows:*

$\varphi ::= \top \mid \bot \mid p \mid (\neg p) \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid EX\varphi \mid AX\varphi \mid EF\varphi \mid AF\varphi \mid EG\varphi \mid AG\varphi \mid E(\varphi U \varphi) \mid A(\varphi U \varphi) \mid E(\varphi R \varphi) \mid A(\varphi R \varphi) \mid (\varphi \rightarrow \varphi)$

The symbols $\top$ and $\bot$ are CTL formulas always *true* respective *false*. Likewise, all atoms p and all literals l are CTL formulas too. All in the implementation of Section

5 as input allowed operators are included in this definition.

Definition 2.2 *[RA15] A CTL formula φ in negation normal form, where l is a literal, is defined as follows:*

$\varphi ::= \top \mid \bot \mid l \mid (\varphi \vee \varphi) \mid (\varphi \wedge \varphi) \mid EX\varphi \mid AX\varphi \mid E(\varphi U \varphi) \mid A(\varphi U \varphi) \mid E(\varphi R \varphi) \mid A(\varphi R \varphi)$

"Excluding the conjunction and the disjunction, all the operators must be bound by path operators" E (a path exists) respective A (all paths) [RA15]. See details e.g. at Huth and Ryan [HR04]. EX means φ holds *in some next state*. AX means φ holds *in every next state*. $E(\varphi U \varphi')$ means φ holds *in some path until* φ' holds. $A(\varphi U \varphi')$ means φ holds *in all paths until* φ' holds. $E(\varphi R \varphi')$ means φ' holds *in some path until it is released* by a holding φ. $A(\varphi R \varphi')$ means φ' holds *in all paths until it is released* by a holding φ. All operators that can be handled by the game rules of Contraction Model Checking in Section 3 are included in this definition.

A Comparison of the two CTL definitions shows: not all operators allowed for input can be dealt with in Contraction Model Checking. In other words, not for all of these operators exists a game rule in Definition 3.1. Nevertheless, we still want to allow those inputs, because some of this operators concerned are very common. Fortunately, all the operators from Definition 2.1 that are missing in Definition 2.2 can be expressed in terms of the other operators from Definition 2.2. The affected operators and their possible implementation are as follows:

F (some future state) can be implemented as [HR04]:

$AF\varphi \equiv A(\top\, U\, \varphi)$
$EF\varphi \equiv E(\top\, U\, \varphi)$

G (all future states) can be implemented as [HR04]:

$AG\varphi \equiv A(\bot\, R\, \varphi)$
$EG\varphi \equiv E(\bot\, R\, \varphi)$

$\rightarrow$ (implies) can be implemented as [Ren23]:

$\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$

2.1.2 2-valued CTL Semantics

CTL formulas are interpreted over transition systems. If we assume a model of such a system consisting of sets of states, relations and labels, the definition of a CTL semantic with only true and false will look like Definition 2.3. We will see later how a definition for a system that maps partial knowledge might look like.

Definition 2.3 *[HR04, Fin12] (CTL Semantics). Let M = (S, R, L) be a model for CTL where S is a finite set of states, $R \subseteq S \times S$ a transition relation over S, L a labelling function over states, $s \in S$, $s \to s_1 ...$ a path starting from $s \in S$ and φ a CTL formula. The relation M, s $\vDash \varphi$ is defined by structural induction on φ:*

1. $M,s \vDash \top$ *always*
2. $M,s \vDash \bot$ *never*
3. $M,s \vDash l \Leftrightarrow l \in L(s)$
4. $M,s \vDash \varphi_1 \vee \varphi_2 \Leftrightarrow M, s \vDash \varphi_1$ *or* $M,s \vDash \varphi_2$
5. $M,s \vDash \varphi_1 \wedge \varphi_2 \Leftrightarrow M, s \vDash \varphi_1$ *and* $M,s \vDash \varphi_2$
6. $M,s \vDash AX\varphi \Leftrightarrow \forall s \to s_i \in R \mid M, s_i \vDash \varphi$
7. $M,s \vDash EX\varphi \Leftrightarrow \exists s \to s_i \in R \mid M, s_i \vDash \varphi$
8. $M,s \vDash E(\varphi_1 U \varphi_2) \Leftrightarrow \exists s \to s_1 ... \mid \exists k \in \mathbb{N} \mid M, s_k \vDash \varphi_2 \wedge \forall i \in \{0, ..., k-1\} \mid M, s_i \vDash \varphi_1$
9. $M,s \vDash A(\varphi_1 U \varphi_2) \Leftrightarrow \forall s \to s_1 ... \mid \exists k \in \mathbb{N} \mid M, s_k \vDash \varphi_2 \wedge \forall i \in \{0, ..., k-1\} \mid M, s_i \vDash \varphi_1$
10. $M,s \vDash E(\varphi_1 R \varphi_2) \Leftrightarrow M,s \vDash \neg A(\neg\varphi_1 U \neg\varphi_2)$
11. $M,s \vDash A(\varphi_1 R \varphi_2) \Leftrightarrow M,s \vDash \neg E(\neg\varphi_1 U \neg\varphi_2)$

2.2 Kripke Structures

CTL assumes a representation of the future over a computation tree that can best be pictured by Kripke structures. A Kripke structure consists of a set of states connected by transitions. Each state can be labeled by a subset of a set of atomic propositions.

Definition 2.4 *[GAW13] A Kripke structure is a tuple $K = (AP, S, S_0, R, L)$ where:*

- *AP is a set of atomic propositions*
- *S is a finite set of states*
- *$S_0 \subseteq S$ is a set of initial states*
- *$R \subseteq S \times S$ is the transition relation over S*
- *$L : S \to 2^{AP}$ is a labelling function of truth assignment over states*

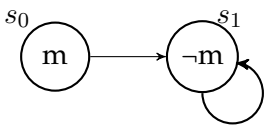

Figure 2.1: Kripke structure with two states s_0, s_1 and two transitions.

As an example, the Kripke structure shown in Figure 2.1 would, for an initial state s_0 not satisfy the CTL formula $\varphi = EXm$, because starting from s_0 there will never **exist** a path to a **next** state labelled with "m", but it would satisfy e.g. $\varphi = A(mU\neg m)$, because s_0 will satisfy with label "m" **until** on **all** possible paths s_1 is reached which satisfies with label "¬m".

2.3 Kripke Modal Transition System (KMTS)

As already revealed above, the capability of handling partial information within a system model is extremely important. To express the presence of partial information, i.e. an existing lack of information within a system, Guerra et. al. [GAW13] have proposed to use a KMTS, because a KMTS allows indetermination in state labels and in transformations.

Definition 2.5 *[GAW13] A KMTS structure is a tuple* $K = (AP, S, S_0, R^+, R^-, L)$ *where:*

- AP *is a set of atomic propositions*
- S *is a finite set of states*
- $S_0 \subseteq S$ *is a set of initial states*
- $R^- \subseteq S \times S$ *is a transition relation over* S
- $R^+ \subseteq S \times S$ *is a transition relation over* S *with* $R^+ \subseteq R^-$
- $L : S \to 2^{Lit}$ *with* $Lit = AP \cup \{\neg p \mid p \in AP\}$ *is a labelling function that associates each state in* S *with a subset of* Lit *such that for all states* $s \in S$ *and* $p \in AP$ *at most one of p and* $\neg$ *p occurs*

In the Definition 2.5 above R^+ corresponds to must-transitions and R^- corresponds to may-transitions. Contraction Model Checking requires a KMTS with exactly one initial state. So, for us, the set S_0 of initial states will always contain exactly one state. Incidentally, the same applies to our Kripke structures. However, this does not imply any limitation of the generality of our model checking and calculating regarding Kripke structures or KMTSs. Kripke structures are nothing more than the variant of a transition system with the semantics of a non-deterministic finite state machines [AY01] used for logical modelling, where the nodes are called states. State machines with multiple, let us say k, entry nodes can easily be replaced by a structure of k state machines, each with a single entry node [BGR01]. Therefore, we can express everything that a KMTS with multiple initial states could express with several of our KMTSs with only one initial state.

As a further restriction, we want to specify that a KMTS may not have multiple transitions with the same outgoing and incoming states. Thus, the transition Relations R^- and R^+ have to be sets of transitions. That is important because otherwise our algorithms would not be able to distinguish multiple transitions from one state to another. Transitions in Kripke structures do not have any labels, keys or similar features.

As already mentioned, a KMTS can explicitly represent not only information about states and transitions between these states like a Kripke structure, but it can also mark unknown information about state labels and transitions. The transitions marked this way then maybe exist or maybe not. Likewise, a KMTS itself can be represented by a set of Kripke structures, called the expansion of the KMTS. This set consists of one structure for every possible shaping the underlying model can take on. Thus,

the size of the set is 2^m with m as the number of existing indeterminations.

To be able to express the presence of partial information also in the results of our model checking, Shoham and Grumberg [SG12] proposed 3-valued-model-checking. This includes beside *true* and *false* a third truth value called *indefinite*. One method to check such a KMTS would be to check all the structures of the expansion, the sometimes called CTL models of the KMTS, individually and calculate the result for the KMTS as is presented in detail by Guerra et al. [GAW13]:

- all CTL model checks are *true* ⇒ KMTS check is *true*
- all CTL model checks are *false* ⇒ KMTS check is *false*
- otherwise ⇒ KMTS check is *indefinite*

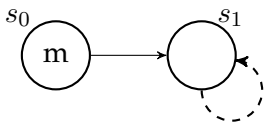

Figure 2.2: KMTS with two indeterminations. The label of s_1 is indefinite, also the transition from s_1 to s_1. The dashed arrow represents a genuine may transition, the solid one represents a must transition.

The Kripke structure from Figure 2.1 represents one of the $2^2 = 4$ Kripke structures respective CTL models of the expansion of the KMTS in Figure 2.2.

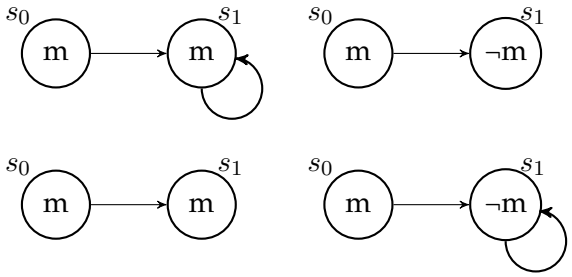

Figure 2.3: Expansion set of the KMTS in Figure 2.2.

Partial sets of this expansion are called **instances** of the KMTS.

2.3.1 Instances of a KMTS

An instance of a KMTS M represents a subset of Kripke structures from K(M). Instances are the basis for the set operations in Section 3.3.1 that will lay the foundation for the Contraction Model Checking. They can be addressed respective generated via a sequence of change operations on M.

Definition 2.6 *[RA15] A KMTS $M(X)$ denotes the instance of a KMTS which was created by applying a set X of changes on M.*

There are three primitive change operations to be used on a KMTS to generate KMTS instances [RA15].

1. $P_1(s, s')$ deleting a genuine may transition
2. $P_2(s, s')$ transforming a may transition into a must transition
3. $P_3(s, l)$ adding a label to a state, when there is not already a label of this kind assigned to it

Not all combinations of changes on a KMTS are possible. So-called complement operations exist which a KMTS cannot satisfy at the same time [RA15].

Definition 2.7 *[RA15] Let p be a primitive operation, the complement operation of p denoted by $\neg p$ is defined as follows:*

(i) $p = P_3(s, l)$ *iff* $\neg p = P_3(s, \neg l)$
(ii) $p = P_1(s, s')$ *iff* $\neg p = P_2(s, s')$

Furthermore, we note two sets of changes as not compatible if both have at least one complement operation of the other one.

Definition 2.8 *[RA15] Let p be a primitive operation. A set X of changes is not compatible with a set Y of changes, denoting by:*

$$X \not\asymp Y \Leftrightarrow \exists p \in X \mid \neg p \in Y.$$

2.3.2 3-valued CTL Semantics

In order to achieve 3-valued model checking results, we need 3-valued CTL semantics. In Ribeiro and Andrade [RA15] all definitions including derivations can be found necessary to build a complete 3-valued CTL semantic with respect to a KMTS. There is also a table that lists all these 3-valued CTL semantics applicable for Contraction Model Checking.

The representation of these semantics is norm-based as introduced by Grumberg [Gru10] and continued by Guerra et al. [GAW13]. From this CTL semantics, Guerra et al. [GAW13] as well as Ribeiro and Andrade [RA15] derived a norm for CTL formulas on states of a KMTS regarding to an expansion of this KMTS. Deviating from them, we want to use the symbols $\top$ and $\bot$, which are usual in CTL, for *true* respective *false* (see Definition 2.1 and Definition 2.2). For *indefinite* we define $\mathtt{I}$.

Proposition *Let M be a KMTS, K(M) the expansion of M and φ a CTL formula, then*

$$\|\varphi\|_M(s) = \begin{cases} \top & \textit{iff } \forall k \in K(M) \mid k, s \vDash \varphi \\ \bot & \textit{iff } \forall k \in K(M) \mid k, s \nvDash \varphi \\ \mathtt{I} & \textit{otherwise} \end{cases}$$

The proof for this proposition follows straight from the semantics of KMTS expansions.

3 Contraction Model Checking

3.1 Game Theory

The core idea of the game theory, on which contraction model checking is based, is that not loosing is better than winning (see also [GLLS07] for a more detailed explanation). For the game two players are defined. A player $\forall$ who tries to prove the formula wrong. As well as a player $\exists$ who tries to verify the formula. The game used for the model checking approach consists of a formula which is to be checked φ. And a graph of configurations which is of the type $s \vdash \psi$. s is a state of the model and ψ is a subformula of φ. The configurations are determined via the decomposition of the subformulas according to the game rules, considering the states and transitions of the KMTS model.

All players move from a configuration to another configuration according to their strategy. A so-called strategy is a function between the configurations of a player and all the configurations of a game graph. A winning strategy of a player makes this player win the game no matter what the strategy of the other player involved in the game is. When both players win, it means that they both have a non-losing strategy and the game results in $\perp$ [GAW13].

The model checking approach consists of a game $M, s \vdash \varphi$ played over a board by two players $\exists$ve and $\forall$belard. The game rules define the possible moves each player can make. The board is constructed according to these rules [RA15].

Now, that we know on what the model checking approach is based on. The next question to answer would be, what actually is model checking?

Model checking is a successful approach for the verification of hardware and software in industry. In other words for verifying whether a system model M satisfies a specification φ written as a temporal logic formula. "[...][I]t is a computer-assisted method for the analysis of dynamical systems that can be modeled by state-transition systems" [CHV18]. Thus, a system model in this context is an abstraction of a real world problem. Many model checking tools, and most of the educational approaches to model checking, use Kripke structures to represent their system models as shown above [CHV18].

An exponential growth in the number of Kripke structures to be checked is one of the problems of the above mentioned method for an increasing number of indeterminations within our model. A solution could be doing the consistency checks not for the expansion set but directly for the original KMTS. Another problem is that the common model checking techniques, for example Grumberg et al. [GLLS07], do not exactly capture the Guerra et al. interpretation of partial information within a KMTS.

Both of these problems can be tackled using the Contraction Model Checking introduced by Ribeiro and Andrade [RA15]. In that work Ribeiro and Andrade proposed the theory of the Contraction Model Checking but without presenting an algorithm strategy for implementation. Because, according to Ribeiro, the complexity of model checking with the semantics of Contraction Model Checking is NP-complete[1] , it is advisable to do some investigations on an efficient way of implementation. We will get to that later in Section 5 when we talk about implementation approaches.

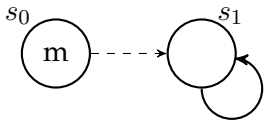

Figure 3.1: KMTS M to be model checked with CTL formula for initial state $s_0 \vdash \neg m \vee EX(m)$.

3.2 The Arena

This model checking technique is based on a game theory approach and defines a colouring function over a previously build game arena, depending on the given KMTS and a given CTL formula. In this case the game arena is a tree structure where the nodes are called configurations. Each configuration can be seen as a pair of CTL formula and state where the root configuration pictures the given formula. All the other configurations are more and more decomposed subformulas of this root formula. This decomposing follows the game rules for the model checking game presented by Ribeiro and Andrade [RA15] as shown in Definition 3.1. The links between these configurations are called edges. Depending on the transitions between the concerned states we also distinguish for the edges between may and must edges. In Figure 3.2 we see a may edge between the configurations $\boldsymbol{c_2}$ and $\boldsymbol{c_3}$ marked by the dashed arrow.

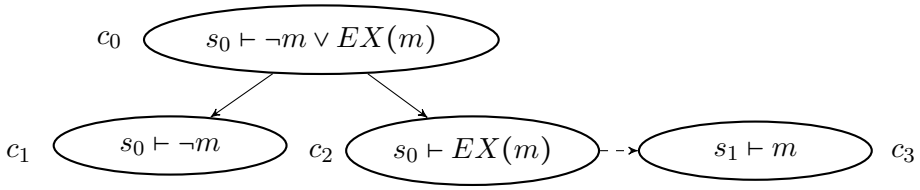

Figure 3.2: Example Arena for the example KMTS of Figure 3.1.

[1] apud Ribeiro (original author) in personal communication about the cited work, 11.28.2022

Definition 3.1 *[RA15] The game rules are defined as follows:*

(1) $\frac{s \vdash \psi_0 \vee \psi_1}{s \vdash \psi_i}$ *: i ∈ { 0,1 } (∃ve)*

(2) $\frac{s \vdash \psi_0 \wedge \psi_1}{s \vdash \psi_i}$ *: i ∈ { 0,1 } (∀belard)*

(3) $\frac{s \vdash EX\,\varphi}{t \vdash \varphi}$ *: (s,t) ∈ R^- (∃ve)*

(4) $\frac{s \vdash AX\,\varphi}{t \vdash \varphi}$ *: (s,t) ∈ R^- (∀belard)*

(5) $\frac{s \vdash A(\varphi_1 U \varphi_2)}{s \vdash \varphi_2 \vee (\varphi_1 \wedge AXA(\varphi_1 U \varphi_2))}$ *(∃ve)*

(6) $\frac{s \vdash E(\varphi_1 U \varphi_2)}{s \vdash \varphi_2 \vee (\varphi_1 \wedge EXE(\varphi_1 U \varphi_2))}$ *(∃ve)*

(7) $\frac{s \vdash A(\varphi_1 R \varphi_2)}{s \vdash \varphi_2 \wedge (\varphi_1 \vee AXA(\varphi_1 R \varphi_2))}$ *(∃ve)*

(8) $\frac{s \vdash E(\varphi_1 R \varphi_2)}{s \vdash \varphi_2 \wedge (\varphi_1 \vee EXE(\varphi_1 R \varphi_2))}$ *(∃ve)*

3.3 The Colouring

A colouring function is defined to map each configuration of the arena to a truth value. The truth value set for the initial or root configuration is then the model checking result. The colouring function is defined over a so-called maximum contraction function $\boldsymbol{\delta}$ which can be calculated bottom up for every configuration in the arena using the definitions made by Ribeiro and Andrade [RA15].

In order to be able to use the colouring function Definition 3.11, in which $\boldsymbol{\delta}$ is called maximum contraction, we first have to look at some basic operations.

As mentioned, we interpret a KMTS as a set of Kripke structures respective CTL models (see Figure 2.3 in Section 2.3). In order to avoid having to decide for every single one of these Kripke strcutures whether a KMTS satisfies a given CTl property, we can check the CTL properties directly on the KMTS M instead of doing this on the extensions $K(M)$. To be able to do this, we need a couple of operations over a KMTS which were first introduced by Ribeiro and Andrade [RA15]. These operations will be presented in the following section.

3.3.1 Basic Set Operations

Definition 3.2 *[RA15] Let M, M_1, M_2 be KMTSs, X_1 and X_2 two sets of changes, such that $M_1 \sqsubseteq M$, $M_2 \sqsubseteq M$ and $M_1 = M(X_1)$, $M_2 = M(X_2)$. We define the operations intersection, union and difference with respect to KMTSs as:*

Union: $M_1 \sqcup M_2 = \{M_1, M_2\}$

Intersection: $M_1 \sqcap M_2 = \begin{cases} \varnothing, & \textit{iff } X_1 \not\approx X_2 \\ \{M(X_1 \cup X_2)\}, & \textit{otherwise} \end{cases}$

Difference: $M_1 \setminus M_2 = \begin{cases} \{M_1\}, & \textit{iff } X_1 \not\approx X_2 \\ \bigcup\limits_{p_i \in (X_1 \setminus X_2)} \{M(X_1 \cup \{\neg p_i\})\}, & \textit{otherwise} \end{cases}$

Sometimes a set of Kripke structures represented by two KMTSs can also be represented by a single KMTS whose extensions do not contain all of the original Kripke structures. One can define a contraction operation, which is a special kind of union of the original two KMTSs, that creates this single KMTS. If two KMTSs cannot be contracted into one single KMTS, the result of the contraction operation will equal the union in Definition 3.2.

Definition 3.3 *[RA15] Let M be a KMTS, M_1 and M_2 instances of M generated, by the sets of changes X_1 respective X_2. The contraction operation, denoted by $M_1 \sqcup^+ M_2$, is defined as:*

$$M_1 \sqcup^+ M_2 = \begin{cases} \{M(X_1 \cap X_2)\}, & \textit{iff } X_1 \subseteq X_2 \textit{ or } X_2 \subseteq X_1 \textit{ or} \\ & \exists p \in X_1 \textit{ s.t. } \neg p \in X_2 \textit{ and} \\ & X_1 \setminus \{p\} = X_2 \setminus \{\neg p\} \\ \{M_1, M_2\}, & \textit{otherwise} \end{cases}$$

In order to be able to deal with a set of Kripke structures we need a KMTS which represents the extensions of these Kripke structures. But sometimes it is not possible to find one single KMTS that corresponds. In this case, we need a set of KMTSs and this is why we need to introduce a new term, the Partition Set.

Definition 3.4 *[RA15] Let M be a KMTS and Γ a set of instances of it. Γ is a Partition Set (PS) of M iff $\forall M_1, M_2 \in \Gamma \mid M_1 \sqcap M_2 = \varnothing$.*

Definition 3.5 *[RA15] Let Γ and Γ' be two sets of instances of a KMTS M. Γ and Γ' are equivalent, denoting by:*

$$\Gamma \equiv \Gamma', \textit{ iff } \bigcup_{M_i \in \Gamma} K(M_i) = \bigcup_{M_i \in \Gamma'} K(M_i).$$

Definition 3.6 *[RA15] Let Γ_1 and Γ_2 be two sets of instances of a KMTS. We define the difference ($\setminus\!\!\setminus$), intersection ($\sqcap$) and union ($\sqcup$) operations as:*

$$\Gamma_1 \setminus\!\!\setminus \Gamma_2 = \bigcup_{M_i \in \Gamma_1, M_k \in \Gamma_2} M_i \setminus M_k$$

$$\Gamma_1 \sqcap \Gamma_2 = \bigcup_{M_i \in \Gamma_1, M_k \in \Gamma_2} M_i \sqcap M_k$$

$$\Gamma_1 \sqcup \Gamma_2 = \Gamma_1 \cup PS(\Gamma_2 \setminus\!\setminus (\Gamma_1 \sqcap \Gamma_2))$$

where PS(Γ) is a Partition Set equivalent to a set of instances Γ.

A special Partition Set is a Full Partition Set.

Definition 3.7 *[RA15] Let* M *be a KMTS and* Γ *a set of instances of it. We say* Γ *is a Full Partition Set (FPS) of* M *iff* Γ *is a Partition Set and* $K(M) = \bigcup_{M_i \in \Gamma} K(M_i)$.

In addition we introduce a Maximal Partition Set.

Definition 3.8 *[RA15] Let* M *be a KMTS,* Γ *a PS of it and* Γ′ *a PS resulting from a finite number of contraction operations on* Γ*:*

$$\Gamma' = \sqcup^{+}(\Gamma)$$

is a Maximal Partition Set (MPS) if there is no more contraction operation possible on Γ.

We note: If a PS is a FPS, then the corresponding MPS will consist of exactly one KMTS, namely the KMTS M.

3.3.2 Maximum Contraction Function

Now that the basics are in place, we can go on to the calculation of the Maximum Contraction Function. But before that we need a little definition about reachable states.

Definition 3.9 *[RA15] Let M be a KMTS. The set of states reachable from a state* $s \in S$ *of M is the set:*

$$\vec{S}(s) = \{\, s' \in S \mid s \to s' \in R^{-} \}.$$

Definition 3.10 *[RA15] Let* M *be a KMTS,* s *and* s' *states of* M*,* φ *a CTL formula and* G *the arena for the model checking* $M, s_0 \vDash \varphi$*. Following from the game rules in Definition 3.1 (see the concerned rule numbers in round brackets) the Maximum Contraction Function* δ *is defined recursively, bottom up over the arena* G *starting at the atomic deadend configurations, as follows [RA15]:*

$$(0)\ \ \delta(s \vdash l) = \begin{cases} \{M\} & \text{iff } l \in L(s) \\ \varnothing & \text{iff } \neg l \in L(s) \\ \{M(\{P_3(s,l)\})\} & \text{otherwise} \end{cases}$$

$$(1)\ \ \delta(s \vdash \varphi_1 \vee \varphi_2) = \sqcup^{+}(\delta(s \vdash \varphi_1) \sqcup \delta(s \vdash \varphi_2))$$

$$(2)\ \ \delta(s \vdash \varphi_1 \wedge \varphi_2) = \sqcup^{+}(\delta(s \vdash \varphi_1) \sqcap \delta(s \vdash \varphi_2))$$

$$(3)\ \ \delta(s \vdash EX\varphi) = \sqcup^{+}(\bigsqcup^{\cdot}_{s' \in \vec{S}(s)} \delta(s' \vdash \varphi) \sqcap \{M(\{P_2(s, s')\})\})$$

$$(4)\ \ \delta(s \vdash AX\varphi) = \sqcup^{+}(\bigsqcap^{\cdot}_{s' \in \vec{S}(s)} \delta(s' \vdash \varphi) \sqcup \{M(\{P_1(s, s')\})\})$$

For the rules *(5)* to *(8)* no defined calculation of the Maximum Contraction Function $\boldsymbol{\delta}$ is necessary. These rules offer no choice for the players of the model checking game, because the configurations that follow from those rules only have one child configuration each. Thus, the values of the function $\boldsymbol{\delta}$ in these configurations are always equal to the value defined in their respective child configuration.

3.3.3 Colouring Function

All that is left to do is assigning every result of the Maximum Contraction Function $\boldsymbol{\delta}$ to one of the possible truth values $\top$ (*true*) $\bot$ (*false*) $\mathtt{I}$ (*indefinite*).

Definition 3.11 *[RA15] Let M be a KMTS, s states of M, φ a CTL formula. The contraction model checking is a Colouring Function χ defined as follows:*

$$\chi(s \vdash \varphi) = \begin{cases} \top & \textit{iff } \delta(s \vdash \varphi) = \{M\} \\ \bot & \textit{iff } \delta(s \vdash \varphi) = \varnothing \\ \mathtt{I} & \textit{otherwise} \end{cases}$$

The KMTS from Figure 3.1 using the arena of Figure 3.2 would, for the root configuration $\boldsymbol{c_0}$, evaluate in the model checking result *indefinite*. This can be derived bottom up by: Colouring Function $\boldsymbol{\chi}$ of configuration $\boldsymbol{c_1}$ is $\bot$. $\boldsymbol{\chi}$ of configuration $\boldsymbol{c_3}$ is $\mathtt{I}$, thus, by game rule *(3)*, Colouring Function $\boldsymbol{\chi}$ of configuration $\boldsymbol{c_2}$ is also $\mathtt{I}$. Following game rule *(1)* this results for the root configuration $\boldsymbol{c_0}$ in $\boldsymbol{\chi}$ equals to $\mathtt{I}$, i.e. *indefinite*.

3.4 Summary

Common model checking approaches, for example Grumberg et al. [GLLS07], do not exactly capture the Guerra et al. interpretation [GAW13] of partial information within a KMTS. They also do not work directly on a KMTS but on the Kripke structures of the expansion. As the amount of partial information within a KMTS grows, the number of these Kripke structures grows exponentially. Therefore, the consistency checks should be done directly on the original KMTS and not on the expansion set. Contraction Model Checking, on the other, hand works directly on a KMTS and captures exactly the Guerra et al. interpretation of partial information within a KMTS.

4 Pinpointing Conflicts

When we do not know whether the given model satisfies the given CTL formula, which means that the result of the model check against the KMTS is *indefinite*, we want to know which indeterminables exactly are the problem. To achieve this, the parts of the game arena that cause the problems, strictly speaking the edges, will be determined based on the rules introduced by Guerra et al. [GAW13]. These resulting edges are called *Failure Witnesses*.

Definition 4.1 *[GAW13] The following 5 rules define the Failure Witnesses:*

*(1) A genuine may edge coming from a configuration of type AX coloured **indefinite** to a child configuration coloured **false** or **indefinite**.*
*(2) A genuine may edge coming from a configuration of type EX coloured **indefinite** to a child configuration coloured **true** or **indefinite**.*
*(3) A must edge coming from a configuration of type EX coloured **indefinite** to a child configuration coloured **indefinite**.*
*(4) An edge coming from a configuration of type $s_i \vdash l \wedge \varphi$ to a child configuration $s_i \vdash l$ coloured with **indefinite**.*
*(5) An edge from a configuration of type $s_i \vdash l \vee \varphi$ where its child configuration $s_i \vdash l$ is coloured with **indefinite** and the other child is coloured **indefinite** or **false**.*

As it can be seen in Figure 3.2 the colouring of the arena results in *indefinite* for the root configuration c_0. If we now determine the Failure Witnesses who are causing this, we will find out it is the genuine may edge from configuration c_2 to c_3 marked with a dashed circle in Figure 4.1.

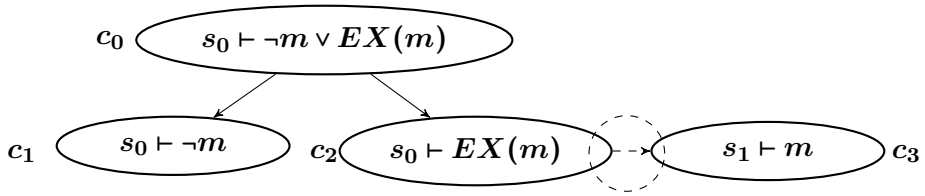

Figure 4.1: The failure witness is marked with a dashed circle.

Relevant is rule *(2)*. Assuming that the checked against CTL formula is an absolutely necessary requirement of the system, you can now try to change the system model in such a way that the requirement is met. In our case these two change operations on KMTS M would bring the desired result: $P_2(s_0, s_1) \rightarrow P_3(s_1, m)$. Guerra et al. [GAW13] have also suggested some rules and algorithms how to automate these adjustments of a system model to the determined failure witnesses. But after all, the system model could also be correct at this point and the CTL formula would have to be adjusted. Finally, whether this really make sense would have to be determined in further investigations and could, therefore, be the subject of further research.

5 Implementation Approach

In this thesis we implement the Contraction Model Checking [RA15] and use Failure Witnesses [GAW13] for pinpointing conflicts. This section is intended to give you an detailed insight about the implementation. Problems including solution approaches for solving these issues are also discussed.

5.1 General Procedure

Initially, without taking into account any possible, maybe necessary, optimisations the general approach of Algorithm 5.1 is to calculate the Maximum Contraction Function δ (Definition 3.10) for each configuration of the game arena. Afterwards we will convert this δ-function to a truth value out of *true*, *false* or *indefinite* and that is what we call the colour of the configuration. The arena will be previously build top down indexing the generated configurations in a depth first search order starting with index zero. Index zero is equivalent to the root configuration which is defined by $s_0 \vdash \varphi$. The top down approach is enforced by the game rules.

Algorithm 5.1 General Procedure

```
1:  global variables
2:      KMTS M
3:      Formula φ
4:      State initialState
5:      Game game
6:      Arena arena
7:
8:  procedure GENERALPROCEDURE
9:      (M, φ, initialState) ← readInputData
10:     game ← new Game
11:
12:     createArena(null, φ) /* top down */
13:
14:     for all created configurations c do /* bottom up */
15:         colourConfiguration(arena, c)
16:     end for
17:
18:     if root configuration coloured indefinite then
19:         for all edges do
20:             determineFailureWitness
21:         end for
22:     end if
23: end procedure
```

We will do the calculations bottom up. Bottom up means we can run through the configurations from the highest index down to zero. In this order deadend configurations are always calculated first before their parent configurations, regardless of their position in the tree as recommended by Guerra et al. [GAW13]. A deadend configuration is a configuration that does not reach any other configuration. This bottom up approach is enforced by the definition of the colouring function.

Assuming that the model checking result for the root configuration was *indefinite*, the next step would be to determine the Failure Witnesses responsible for this result. To do this, it is sufficient to run once through all edges of the arena. Each edge is checked against the five rules of Guerra et al. [GAW13] as listed in Definition 4.1.

5.1.1 Create the Arena

For a KMTS M, Formula φ, an initial state *initialState* and a current configuration c Algorithm 5.2 creates the arena.

Algorithm 5.2 Create the Arena

```
1:  return value type
2:      boolean
3:
4:  function CREATEARENA(Configuration currentConfig, Formula φ)
5:      if currentConfig == null then
6:          arena ← new Arena
7:          /* currentConfig is set to the root configuration */
8:          currentConfig ← new Configuration('c0', initialState, φ)
9:          arena.configurations.add(currentConfig)
10:     end if
11:
12:     return φ.createConfiguration(game, M, c)
13: end function
```

Strictly speaking, calling *createArena* does not create the whole arena but adds a new configuration to an existing arena. When it is the first call of the program run, two configurations are added to a new arena, the root configuration directly and the next configuration by the normal *createConfiguration* method of the concerned formula. All calls, except the first one, are recursive calls out of the called *createConfiguration* method. When the first call is finished, which means the first *createConfiguration* call comes successfully back, the arena is fully set up.

5.1.2 Fixed Point Calculation

Let us now have a look at the calculations of the Maximum Contraction Function δ itself. Doing it bottom up, as described, the calculation seems to be straightfor-

ward and always deducible from the respective child configurations. Yet, because of the apparently circular definitions of the game rules for the operators AU, EU, AR and ER, it is not. For example, a configuration $s \vdash E(\varphi_1 U \varphi_2)$ will be decomposed for the arena to configuration $s \vdash \varphi_2 \vee (\varphi_1 \wedge EXE(\varphi_1 U \varphi_2))$. It is obvious that after three more steps of decomposing, assumed we do not change the state in the EX-configuration, we again reach a configuration with $s \vdash E(\varphi_1 U \varphi_2)$. This is sometimes called the fixed point characterisation of CTL [HR04] because in case of these operators an incremental fixed point approach is used to break the cycle. Instead of adding an additional EU-configuration, we just attach an edge from the current EX-configuration to the last created EU-configuration with matching state and matching formula. An example would be the edge from $\boldsymbol{c_{14}}$ to $\boldsymbol{c_9}$ in Figure 5.6. This creates a loop we can cycle until all configurations within the loop are coloured and the last run brought no more changes as can be seen in Algorithm 5.3.

Algorithm 5.3 Colour the Configurations

```
1:  function COLOURCONFIGURATION(Arena arena, Configuration currentConfig)
2:      φ.calculateConfiguration(game, M, arena, currentConfig)
3:      setTruthValue(currentConfig)
4:
5:      if start of fixed point loop then
6:          repeat
7:              for all configurations l of the fixed point loop do /* bottom up */
8:                  φ.calculateConfiguration(game, M, arena, l)
9:                  setTruthValue(l)
10:             end for
11:
12:             if start of fixed point loop and l.delta not changed and not null then
13:                 fixed point found ← true
14:             end if
15:         until fixed point found
16:     end if
17: end function
```

In the case of EU and AU, we start the loop with the assumption that the fixed point will be the empty set $\varnothing$ because we are looking for the greatest fixed point and, if there is no greater one, we are already finished. Similarly, we start for ER and AR with $\{M\}$ because $\{M\}$ is the maximum possible result and this time we are looking for the least fixed point. Figure 5.1 shows the result of a fixed point calculation which took place while model checking the KMTS in Figure 5.9 with CTL formula for initial state $s_0 \vdash$ EX(m) $\vee$ E(m U $\neg$m). By colouring the arena bottom up the start of a fixed point loop is reached at configuration $\boldsymbol{c_9}$ and delta of $\boldsymbol{c_9}$ is initialised with the empty set $\varnothing$. After one loop delta of $\boldsymbol{c_9}$ reads $\{M(\{P3(s_1, \neg m)\})\}$ and after the next loop this result does not change. Thus, this result is fixed, colouring can go on.

```
c14 , c12 , ___ , null              , U , s1 ⊢ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})}   , I , s1 ⊢ m , deadend
c12 , c10 ,     , null              , U , s1 ⊢ (m ∧ EX(E(m U ¬m)))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c10 , c9  ,     , null              , U , s1 ⊢ (¬m ∨ (m ∧ EX(E(m U ¬m))))
c9  , c8  , c14 , ∅                 , U , s1 ⊢ E(m U ¬m)
********** Loop starting: c9 , c14
c14 , c12 , ___ , ∅                 , ⊥ , s1 ⊢ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})}   , I , s1 ⊢ m , deadend
c12 , c10 ,     , ∅                 , ⊥ , s1 ⊢ (m ∧ EX(E(m U ¬m)))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c10 , c9  ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ (¬m ∨ (m ∧ EX(E(m U ¬m))))
c9  , c8  , c14 , {M({P3(s1,¬m)})} , I , s1 ⊢ E(m U ¬m)
********** Loop starting: c9 , c14
c14 , c12 , ___ , {M({P3(s1,¬m)})} , I , s1 ⊢ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})}   , I , s1 ⊢ m , deadend
c12 , c10 ,     , ∅                 , ⊥ , s1 ⊢ (m ∧ EX(E(m U ¬m)))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c10 , c9  ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ (¬m ∨ (m ∧ EX(E(m U ¬m))))
c9  , c8  , c14 , {M({P3(s1,¬m)})} , I , s1 ⊢ E(m U ¬m)
********** Loop ended:
```

Figure 5.1: Program output of a fixed point calculation.

5.1.3 Determine Failure Witnesses

All Failure Witnesses, respective all edges recognised as Failure Witnesses, are marked accordingly and will be listed as program output. The KMTS in Figure 5.2, checked for the already known CTL formula $(EXm) \vee E(mU\neg m)$, would result in the output shown in Figure 5.3. All edges marked as Failure Witnesses are there listed according to their from-configuration and their to-configuration, both in combination with the concerned states.

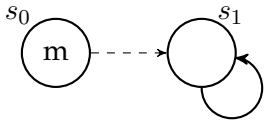

Figure 5.2: Input KMTS for the Failure Witnesses example.

```
********** Model Checking Result:
c0 ,    ,     , {M({P2(s0,s1)})} , I , s0 ⊢ (EX(m) ∨ E(m U ¬m)) ,
********** Failure Witnesses:
from:state , to:state , rule , must
c12:s1     , c13:s1   , 4    , true
c10:s1     , c11:s1   , 5    , true
c8:s0      , c9:s1    , 2    , false
c1:s0      , c2:s1    , 2    , false
****************************************
```

Figure 5.3: Failure Witnesses as listed by program output.

The responsible rule and whether it is a must edge or not is also displayed. In Figure 5.4 we see the most important parts of the arena for this model checking game. Let us have a look at the edge($\boldsymbol{c_1}, \boldsymbol{c_2}$), the first Failure Witness. It is a genuine may edge derived from the may transition($\boldsymbol{s_0}, \boldsymbol{s_1}$) in the KMTS of Figure 5.2. Therefore, rule 2 applies[2]. For the second found Failure Witness edge($\boldsymbol{c_{10}}, \boldsymbol{c_{11}}$) rule 5 applies[3], because the label of state $\boldsymbol{s_1}$ is not defined. Thus, configuration $\boldsymbol{c_{11}}$ is coloured *indefinite* and configuration $\boldsymbol{c_{12}}$ is coloured *false*. Right now, you have to belief the colour of $\boldsymbol{c_{12}}$. We will see this later.

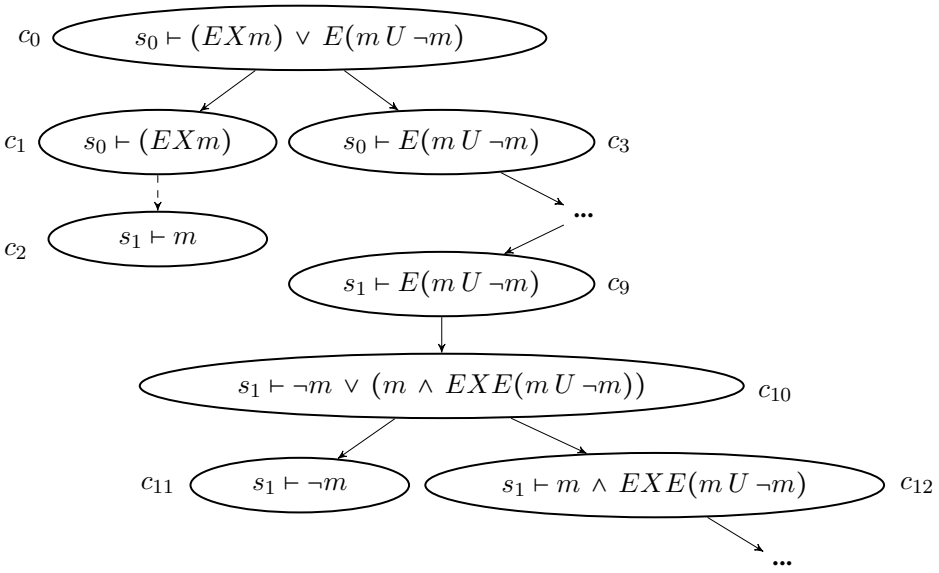

Figure 5.4: Concerned parts of the arena for the Failure Witnesses example.

5.2 Optimisation Strategies

In this section, we will at least think about the existing possibilities to optimise the construction of the arena as well as the calculations of the individual configurations. Some possibilities are obvious, but let us start with some considerations about the to be expected size of the game arena while, at the end, we do a short discussion of this optimisation ideas.

5.2.1 The Size of the Arena

All the calculation is done on the game arena. This arena is created, as already mentioned, by decomposing the given CTL formula while applying the rules for the

[2]Rule 2: A genuine may edge, coming from a configuration of type EX coloured *indefinite*, to a child configuration coloured *true* or *indefinite*.

[3]Rule 5: An edge from a configuration of type $\boldsymbol{s_i \vdash l \vee \varphi}$ where its child configuration $\boldsymbol{s_i \vdash l}$ is coloured with *indefinite* and the other child is coloured *indefinite* or *false*.

model checking game. Therefore, the size and complexity of the arena depends on the length of the CTL formula i.e. the number of subformulas the given formula φ can be split into as well as the given KMTS to be checked. Every operator AU, EU, AR or ER adds at least five more configurations to the arena according to the game rules. Five only if both formulas before and after the U, respective R, are atomic otherwise there will eventually be much more. Likewise, every operator EX or AX multiplies the amount of configurations anywhere below itself by the number of existing outgoing transitions from the current state.

Based on the commonly known so-called *state explosion problem*, the state spaces tend to become very large. Indeed increasing their size exponentially with the number of variables or components of the underlying system. We could call this the *configuration explosion problem*. Suggestions to overcome the *state explosion problem*, with the exception of the BDD data structures[4] (Binary Digital Diagrams, compare e.g. [CG18].) not discussed further here (details see in footnote), are mostly reduced to abstraction, partial order reduction, induction or composition. Without going into detail, the common point of these methods is keeping the models simple or at least breaking them down into several simple models (see e.g. [HR04]).

```
********** Coloured Arena:
configuration , parent , fploop , delta , colour , state ⊢ formula, info
c0  ,     ,     , {M}              , I , s0 ⊢ (EX(m) ∨ E(m U ¬m))
c1  , c0  ,     , {M({P3(s1,m)})}  , I , s0 ⊢ EX(m)
c2  , c1  ,     , {M({P3(s1,m)})}  , I , s1 ⊢ m , deadend , may
c3  , c0  ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ E(m U ¬m)
c4  , c3  ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ (¬m ∨ (m  ∧ EX(E(m U ¬m))))
c5  , c4  ,     ,  ∅               , ⊥ , s0 ⊢ ¬m , deadend
c6  , c4  ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ (m ∧ EX(E(m U  ¬m)))
c7  , c6  ,     , {M}              , ⊤ , s0 ⊢ m , deadend
c8  , c6  ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ EX(E(m U ¬m))
c9  , c8  , c14 , {M({P3(s1,¬m)})} , I , s1 ⊢ E(m U ¬m) , may
c10 , c9  ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ (¬m ∨ (m  ∧ EX(E(m U ¬m))))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c12 , c10 ,     ,  ∅               , ⊥ , s1 ⊢ (m ∧ EX(E(m U  ¬m)))
c13 , c12 ,     , {M({P3(s1,m)})}  , I , s1 ⊢ m , deadend
c14 , c12 , ___ , {M({P3(s1,¬m)})} , I , s1 ⊢ EX(E(m U ¬m))
****************************************
```

Figure 5.5: Coloured arena as listed by program output when model checking KMTS M of Figure 5.2.

[4] Although BDD-based model checking, often called symbolic model checking, is commonly seen as the state of the art technique, it does not seem to be a recommended option for this thesis. A lot of research would have to be done how and to what extent Contraction Model Checking over a KMTS representing partial information and the with Contraction Model Checking associated special operations on sets can be mapped with symbolic model checking. That would be far beyond the scope of this thesis. There is also another knockout criterion. We, this way, would not have any game arena with the help of whose edges we could determine the Failure Witnesses. Nevertheless, further research work to combine Contraction Model Checking and Failure Witnesses with the possibilities of symbolic model checking would be a promising perspective.

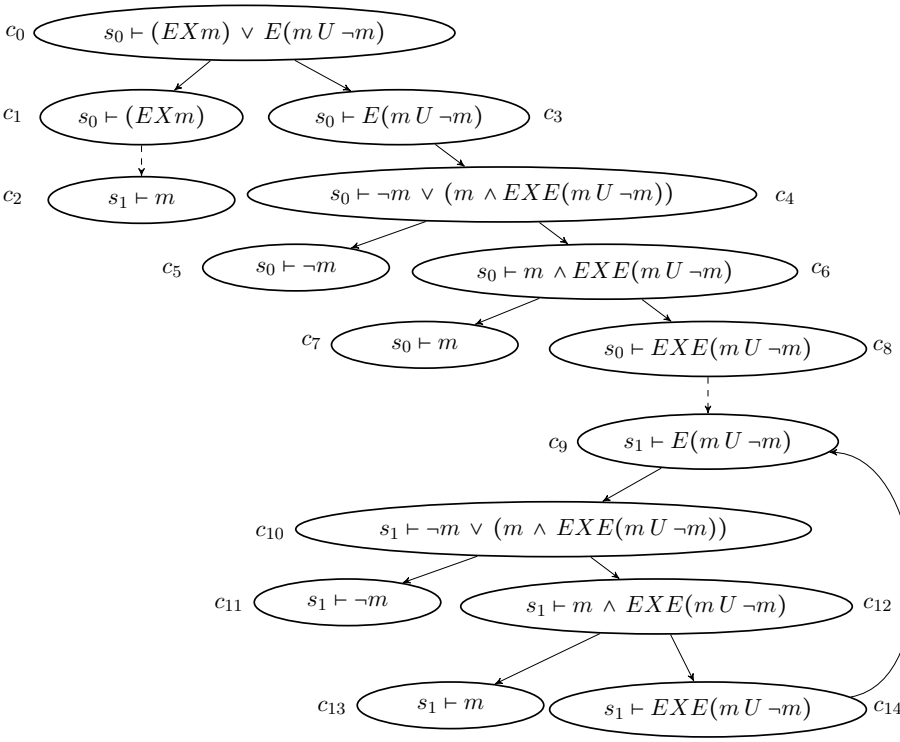

Figure 5.6: Complete arena when model checking KMTS M of Figure 5.2.

That means, we should expect our game arena to be quite big whatever quite big will mean. It is shown by Figure 5.2 that even the small KMTS with only two states depending on the checked for CTL formula already results in an arena with 15 configurations as can be seen in Figure 5.6. Figure 5.5 shows the same arena as it would be listed during a program run after the colouring of the configurations. The columns of the list, from left to right, are the current configuration with $\boldsymbol{c_0}$ as the root configuration and the parent configuration, of course empty for $\boldsymbol{c_0}$. The next column shows the end configuration of a fixpoint calculation loop which starts at the current configuration in this line. Look at the edge from $\boldsymbol{c_{14}}$ to $\boldsymbol{c_9}$ in Figure 5.6. Column 4 contains the Maximum Contraction Function $\boldsymbol{\delta}$ calculated for the current configuration and column 5 the colour of the configuration (*true*, *false*, or *indefinite*) resulting from $\boldsymbol{\delta}$. State and formula that define the current configuration finalise the list.

5.2.2 Junctions

Subsequently, we will call configurations with more than one child configuration junctions. That can be disjunctions, conjunctions as well as EX-junctions or AX-junctions, depending on the formula that defines the configuration. With us, disjunctions and conjunctions always have exactly two children, because the relevant CTL operators defined in Section 2.1.1 do not allow anything else. Any disjunctions containing true or conjunctions containing false can be immediately decided

[SK05]. More specifically, disjunctions will be decided as *true* if one side, respective one child configuration, is *true* without calculating the other side. Conjunctions will be decided as *false* if one side, respective one child configuration, is *false* without calculating the other side. Thus, there is always a chance to colour these junctions without having calculated both children. The same applies to EX-junctions and AX-junctions. EX-junctions can be optimised in a similar way as disjunctions and AX-junctions in a similar way as conjunctions. Although these junctions can have more than just two children, namely as many as transitions emanate from the affected state, again one *true* coloured child decides an EX-junction and one *false* coloured child decides an AX-junction.

The more partial respective unknown information exists in a KMTS, the more likely it is that *indefinite* results will occur. But, *indefinite* on one side can never be used to determine the result of a junction. Therefore, the chances of not having to calculate all children or, in other words, of finding shortcuts when calculating the configurations decrease with the number of indeterminations occurring in the KMTS. Nevertheless, this optimisations have definitely to be done, not least because at runtime they generate almost no overhead of any kind even with bad luck at every junction.

5.2.3 On-the-Fly

Consequently thought ahead, for the arena parts below one side of a junction configuration you could not only save on the calculations of the Maximum Contraction Function, as shown in the last section, but you could also avoid creating the arena for this part at all. This strategy would be called on-the-fly construction of the game arena based on the on-the-fly model check approaches for state spaces which can be seen for example in [HKV96] or [LLM14]. It means, while the arena is being set up, the configurations are calculated at the same time. In this way, you can achieve that if you do not have to calculate a configuration, you will not even create this configuration. Unfortunately, as already mentioned in Section 5.1, creating the arena by decomposing can only be done top down while calculating the Maximum Contraction Function according to the calculation rules of Definition 3.10 has to be done bottom up. That is why, creating and calculating can not really be done in parallel.

A solution might be finding partial trees within the arena configuration tree while creating the arena in depth first order. The top configuration of such a partial tree would be the child of a junction configuration. The bottom configuration, in general, will be a deadend. We can then calculate from the deadend to the top of the partial tree and the result will be the truth value of one child of the junction configuration above. Now, as already shown, there is a chance that creating and calculating any other child is no longer necessary. Using on-the-fly, we are creating the arena and colouring the configurations simultaneously. Whenever *creating the arena top down* reaches a deadend, we start *colouring configurations bottom up* starting

from this deadend configuration. And, whenever *colouring configurations bottom up* reaches the child of a junction that cannot be coloured yet, we will start *creating the arena top down* from the next child, if a next child exists. If no next child exists we will go on up with the normal colouring of the parent junction.

5.2.4 Applying some Optimisations

Algorithm 5.4 General Procedure (optimised)

```
 1: global variables
 2:     KMTS M
 3:     Formula φ
 4:     State initialState
 5:     Game game
 6:     Arena arena
 7:
 8: procedure GENERALPROCEDURE
 9:     (M, φ, initialState) ← readInputData
10:     game ← new Game
11:     createArena(null, φ)
12:
13:     if root configuration coloured indefinite then
14:         for all edges "bottom up" do
15:             determineFailureWitnesses
16:         end for
17:     end if
18: end procedure
```

Please note that the loop for the *colourConfiguration* method call has vanished in this optimised version of the *GeneralProcedure* in Algorithm 5.4. As can be seen in Algorithm 5.5, *colourConfiguration* is now already called within *createArena*. This makes it possible to pass on the two output possibilities of *colourConfiguration* in Algorithm 5.6 to *createArena* and to use them directly there as return value. The two possibilities are returned as a boolean type where *goOnRight* corresponds to *true* and *goOnUp* corresponds to *false*. After returning from *createArena*, *goOnUp* means that the parent configuration should be processed next, respective should be coloured. *goOnRight* means that it should be continued with the next child configuration to the right of the configuration which has just been edited. In this case, to continue means that this child configuration should now be created. Of course, only if there was another child at all.

Algorithm 5.5 Create the Arena (optimised)

```
1:  return value type
2:      boolean
3:
4:  function CREATEARENA(Configuration currentConfig, Formula φ)
5:      if currentConfig == null then
6:          arena ← new Arena
7:          /* currentConfig is set to the root configuration */
8:          currentConfig ← new Configuration('c0', initialState, φ)
9:          arena.configurations.add(currentConfig)
10:     end if
11:
12:     if φ.createConfiguration(game, M, currentConfig) then
13:         return φ.colourConfiguration(arena, currentConfig)
14:     else
15:         return goOnUp /* behind the bottom end of a fixed point loop */
16:     end if
17: end function
```

This combination of *createConfiguration* and *colourConfiguration* in createArena results in exactly that on-the-fly optimisation process described in Section 5.2.3. The only exception is when *createConfiguration* returns *false*. However, this only happens if the configuration to be created is the end of a fixed point loop, triggered by one of the CTL formulas AU, EU, AR or ER. In this case, it also continues upwards but not to the parent configuration but further up to the start of the fixed point loop.

The logic of the optimisation of junctions described in Section 5.2.2, on the other hand, takes place in *colourConfiguration* in Algorithm 5.6. It is processed in the If-statement between lines 21 and 28 and is actually self-explanatory, not least because of the comments in the pseudocode.

Because, in a depth first order, the left operand of a dis- or conjunction, respective the leftmost child of an EX-/AX-junction, will be handled first during the arena creation process, it is probably advisable to decompose junctions in a way that the *smaller* subformula will be placed on the left side. Of course a definition is needed what smaller means. An approach could be:

$$Atom < Or, And < EX, AX < EU, AU, ER, AR$$

That would ensure for the creation of the arena as well as the colouring of the configurations that the side where we expect to get a result faster is always processed first.

Algorithm 5.6 Colour the Configurations (optimised)

```
1: return value type
2:     boolean
3:
4: function COLOURCONFIGURATION(Arena arena, Configuration currentConfig)
5:     φ.calculateConfiguration(game, M, arena, currentConfig)
6:     setTruthValue(currentConfig)
7:
8:     if start of fixed point loop then
9:         repeat
10:            for all configurations l of the fixed point loop do /* bottom up */
11:                φ.calculateConfiguration(game, M, arena, l)
12:                setTruthValue(l)
13:            end for
14:
15:            if l.delta not changed and not null then
16:                fixed point found ← true
17:            end if
18:        until fixed point found
19:    end if
20:
21:    if parent of currentConfig is a junction then /* And, Or, AX, EX */
22:        if parent of currentConfig is (And or AX) and colour is false or
23:          parent of currentConfig is (OR or EX) and colour is true then
24:            return goOnUp /* this is a shortcut */
25:        else
26:            return goOnRight /* this goes to the next child if one exists */
27:        end if
28:    end if
29:
30:    return goOnUp /* this is the normal exit upwards */
31: end function
```

5.2.5 Discussion of Optimisation Strategies

One possible disadvantage of the above optimisation approaches should be noted. At least there should be some thoughts allowed whether it is a problem or not. One of the main points of the thesis is exactly pinpointing the Failure Witnesses. This would be done following the model checking in the case of an *indefinite* result. Implementing some of the above mentioned optimisations, e.g. on-the-fly, would mean we can only determine the Failure Witnesses for the part of the arena that was calculated at all. It is hard to say off the cuff whether, therefore, results will be lost or not. Nevertheless, we decided to include the disjunction and conjunction optimi-

sations as well as the on-the-fly optimisation into the implementation right from the start.

The most natural way, not to say the most elegant and also clearest way regarding the code, to build the game arena by decomposing the CTL formulas is surely to do it by recursive calls for every configuration top down. But, during evaluation, we should keep in mind that with the expected big arenas, by crossing an upper limit depending mainly on the available memory, stack overflow exceptions are likely to occur. Therefore, if necessary and possible, there should be a plan B like using iterations instead of recursion. Perhaps this could be a topic for further research work, eventually combined with the above mentioned BDDs, since recursive decomposition is usually how it is done there too.

Future applications of the program will show whether further optimisations in terms of time or space are necessary at all. Then larger amounts of data are to be processed using a code that has, in scope of this thesis, been extensively tested for precisely circumscribed smaller input cases (see Section 8 Future Work). What we can say right now is that Ribeiro and Andrade [RA15] have claimed the algorithms necessary for CMC as NP-complete. Other authors like Schnoebelen [Sch02] or Acar et al. [ABM19] come to a result of P-complete for the normal model checking problem using a model and a CTL formula. The latter explicitly mention the corresponding satisfiability problem would be NP-complete. Shoham and Grumberg [SG12] name for a 3-valued model checking a complexity of $O(|M| \times |\varphi|)$ where $|M| = |S| + |R|$ and $|\varphi|$="number of all possible subformulas of φ". This is in linear-time with respect to the size of the game arena. Another variant by Chechik et al. [CDEG04] also considers fixed point processing The actually impossible worst case would be everytime and with a loop over all states. This would be complexity $O(|S| \times |M| \times |\varphi|)$ = $O(|S|^2 \times |R| \times |\varphi|)$. All these different complexities are polynomial and thus at least in the complexity class of P.

At first glance we colour an arena just like Shoham and Grumberg [SG12], since the existence of indeterminations in the model makes no difference in the size of the derived arena. We use fixed point loops like Chechik et al. [CDEG04]. Even considering that our fixed point worst case would loop not only all states but the whole arena the complexity still would be something polynomial like $O((|M| \times |\varphi|)^2)$. And still the question is, will CMC behave similar to these examples or are there some major differences that make it more complex? Maybe we cannot be sure how long our fixed point processing needs to converge, because we do not wait for a fixed truth value but for a no more changing δ-function? At least in case of *indefinite* that could be a difference. We cannot and do not want to clarify this questions here and now, and it would not be appropriate to make assumptions. The topic is not decisive for the current application of our algorithms and should there be further developments in the future, the question must be asked again.

5.3 General Considerations

Probably the most important decision for the implementation of the Contraction Model Checking was about what programming language will be used. If that was a strategic decision for a new professional software system, a little market research about current trends would be appropriate. Thus, let us have a look at the current most popular languages. There are several indices that try to reflect this. One of the most common and most recognised is the TIOBE Programming Community index released monthly by the software company TIOBE since 2001. It is based on hundreds of different sources and uses as metrics, for example, the number of experienced engineers worldwide, the number of courses available and the popularity of the language in search engines. The current edition of the index from march 2023 [TIO23] shows a head-to-head race for the lead. Python, C, Java and C-derivates like C++ or C# occupy the places 1 to 4 with only minimal differences to each other. Together these programming languages represent a market share of more than 60 %.

Another argument in favor of the decision would, of course, be a technical advantage of one of the languages for the planned implementation. Without extensive trials and tests suitability is difficult to reveal. But in our implementation we can expect decomposing CTL formulas combined with running through the configurations of the arena and that will almost certainly lead to deep recursive method calls. Unfortunately, as far as I know, the languages mentioned do not differ much in this area. Sooner or later all the languages will respond to too many open recursive calls with stack overflow errors [NC13].

Therefore, since there is no compelling argument for one of these programming languages, the decision is to code thy implementation in Java. This is the programming language I was at least able to gain some experience through lectures and internships during my studies. Java is a object-oriented programming language designed by James Gosling for Sun Microsystems. The first official release came in 1996. 2010 Sun Microsystems was taken over by Oracle. Since then, Oracle continues to develop Java on a regular basis [Par20]. We use the JDK (Java Development Kit) in version Java SE 18 and as development environment the Eclipse IDE (Integrated Development Environment) in version 4.23 from 2022-03. This Eclipse version restricts the Java functionalities mostly to the level of SE 17 which was the last so-called LTS (Long-Term Support) version released by Oracle [Ora23].

5.4 Program Structure

Our Java implementation has got a specific structure which we will elaborate on in this section. As usual in an Java/Eclipse environment the source code is placed in a high level package called **src**. Below *src* the system is split into five packages. The five packages are:

game	Everything needed to start and run a model checking game including setting up the game arena.
model	Objects necessary to build a KMTS model.
ctl	Objects necessary to build a CTL formula.
input	Different data types for reading the input data.
utilities	Utilities for set theory operations, handling json files etc..

Let us now take a look at the different packages on a more detailed level.

The package **game** contains two classes to start a model checking game. One is called *Game*, and the other one is called *ModelChecking*. The class *ModelChecking* contains the *main* method of the system. Here, the input data from a json file is read, and the game is actually started by creating an instance of the *Game* class. The class *Game* can be seen as the real place where everything around a running model checking game happens. The arena is build, the configurations are coloured and, eventually, Failure Witnesses pinpoint the conflicting areas.

The package **model** contains classes needed to represent a model, respective to represent a KMTS with states, labels and transitions.

The package **ctl** contains an interface *Formula* and classes for every single possible operand that can occur in a CTL formula. All these classes implement the interface *Formula*. The two most important methods this interface offers are *createConfiguration* to add a new configuration to the arena and *calculateConfiguration* to calculate the Maximum Contraction Function for a configuration (see Definition 3.10).

The package **input** contains data types into which the contents of the json input file can be read to then be distributed into the actual target classes in the *model* and *ctl* packages.

The package **utilities** mainly contains a class *JsonUtilities* for handling json files and a class *SetTheory* that contains all pure set theory methods for the Contraction Model Checking.

5.5 Input Specification

The functional input of the implemented algorithm will be a KMTS M, a CTL formula φ and an initial state s_0. This input for the program is written in Json files. The exact format of these files can be seen in Listings 5.1 and 5.3 for a small example. As already mentioned in Section 2.3, deviating from the general definition of a KMTS (see Definition 2.5), we do not use a set of initial states but only exactly one initial state **"SI"** as input. Furthermore, it is noteworthy that the index for the value of the labels of the states of **"S"** which selects one of the atomic propositions of **"AP"** does

not start with zero as usual in Java and like all other indices in the Json file. That is because a *not* before an atomic proposition would be marked as a negative value of the index. This would not be possible with an index starting at zero, at least not for the starting index. To keep it flexible, especially for testing, there is a third technical input file which is also in Json format. This file has to be named *runcontrol*. The file names for the two KMTS and CTL files are defined there, a switch *opti* can be used to deactivate all optimisations discussed earlier as well as a detailed control of the runs log file scope. The switch *rc* turns on and off the log of the runcontrol file itself, *rt* controls the displays of timings at some meaningful points of the process. *prepro* is responsible for the formula pre-processing, *fploop* for detailed information about the fixed point loops, *arena* shows the completet and coloured arena and *stats* brings some statistics about the run. An example can be seen in Listing 5.2. In all three files a slash at position one of a line declares a comment line.

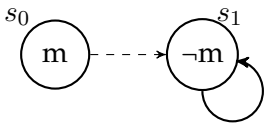

Figure 5.7: Input KMTS of Listing 5.1.

Listing 5.1 would be the input file for the KMTS in Figure 5.7 and Listing 5.3 would load the CTL formula $(EXm) \vee E(m\,U\,\neg m)$. Next, we will have a look at the classes needed to represent a KMTS and in the following understand how to create instances of these classes from our Json file to get a complete KMTS to work with.

```
/Input KMTS
{
      /Atomic Propositions
      "AP":       [{"label":"m"}],
      /States (Labels defined by 'value' as index of AP starting at 1)
      /       ('not' marked as negativ value)
      "S":        [{"name":"s0","label":[{"value":"1"}]},
                  {"name":"s1","label":[{"value":"-1"}]}],
      /Initial State (defined by index of S starting at 0)
      "SI":       "0",
      /Transitions (States defined by index of S starting at 0)
      "T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"1"}],
      /Must Transitions (Transitions defined by index of T starting at 0)
      "Rplus":    [{"transition":"1"}],
      /May Transitions (Transitions defined by index of T starting at 0)
      "Rminus":   [{"transition":"0"},
                  {"transition":"1"}]
}
```

Listing 5.1: Input Json File for KMTS

```
/Input Runcontrol
{
      /Name of KMTS and CTL Formula file (without file type; file type has to be .json)
      "kmtsfile": "KMTS1",
      "ctlfile":  "CTL1",
      /Apply Optimisations ('1': with Optimisations; '0': without Optimisations)
      "opti":     "1",
      /Log Options ('1': log it; '0': do not log it)
      /Log Runcontrol
      "rc": "1",
      /Log Run Timings
      "rt": "1",
      /Log Preprocessing
      "prepro":   "1",
      /Log Fixed Point Loops
      "fploop":   "1",
      /Log Coloured Arena
      "arena":    "1",
      /Log Statistics
      "stats":    "1"
}
```

Listing 5.2: Input Json File for Runcontrol

```
/Input CTL Formula
{
      /CTL Formula)
      "CTL":      "OR(EX('m'),EU('m',NOT('m')))"
}
```

Listing 5.3: Input Json File for CTL Formula

5.5.1 Basic KMTS Data Types

The Listings of the basic data types used for representation of a KMTS show only the data structure of these types and not the constructors or other code defined within these classes.

```
1 public class KMTS {
2       ArrayList<Atom> AP;
3       ArrayList<State> S;
4       ArrayList<Transition> Rplus;
5       ArrayList<Transition> Rminus;
6       ArrayList<Change> X;
7 }
```

Listing 5.4: Class KMTS

```
1 public class State {
2       String name;
3       ArrayList<Formula> label;
4 }
```

Listing 5.5: Class State

```
1 public class Transition {
2       State stateFrom;
3       State stateTo;
4 }
```

Listing 5.6: Class Transition

```
1  public class Change {
2        CType type;
3        State state;
4        State stateTo;
5        Formula label;
6  }
```

Listing 5.7: Class Change

```
1  public class Atom {
2        String label;
3  }
```

Listing 5.8: Class Atom

The enum *CType*, as used in the class *Change*, offers the three possible characteristics of a change as described in Section 2.3.1. These are the three primitive change operations P_1, P_2 and P_3.

5.5.2 Json Handling

First, we load the runcontrol file into the data structure *InputFileRunControl* of Listing 5.9.

```
1  /**
2  * Structure of input runcontrol
3  * (loaded from a json-file called
4  * runcontrol.json).
5  */
6  public class InputFileRunControl {
7    public String kmtsfile;
8    public String ctlfile;
9    public int opti;
10   public int rc;
11   public int rt;
12   public int prepro;
13   public int fploop;
14   public int arena;
15   public int stats;
```

Listing 5.9: Class InputFileRunControl

```
1  /**
2  * Structure of input data for KMTS
3  * (loaded from a json-file).
4  */
5  public class InputFileKMTS {
6    Atom [] AP;
7    InputState [] S;
8    int SI;
9    InputTransition [] T;
10   ListTransition [] Rplus;
11   ListTransition [] Rminus;
12 }
```

Listing 5.10: Class InputFileKMTS

All the KMTS data will be loaded from the Json file named in the *runcontrol* into a defined structure called *InputFileKMTS* which is shown in Listing 5.10. Loading means, as of course already for the *runcontrol* file, that we first read the Json file into a string variable. Then the content of this string, respective the Json structure from the file, will be directly parsed into the data type structure of class *InputFileKMTS*. This works with the help of the high level api (application program interface) of Gson. This api is able to parse data from Json structures to object oriented Java structures, i.e. classes or objects, and vice versa. A Json object is understood here as an instance of a specific class, with the members then being mapped to instance variables of the class. We use Gson in version 2.8.6 [SLW23]. With this input we can generate a new KMTS with the help of the class *KMTS*, as shown in Section 5.5.1, and mark one state as the initial state for the game. Similarly, we load the CTL formula from the

Json file in Listing 5.3 via the structure *InputFileCTL* shown in Listing 5.11. All that is left now is to convert the CTL formula into a workable form for our program.

```
1 /**
2 * Structure of input data for CTL
3 * formula (loaded from a json-file).
4 */
5 public class InputFileCTL {
6   public String CTL;
7 }
```

Listing 5.11: Class InputFileCTL

5.5.3 Logical Symbol Interpretation for CTL Formulas

In order to be recognised and processed by our program, the logical symbols of a CTL formula on the left side of the arrows have to be written in a certain way for the input file. You can see the required format on the right side. This format for representing CTL formulas in program code is inspired by some small but committed projects on GitHub, the well known collaboration platform in the net [Bre22, CFR22].

(1)	$\top$	$\Rightarrow$	TRUE
(2)	$\bot$	$\Rightarrow$	FALSE
(3)	l	$\Rightarrow$	'l'
(4)	$\neg\varphi$	$\Rightarrow$	NOT(φ)
(5)	$\varphi_1 \vee \varphi_2$	$\Rightarrow$	OR(φ_1, φ_2)
(6)	$\varphi_1 \wedge \varphi_2$	$\Rightarrow$	AND(φ_1, φ_2)
(7)	$\varphi_1 \rightarrow \varphi_2$	$\Rightarrow$	IMP(φ_1, φ_2)
(8)	$EX\varphi$	$\Rightarrow$	EX(φ)
(9)	$AX\varphi$	$\Rightarrow$	AX(φ)
(10)	$EF\varphi$	$\Rightarrow$	EF(φ)
(11)	$AF\varphi$	$\Rightarrow$	AF(φ)
(12)	$EG\varphi$	$\Rightarrow$	EG(φ)
(13)	$AG\varphi$	$\Rightarrow$	AG(φ)
(14)	$E(\varphi_1 U \varphi_2)$	$\Rightarrow$	EU(φ_1, φ_2)
(15)	$A(\varphi_1 U \varphi_2)$	$\Rightarrow$	AU(φ_1, φ_2)
(16)	$E(\varphi_1 R \varphi_2)$	$\Rightarrow$	ER(φ_1, φ_2)
(17)	$A(\varphi_1 R \varphi_2)$	$\Rightarrow$	AR(φ_1, φ_2)

That is why in Listing 5.3 the CTL formula had to be entered as follows:

$(EXm) \vee E(m\,U\,\neg m)$	$\Rightarrow$	OR(EX('m'),EU('m',NOT('m')))

This input CTL formula can now be stored and processed during the game as *Formula* using the classes in the *ctl* package that implement the *Formula* interface..

5.5.4 Pre-Processing

The game rules defined by Ribeiro and Andrade [RA15] for Contraction Model Checking can only handle CTL formulas in negation normal form and also only special operators of CTL (see Definition 2.2). Thus, there are two possibilities. Either we restrict the input exactly to the processable CTL operators and forms or we convert the input into a processable CTL formula with the help of the transformations shown in Section 2.1.1. We chose the second possibility, because, as already mentioned, some of the missing operators are very common. Therefore, every input CTL formula has first to be subject of some pre-processing. This pre-processing consists of two stages. The first stage reduces a given CTL formula to the valid operators for Contraction Model Checking through equivalence transformations. Some examples would be (compare [HR04]):

the formula	$\varphi_1 \rightarrow \varphi_2$	would be transformed to	$\neg\varphi_1 \vee \varphi_2$;
the formula	$AF\varphi$	would be transformed to	$A(\top U \varphi)$.

The second phase brings the resulting CTL formula into negation normal form. The algorithm we use for this is inspired by [Ren23]. Figure 5.8 shows the result of such a pre-processing using a small and clear example in the output form as it will be logged by the program.

```
Input Initial State: s1
Input KMTS-AP: Start, Close, Heat, Error,
Input CTL Formula: AG((Start ∨ Error) → AFHeat)
Contraction Model Checking Form: A((¬(Start ∨ Error) ∨ A(⊤ U Heat)) R ⊥)
Negation Normal Form: A(((¬Start ∧ ¬Error) ∨ A(⊤ U Heat)) R ⊥)
****************************************
```

Figure 5.8: Program output of pre-processing a CTL formula.

5.5.5 Colour the Configurations

As shown in Section 3.3.1, we need quite a lot of special set operations, all defined by Ribeiro and Andrade [RA15], to be able to calculate the Maximum Contraction Function δ. The implementation of all these operations is stored centrally in the *SetTheory* class of the *utilities* package. This class is used as a library for all set theory calculations. An overview of the content of this set operations library follows:

Operations on Sets of Changes

- unionArrayListChanges(ArrayList<Change>, ArrayList<Change>)
  returns ArrayList<Change>, defines Operator (∪)
- intersectionArrayListChanges(ArrayList<Change>, ArrayList<Change>)
  returns ArrayList<Change>, defines Operator (∩)
- differenceArrayListChanges(ArrayList<Change>, ArrayList<Change>)
  returns ArrayList<Change>, defines Operator (\)

Working with Changes

- complementChange(Change) (Definition 2.7)
  returns Change
- compatibleChanges(ArrayList<Change>, ArrayList<Change>) (Definition 2.8)
  returns Boolean
- existsComplementChange(ArrayList<Change>, ArrayList<Change>)
  returns Boolean

Operations on KMTSs (Definition 3.2)

- unionKMTS(KMTS, KMTS, KMTS)
  returns List<KMTS>, defines Operator (⊔)
- intersectionKMTS(KMTS, KMTS, KMTS)
  returns List<KMTS>, defines Operator (⊓)
- differenceKMTS(KMTS, KMTS, KMTS)
  returns List<KMTS>, defines Operator (\)

Operations on KMTSs (Definition 3.3)

- contractionOperation(KMTS, KMTS, KMTS)
  returns List<KMTS>, defines Operator ($\sqcup^+$)

Operations on Sets of KMTSs (Definition 3.6)

- unionSetKMTS(KMTS, List<KMTS>, List<KMTS>)
  returns List<KMTS>, defines Operator (⊎)
- intersectionSetKMTS(KMTS, List<KMTS>, List<KMTS>)
  returns List<KMTS>, defines Operator (⊓)
- differenceSetKMTS(KMTS, List<KMTS>, List<KMTS>)
  returns List<KMTS>, defines Operator (⫽)

Operations on Partition Sets of KMTSs

- isFPS(List<KMTS>) (Definition 3.7)
  returns Boolean
- isPS(KMTS, List<KMTS>) (Definition 3.4)
  returns Boolean
- PS(KMTS, List<KMTS>) (Definition 3.4)
  returns List<KMTS>
- MPS(KMTS, List<KMTS>) (Definition 3.8)
  returns List<KMTS>

As an example how these operations look like in Java, in Listing 5.12 the implementation of the last one of the listed methods is shown. Its about the creation of a Maximum Partition Set (MPS) according to Definition 3.8.

```
/**
* Create Maximum Partition Set from a set of instances of M.
*
* @param M - KMTS
* @param inSet - set of instances of M
* @return Set of instances of M as result
*/
static List<KMTS> MPS(KMTS M, List<KMTS> inSet) {
  if (inSet == null || inSet.size() == 0) return null;
  if (inSet.size() <= 1) return inSet;

  List<KMTS> sk = new ArrayList<>(inSet);
  Boolean noMPS = true;

  while (noMPS) {
    noMPS = false;
    KMTS[] tempArray = sk.toArray(new KMTS[0]);

    for (int i = 0; i < tempArray.length - 1; i++) {
      List<KMTS> coResult = contractionOperation(M, tempArray[i], tempArray[i + 1]);

      if (coResult.size() == 1) {
        noMPS = true;
        sk.addAll(coResult);
        sk.remove(tempArray[i]);
        sk.remove(tempArray[i + 1]);
        break;
      }
    }
  }

  return sk;
}
```

Listing 5.12: Create Maximum Partition Set (Method *MPS* of class *SetTheory*)

5.6 Example

Here a short example is presented which, nevertheless, will show some of the above mentioned features. The KMTS of Figure 5.9 consists of two states and two transitions. It has one indetermination, namely the label of the state s_1.

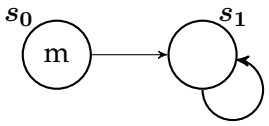

Figure 5.9: KMTS M to be model checked with CTL formula for initial state $s_0 \vdash \mathrm{EX}(\mathrm{m} \vee \mathrm{EX}(\neg\mathrm{m})) \vee \mathrm{A}(\neg\mathrm{m}\ \mathrm{U}\ \mathrm{m})$.

Figure 5.10 shows the program output of a model checking run for this KMTS against the formula named in Figure 5.9. Although, both sides of the disjunction within the EX operator result in *indefinite* (c_3, c_4), the disjunction itself evaluates *true* (c_2) and, therefore, also the enclosing EX operator (c_1). For this reason, the right side of the outer disjunction of the root configuration c_0 did not even have to be decomposed. Instead, a junction short cut to the finish could be taken, because with the on-the-fly optimising this part of the arena was not needed at all to see that the overall model checking result already was known as *true*. This is particularly remarkable, since otherwise the AU operator would have led to a complex fixed point calculation.

```
********** Coloured Arena:
configuration , parent , fploop , delta , colour , state ⊢ formula, info
c0  ,     ,     , {M}              , ⊤ , s0 ⊢ EX(m ∨ EX(¬m)) ∨ A(¬m U m) , shortcut
c1  , c0  ,     , {M}              , ⊤ , s0 ⊢ EX(m ∨ EX(¬m))
c2  , c1  ,     , {M}              , ⊤ , s1 ⊢ m ∨ EX(¬m)
c3  , c2  ,     , {M({P3(s1,m)})}  , I , s1 ⊢ m , deadend
c4  , c2  ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ EX(¬m)
c5  , c4  ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
****************************************
Configurations:  6 , Thereof Deadends:                  2
Junctions:       2 , Junction Short Cuts used           1
Fixpoint Loops:  0 , Calculations within Fixpoint Loops: 0
MPS generated:   3 , Contraction Operations:            1
Recursion Calls: 6 , Max Recursion Depth:               5
****************************************
********** Model Checking Result:
c0 ,    ,     , {M} , ⊤ , s0 ⊢ EX(m ∨ EX(¬m))) ∨ A(¬m U m) , shortcut
```

Figure 5.10: Coloured arena as listed by program output when model checking KMTS M of Figure 5.9.

Another very interesting point in this example is the mentioned fact that two *indefinite* operators of a disjunction are no guarantee that the disjunction itself will also result in *indefinite*. Having a look at the Maximum Contraction Functions (see Definition 3.10) of the two child configurations c_3 and c_4 in the delta column of Figure

5.10, you can see that the two concerned changes P3(s1,m) and P3(s1,¬m) are complement operations of each other. Thus, the union of the two instances of the KMTS M defined by M(P3(s1,m)) and M(P3(s1,¬m)) will result in the set {M} containing the whole KMTS itself, and this {M} is clearly the trigger for colouring the configuration c_2 *true* as can be seen in Definition 3.11.

5.7 Class Diagram

We are now ready to create a high level class diagram of our system like the one from Figure 5.11. This class diagram pictures only the most important classes with a selection of relations between them. No attributes or methods are shown in order to keep it easy for a better overview.

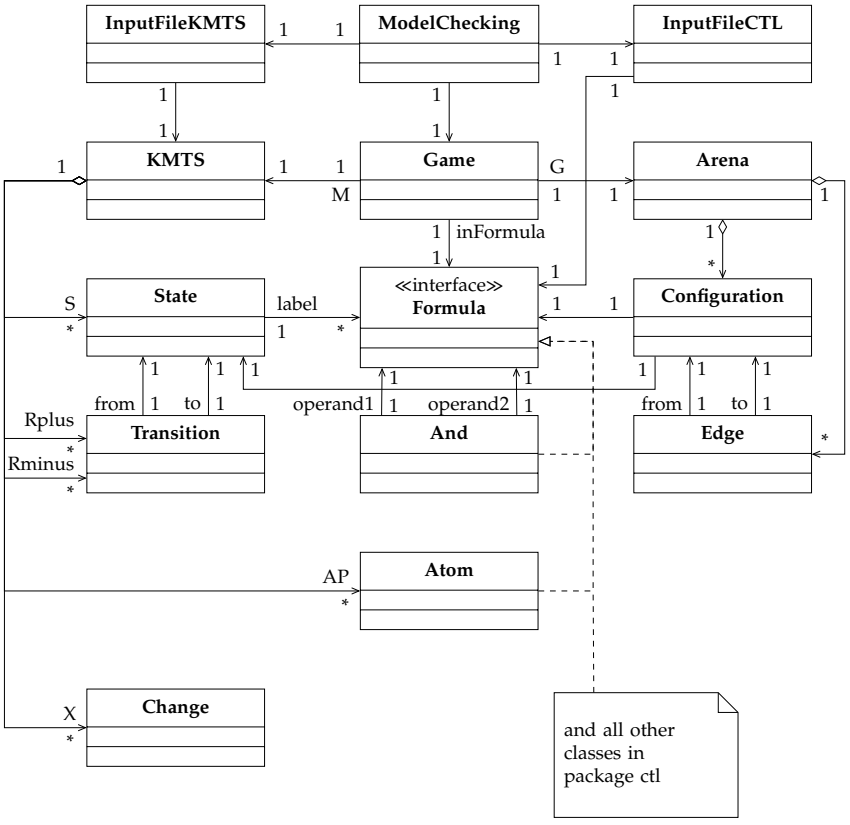

Figure 5.11: High level class diagram.

The central point of the system is the *Game* class surrounded by the input *KMTS*, the input *Formula* and the constructed *Arena*. Between the classes *And* and *Atom* which both implement the interface *Game*, you can imagine about 15 more classes like *Or, EX, AX* and so on also implementing the interface and representing CTL operators.

6 Experiment and Evaluation

Unfortunately, I could not find any necessary amount of Kripke structures, not to mention KMTSs, to use as test input for the implementation. This is because most of the models in model checking contests like the one at the Sorbonne University [ABC+19, LIP23], for example, are portrayed in Petri nets [Pet81]. A Petri net is a graph with two types of nodes called places and transitions. Arcs connect the nodes and a marking of a Petri net is a function from the set of places to the non negative integers [IM13]. Unfortunately, these data sets could not be used for the implementation tests, because Contraction Model Checking expects a KMTS as input. Of course, there are possibilities to transform Petri nets to transition systems like Kripke structures [FGHHO20, Oli21]. Islam and McCaull [IM13],for example, propose the following for representing a Petri net as a Kripke structure: the states are markings and there is a transition in the Kripke structure from state M to M' whenever a transition in the Petri net creates a marking M' from M. Yet, to my knowledge, there is no simple way to do this and a complicated transformation resulting in even more complicated Kripke structures would be far beyond the scope of this thesis. That is why I composed a set of test cases as comprehensive as possible to create a stable basis for this first draft of a Contraction Model Checking implementation.

Below you can see a selection of use cases that I consider interesting enough to take a closer look at. Input data files, for the cases listed here, are attached to the GitHub project in the \data folder and are also listed together with some log files in Appendix A. If some experimental runs show interesting or, above all, surprising results, you will find the details in the corresponding evaluation results section. Of course, the implemented Java program was tested against many more and more extensive cases.

6.1 Temporal Operator Until

We will simply use some easy combinations of CTL formulas containing an Until operator bound with different path operators. First, we check over a KMTS with no indetermination (Figure 6.1) and then we add one indetermination concerning a label for a second round (Figure 6.2).

6.1.1 Use Cases

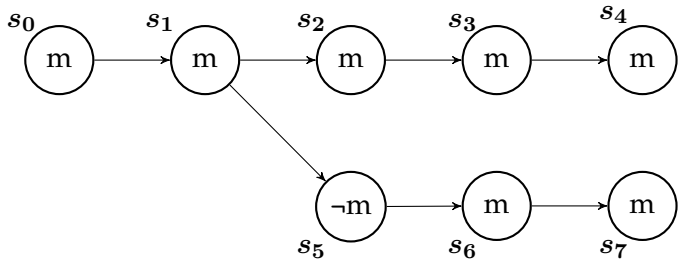

Figure 6.1: KMTS1: Use Case for Operator Until.

(a) Corresponding formula (CTL1): $s_0 \vdash \boldsymbol{E(m\,U\neg m)}$
Expected result: *true*

(b) Corresponding formula (CTL2): $s_0 \vdash \boldsymbol{E(\neg m\,U m)}$
Expected result: *true*

(c) Corresponding formula (CTL3): $s_0 \vdash \boldsymbol{A(m\,U\neg m)}$
Expected result: *false*

(d) Corresponding formula (CTL4): $s_0 \vdash \boldsymbol{A(\neg m\,U m)}$
Expected result: *true*

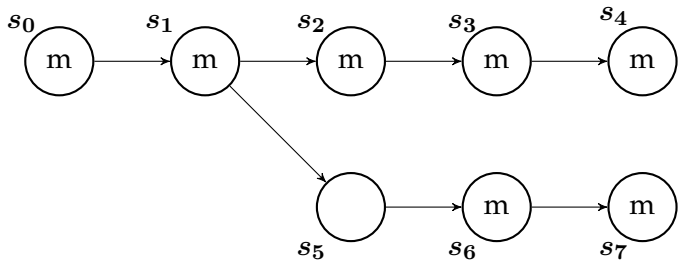

Figure 6.2: KMTS2: Use Case for Operator Until.

(e) Corresponding formula (CTL1): $s_0 \vdash \boldsymbol{E(m\,U\neg m)}$
Expected result: *indefinite* (Failure Witnesses around s_5)

(f) Corresponding formula (CTL2): $s_0 \vdash \boldsymbol{E(\neg m\,U m)}$
Expected result: *true*

(g) Corresponding formula (CTL3): $s_0 \vdash \boldsymbol{A(m\,U\neg m)}$
Expected result: *false*

(h) Corresponding formula (CTL4): $s_0 \vdash A(\neg m\, U m)$
Expected result: *true*

6.1.2 Evaluation

All runs with the exception of use case 6.1.1 (e) brought the expected results. For us an expected but, nevertheless, noteworthy result was the value *true* for model checking the use cases 6.1.1 (f) and (h), although not one label ¬m existed in the KMTS concerned. Reason is, s_0 with label m always holds for $s_0 \vdash E(\neg m\, U m)$. No value before reaching the target value is as good as the right value.

Not entirety expected was the result of use case 6.1.1 (e) (see Figure 6.3) where, of course, *indefinite* was expected. The Failure Witnesses coming from rules (4) and (5) were expected because of the indetermination in state s_5 of KMTS2 in Figure 6.2, namely the missing label. But to see the edge from configuration c_5 to configuration c_6 which originates in the transition between state s_0 and state s_1 marked as Failure Witness coming from rule (3) was a surprise. Although no indetermination was anywhere to find near the mentioned transition in the concerned KMTS, various tests showed that this result was correct for the definition of rule (3) as listed in Section 4. Furthermore, this definition captures exactly the definition of Guerra et al. in [GAW13]. We will discuss this topic later in this work. First of all, we just briefly show you the extended version of the rule (3) from the tests which, in our case, delivered the best results.

(3) A must edge, coming from a configuration of type EX coloured *indefinite*, to a child configuration coloured *indefinite*.

becomes

(3) A must edge, coming from a configuration of type EX coloured *indefinite*, to a child configuration $s_i \vdash l$ coloured *indefinite*.

The required form of the formula in the target configurations limits these to dead-ends.

```
********** Model Checking Result:
c0 ,    ,     , {M({P3(s5,¬m)})} , I , s0 ⊢ E(m U ¬m)) ,
********** Failure Witnesses:
from:state , to:state , rule , must
c5:s0      , c6:s1    , 3    , true
c31:s5     , c32:s5   , 5    , true
c33:s5     , c34:s5   , 4    , true
****************************************
```

Figure 6.3: Failure Witnesses for use case 6.1.1 (e) (KMTS2/CTL1).

6.2 Genuine May Transitions

Now we want to emphasize indeterminations caused by may transitions and their effects. We start with two very simple KMTSs and then move on to somewhat more complex structures even including transition cycles. A path within a KMTS that at some point meets a state that has already been visited, is called a transition cycle.

6.2.1 Use Cases

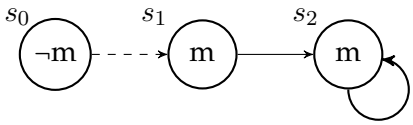

Figure 6.4: KMTS3: Use Case for Genuine May Transitions.

(a) Corresponding formula (CTL5): $s_0 \vdash m \vee EX(m)$
Expected result: *indefinite* (Failure Witnesses around s_1)

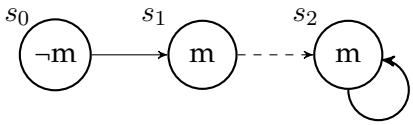

Figure 6.5: KMTS4: Use Case for Genuine May Transitions.

(b) Corresponding formula (CTL5): $s_0 \vdash m \vee EX(m)$
Expected result: *true*

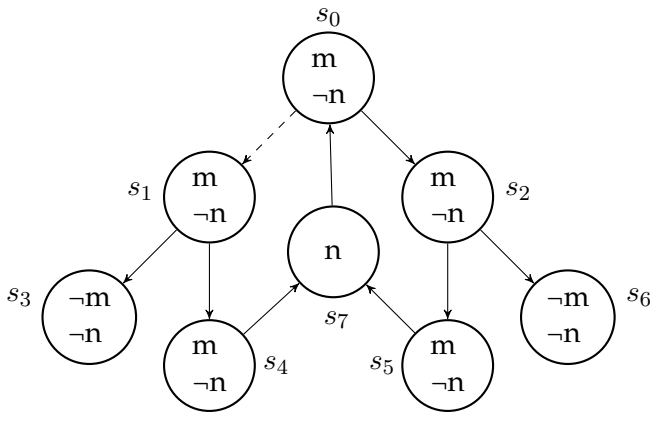

Figure 6.6: KMTS5: Use Case for Genuine May Transitions.

(c) Corresponding formula (CTL5): $s_0 \vdash m \vee EX(m)$
Expected result: *true*

(d) Corresponding formula (CTL6): $s_0 \vdash m \vee AX(m)$
Expected result: *true*

(e) Corresponding formula (CTL7): $s_0 \vdash E(m\,Un)$
Expected result: *true*

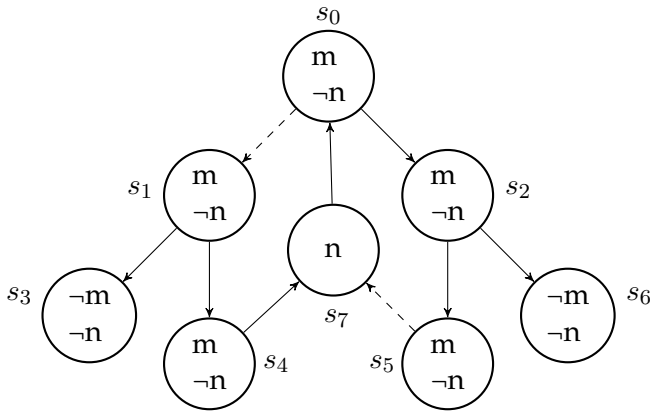

Figure 6.7: KMTS6: Use Case for Genuine May Transitions.

(f) Corresponding formula (CTL7): $s_0 \vdash E(m\,Un)$
Expected result: *indefinite* (Failure Witnesses around s_1 and s_7)

6.2.2 Evaluation

The experimental runs of use case 6.2.1 (a) to (d) brought the expected results. Figure 6.8 shows the responsible Failure Witness as listed for the coloured value *indefinite* of use case 6.2.1 (a).

```
********** Model Checking Result:
c0 ,    ,     , {M({P2(s0,s1)})} , I , s0 ⊢ m ∨ EX(m) ,
********** Failure Witnesses:
from:state , to:state , rule , must
c2:s0      , c3:s1    , 2    , false
****************************************
```

Figure 6.8: Failure Witnesses for use case 6.2.1 (a) (KMTS3/CTL5).

One could also emphasise the result *true* for use case 6.2.1 (d) although the transition from s_0 to s_1 is a genuine may transition and the formula $s_0 \vdash m \vee AX(m)$ requires

satisfying on all paths.The explanation for this phenomenon can be imagined even without calculation. A genuine may transition does exist or does not exist. If it exists, because with state s_1 labeled m, the connected child configuration will be coloured *true* and so all paths satisfy. If it does not exist, the only remaining path via state s_2 satisfies and so again all paths satisfy.

```
********** Model Checking Result:
c0 ,    ,     , {M({P2(s0,s1)}),M({P2(s5,s7),P1(s0,s1)})} , I , s0 ⊢ E(m U n) ,
********** Failure Witnesses:
from:state , to:state , rule , must
c5:s0      , c6:s1    , 2    , false
c37:s5     , c38:s7   , 2    , false
****************************************
```

Figure 6.9: Failure Witnesses for use case 6.2.1 (f) (KMTS6/CTL7).

KMTS5 of Figure 6.6 and KMTS6 of Figure 6.7 introduce the possibility of having labeled a state with more than one label, in this case the atomic propositions m and n. The only difference between the two KMTSs is the additional may transition from s_5 to s_7 in KMTS6. Both are model checked against the formula $s_0 \vdash E(\neg m\, Un)$. For KMTS5, as expected, the model checking results in *true*. The satisfying path $s_0 > s_2 > s_5 > s_7$ was easily found. As expected, for KMTS6 the model checking results in *indefinite*. Figure 6.9 shows the Failure Witnesses. This time there are two witnesses, because both places would, by changing the concerned transition to a must transition, turn the model checking result to true.

6.3 Applied Optimisations

We apply again some use cases from Section 6.2.1. But this time we do a second run for every case with switched off optimisation option to see the difference.

6.3.1 Use Cases

(a) Corresponding formula (CTL5): $s_0 \vdash m \vee EX(m)$ on KMTS3 of Figure 6.4
Expected result: *indefinite* (Failure Witnesses around s_1)

(b) Corresponding formula (CTL5): $s_0 \vdash m \vee EX(m)$ on KMTS4 of Figure 6.5
Expected result: *true*

(c) Corresponding formula (CTL7): $s_0 \vdash E(m\, Un)$ on KMTS6 of Figure 6.7
Expected result: *indefinite* (Failure Witnesses around s_1 and s_7)

6.3.2 Evaluation

As expected, the result of the first two use cases remained the same without any optimisations. Use case 6.3.1 (a) brought with optimisations a noticeable saving in generated configurations (33 configurations with optimisations to 48 configurations without optimisations). This is shown in Figure 6.10.

```
(i)  With optimisations:
****************************************
Configurations: 33  , Thereof Deadends: 11
Junctions: 12       , Junction Short Cuts used: 1
Fixpoint Loops: 1   , Calculations within Fixpoint Loops: 4
MPS generated: 11   , Contraction Operations: 0
Recursion Calls: 34 , Max Recursion Depth: 21
****************************************

(ii) Without optimisations:
****************************************
Configurations: 48  , Thereof Deadends: 16
Junctions: 17       , Junction Short Cuts used: 0
Fixpoint Loops: 2   , Calculations within Fixpoint Loops: 8
MPS generated: 17   , Contraction Operations: 0
Recursion Calls: 50 , Max Recursion Depth: 21
****************************************
```

Figure 6.10: Statistics for use case 6.3.1 (a) (KMTS3/CTL5).

As can be seen in Figure 6.11, for use case 6.3.1 (b), the savings were even more extreme. Here the ratio was 3 configurations created with optimisations to 48 configurations created without optimisations.

```
(i)  With optimisations:
****************************************
Configurations: 3   , Thereof Deadends: 1
Junctions: 1        , Junction Short Cuts used: 1
Fixpoint Loops: 0   , Calculations within Fixpoint Loops: 0
MPS generated: 0    , Contraction Operations: 0
Recursion Calls: 3  , Max Recursion Depth: 3
****************************************

(ii) Without optimisations:
****************************************
Configurations: 48  , Thereof Deadends: 16
Junctions: 17       , Junction Short Cuts used: 0
Fixpoint Loops: 2   , Calculations within Fixpoint Loops: 8
MPS generated: 9    , Contraction Operations: 0
Recursion Calls: 50 , Max Recursion Depth: 21
****************************************
```

Figure 6.11: Statistics for use case 6.3.1 (b) (KMTS4/CTL5).

For use case 6.3.1 (c) we could exaggerate that the savings with optimisations were infinitely large, because without the optimisation option the experimental runs ended

in an endless loop along the already mentioned transition cycles within KMTS6. The cause was, that a KMTS constellation with a transition cycle, also for Contraction Model Checking obviously allowed, during the construction of the arena according to the defined rules of the game led to the loop.

We added some code to stop the loop as soon as it occurs. The loop is recognised by the fact that a configuration is to be created with a state that already exists in the state path of the current configuration with absolutely the same CTL formula. This state must be different to the state of the current configuration because, otherwise, a fixed point loop, as usual, would have been triggered already. Nevertheless, the situation will be handled quite similarly to a fixed point loop. In contrast to a fixed point loop, we mark the loop end in the log with "VVV" in stead of "___". For KMTS6 of Figure 6.7 this would be the case for every configuration created with a state s_0, coming from a configuration with state s_7, because of the transition from s_7 to s_0 and trying to apply the exact same formula $s_0 \vdash E(m\,Un)$ as was applied at the beginning to the root configuration s_0.

We can now compare the results of the two runs and find a quite interesting difference between them. Although the actual model checking result of the runs with and without optimisations are indefinite and absolutely identical with regard to the calculated Maximum Contraction Function, the determined Failure Witnesses differ. Figure 6.12 shows one more Failure Witness for the not optimised runs. Thus, our concerns from Section 5.2.5 have become true. It seems that we lost a Failure Witness due to optimisations.

```
(i)  With optimisations:
********** Model Checking Result:
c0 ,    ,     , {M({P2(s0,s1)}),M({P2(s5,s7),P1(s0,s1)})} , I , s0 ⊢ E(m U n)
****************************************
********** Failure Witnesses:
from:state , to:state , rule , must
c5:s0      , c6:s1    , 2    , false
c37:s5     , c38:s7   , 2    , false
****************************************

(ii) Without optimisations:
********** Model Checking Result:
c0 ,    ,     , {M({P2(s0,s1)}),M({P2(s5,s7),P1(s0,s1)})} , I , s0 ⊢ E(m U n)
****************************************
********** Failure Witnesses:
from:state , to:state , rule , must
c5:s0      , c6:s1    , 2    , false
c27:s7     , c28:s7   , 4    , true
c41:s5     , c42:s7   , 2    , false
****************************************
```

Figure 6.12: Model Checking Result for use case 6.3.1 (c) (KMTS6/CTL7).

A closer look at the additional Failure Witness line shows that it is caused by game

rule (4). This rule, as listed in Section 4, is: an edge coming from a configuration of type $s_i \vdash l \wedge \varphi$ to a child configuration $s_i \vdash l$ coloured with *indefinite*. So, we talk about a conjunction with an atomic proposition as the first child. This first child has to be coloured *indefinite* to trigger the rule. Rule (5), the analogous rule for disjunctions, reads: an edge coming from a configuration of type $s_i \vdash l \vee \varphi$ to a child configuration $s_i \vdash l$ coloured with *indefinite* and the other child is coloured *indefinite* or *false*. The additional condition in rule (5) for the other child means that no Failure Witness would be triggered if the other child already decided the disjunction. Why should this not also apply to conjunctions and, thus, to rule (4)? The following change to game rule (4) brought good test results.

(4) An edge coming from a configuration of type $s_i \vdash l \wedge \varphi$ to a child configuration $s_i \vdash l$ coloured with *indefinite*.

becomes

(4) An edge coming from a configuration of type $s_i \vdash l \wedge \varphi$ to a child configuration $s_i \vdash l$ coloured with *indefinite* and the other child is coloured *indefinite* or *true*.

A conjunction is finally decided when one operand is *false*. The restriction for the other child limits the result to cases that have not already been decided by the other child, because only such cases could be Failure Witnesses.

An interesting side effect of the endless loop problem was the understanding, what size of an arena the Java program can sustain. The termination always occurred after some 3000 created configurations caused by a stack overflow exception because of round about 2000 not released recursive calls.

6.4 Literature Examples

To finalise our experiments, we now will have a look at two example cases from our main underlying literature by Ribeiro and Andrade [RA15] and Guerra et al. [GAW13]. These are particularly interesting for the consideration of fixed point processing, respectively the determination of Failure Witnesses. The related KMTSs are shown in Figure 6.13, respectively Figure 6.14. The third example is a slightly simplified version of the the well-known microwave oven scenario presented by Clarke et al. [CGP99] like it is used by Biswas and Deka [BD13] for online lectures. It will highlight the possibilities of the implementation concerning several meaningful labels, transition cycles as well as the functionality of the preprocessing. KMTS, CTL formulas and logfiles can be found in Appendix A.3.

6.4.1 Use Cases

(a) Corresponding formula (CTL_RA15): $s_0 \vdash EX(m) \vee E(m\,U\neg m)$
Expected result: *true*

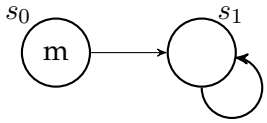

Figure 6.13: KMTS_RA15: Example by Ribeiro and Andrade.

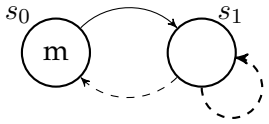

Figure 6.14: KMTS_GAW13: Example by Guerra et al..

(b) Corresponding formula (CTL_GAW13): $s_0 \vdash AX(AG(\neg m) \vee AF(m))$
Expected result: *indefinite* (Failure Witnesses around s_1)

(c) KMTS_Microwave: Microwave oven scenario (see Appendix A.3).
Corresponding formula (CTL_Microwave1): $s_0 \vdash AG(\neg(\neg Close \wedge Heat))$
No heating with open door - Expected result: *true*
Corresponding formula (CTL_Microwave2): $s_0 \vdash AG(Start \rightarrow AF(Heat))$
Starting at some point results in heating - Expected result: *false*

6.4.2 Evaluation

The details of use cases 6.4.1 (a) fixed point loop have already been discussed using Figure 5.1 in Section 5.1.2. Figure 5.5 in Section 5.2.1 shows the game arena of this use case with the fixed point loop marked between configuration $\boldsymbol{c_9}$ and configuration $\boldsymbol{c_{14}}$. The complete logfile including fixed point loop and arena can be found in Appendix A.4.

```
********** Failure Witnesses:
from:state , to:state , rule , must
c16:s1     , c11:s1   , 1    , false
c12:s1     , c13:s1   , 5    , false
c7:s1      , c8:s0    , 1    , false
c7:s1      , c2:s1    , 1    , false
c3:s1      , c4:s1    , 4    , false
****************************************
```

Figure 6.15: Failure Witnesses for use case 6.4.1 (b) (KMTS_GAW13/CTL_GAW13).

The KMTS in Figure 6.14 for use case 6.4.1 (b) contains three indeterminations. Two transitions and one state label. This large number, especially compared to the size of the KMTS, leads to quite an amount of Failure Witnesses listed in Figure 6.15. What is most interesting about this list of Failure Witnesses is the fact that two of them relate to edges resulting from fixed point loops. These are the edges from configuration $\boldsymbol{c_{16}}$ to configuration $\boldsymbol{c_{11}}$ in the first line and from configuration $\boldsymbol{c_7}$ to configuration $\boldsymbol{c_2}$ in the fourth line. Again the complete logfile can be found in Appendix A.4. For the microwave oven scenario, the evaluation is left to the interested reader. As mentioned, all necessary information can be found in Appendix A.3.

7 Discussion and Conclusion

In this thesis, an algorithm for model checking of structures with partial information templated as KMTSs was proposed. The algorithm is based on the theory of Contraction Model Checking (CMC) introduced by Ribeiro and Andrade [RA15] and extends this theory by pinpointing any conflicts by Failure Witnesses within the model that may have arisen as a result of the partial information. The Failure Witnesses logic was taken over by Guerra et al. [GAW13]. The complete algorithms were also implemented as a Java program.

The first part of the thesis contains formal definitions of CTL, Kripke structures and KMTSs including the associated semantics. Then the definitions, necessary for the implementation of CMC and Failure Witnesses, are presented. Based on these preliminaries, an implementation approach for the algorithms was developed and supplemented by a description of the most important parts of the Java implementation. The topic was rounded off by a series of experimental test runs of the Java program and an evaluation and presentation of the most important results and findings of these runs.

CMC is a game based approach that interprets a KMTS as a set of Kripke structures, called the expansion of the KMTS. This was first introduced by Guerra et al. [GAW13] but, to my knowledge, CMC is the first method that directly checks the KMTS and not the Kripke structure of the expansion. The mentioned experimental runs showed that this new method could be successfully implemented, not least, because of the absolutely complete theoretical CMC definitions of Ribeiro and Andrade [RA15]. The fact that CMC does not require a CTL formula first to be formally translated into a μ-calculus formula [BW18], as Guerra et al. [GAW13] and its pioneers [GLLS07, SG12] proposed, turned out to be very positive. CMC can handle the pure CTL formulas because of the game rules formulated in CTL. Nevertheless, the incoming CTL formulas have to be preprocessed. Actually, CMC can only handle the temporal operators X, U and R, always bound by one of the path operators E respective A, but that does no harm. The translations for F, G and $\rightarrow$, proposed, in Section 2.1.1, work without a problem for almost all kinds of CTL formulas. It is obvious that the fewer operators that have to be taken into account for coding, the easier and shorter is the development of the Java code. Considering also that U and R only differ in details, but not in the necessary program logic, this selection by Ribeiro and Andrade [RA15] cannot be overestimated, although it seems quite idiosyncratic compared to other model checker variants which mostly use X, G and U for a minimal representation of CTL. Of course, the allowed operators could be restricted even further, because not every combination of temporal and path operators, sometimes called connectives, is necessary for a minimal representation of CTL. Martin [Mar01] shows that, for example, the connectives EX, AU and EU would be sufficient for this purpose. That means for CMC that only the three game rules con-

cerning these connectives would be needed, of course, in addition to the rules for the logical operators And and Or, because all other connectives can be translated to these. However, his extreme solution has one disadvantage. The resulting CTL formulas of this translations would quickly become very confusing and what is worth very large. This would lead, via decomposing, to very large arenas and that is what we actually wanted to prevent. From this point of view, the six in the game rules of CMC used connectives EX, AX, EU, AU, ER and AR seem to be a very good compromise.

As shown in Section 6, some of the *indefinite* resulting use cases pointed out too many Failure Witnesses. At least, the existence of these Failure Witnesses could not be explained. To reduce the number to a degree that we considered correct, it was necessary to slightly adapt two of the Failure Witness rules from Section 4. The adaptions are listed here again. Of course, under different circumstances in a different environment, the original rule variations may have been correct and the only possible solution. In any case, with the stronger restrictions of the new versions, the results for our experimental runs came out much better.

(3) A must edge, coming from a configuration of type EX coloured *indefinite*, to a child configuration $s_i \vdash l$ coloured *indefinite*.

Effect: the required form of the formula in the target configurations limits these to deadends.

(4) An edge coming from a configuration of type $s_i \vdash l \wedge \varphi$ to a child configuration $s_i \vdash l$ coloured with *indefinite* and the other child is coloured *indefinite* or *true*.

Effect: a conjunction is finally decided when one operand is *false*. The restriction for the other child limits the result to cases that have not already been decided by the other child, because only such cases could be Failure Witnesses.

The experimental runs carried out could all be measured in thousandth of a second. Not only because of the lack of extensive test cases, but mostly because of the necessity of testing as many different constellations as possible in a very short time available. Nevertheless, a good amount of optimisation was already implemented right from the beginning. Specifically, the on-the-fly optimisation was created successfully and completely. Of course, the measurable success always depends on the specific circumstances. Sometimes, depending on the involved KMTS and CTL formula, only a few configurations of an arena are saved during construction, other times a large part of the arena must not get created at all to receive a result. In any case, you can never notice a worsening through the optimisation. Also the original fears that Failure Witnesses could be lost as a result of optimisation have not materialised.

Another problem that only came up during the test runs was the fact that transition cycles in the KMTS could lead to endless loops when building the arena and therefore to infinite arenas. That is because the arena is build top down from each configuration to their children. Thus, if it happens that the state of a new configuration has already been on the state path covered to this point of the game, it will start all over again from here. Considering the time available, we decided for a maybe quick and dirty solution. If the constellation is recognised, the loop will be broken by handling the situation as it would be a normal fixed point loop. Further investigation in the future is necessary to see whether information for colouring the arena is lost as a result of this procedure.

The unintentional endless loops also had their good side. They helped to determine fairly accurate how many recursive calls the Java program can handle since that depends on the exact size, number and type of classes that have to be stored on the stack at every recursive call. This number is not easy to determine other than by experimental challenge. The stack overflow exception occurred regularly after about 2000 not yet returned recursive calls. That means, the arena can have a maximum depth of about 2000 configurations which is quite a lot. The cycling runs created for this depth regularly over 3000 configurations. This means, at least for the current development stage of the Java program, that there is no need at all to worry about the recursive calls.

8 Future Work

Due to the short development time, there are still a few open issues in connection with our implementation. Some of them are already commented in the code as *TODOs* like, for example, checking the input data, error output at all critical code passages or improvements in the mentioned transition cycles, in particular allow more than one cycle to start at one configuration. Yet, the current level of development of the algorithms and the Java program should be a good starting point for future work. Here one topic could be testing with large amounts of data, i.e. with large models and challenging formulas, and any eventually necessary adjustments to the program. Ideally, these big data tests should be based on real-life use cases. As we have already found, it is difficult, if not impossible, to obtain such data in the form of KMTS models. What is widely used and what would be easily available are test cases in the form of Petri nets, for example in connection with the already mentioned yearly French model checking contest[LIP23]. A solution could be to search for an applicable existing algorithm to transform Petri nets to a KMTS or, at least, to a Kripke structure as an intermediate step. Should this search be not successful, the algorithm can also be developed from scratch. As earlier mentioned, comparable approaches can definitely be found in the literature [IM13, FGHHO20, Oli21]. They would have to be adapted to the specific requirements.

The most important point for future research on the subject could be a scientifically sound statement at a level of significance to be determined about the reliability of the CMC results. For this, the results of the CMC Java implementation, developed through this thesis, should be compared to a model checking of the individual CTL models that represent a KMTS according to Guerra et al. [GAW13]. Of course, here it would be very important that an existing, available and reliable model checker, and not an in-house development or even this Java implementation itself, is used for the model checking of the individual Kripke structures. The effort involved in selecting such a model checker should not be underestimated, but this seems to be the only way to achieve really verifiable statements when comparing the results. Maybe the survey for model checkers by Raj and Suryaprasad [RJ16] could be a helpful starting point. Whether something like this could and should be approached in conjunction with the already mentioned desirable translation of Petri nets to usable Kripke structures or even KMTSs, would have to be carefully determined beforehand.

With the developments of this thesis, only conflicts responsible for the model checking outcome *indefinite* can be pointed out. Would it not be desirable to also pinpoint the reasons for an outcome of *false*? Of course, new Failure Witnesses rules would have to be established first. Apart from that, it would be a small program change with, in my opinion, a major impact on the usability of the application.

Furthermore, so far conflicts can only be pointed out but not revised. One more in-

teresting aspect for the future, based on the implementation of the Failure Witnesses logic, could be the automatic revision of the KMTS models proposed by Guerra et al. [GAW13]. What would that mean for some of the real world use cases presented in the introduction? Including such a revision directly in the model checking implementation could have positive and negative aspects. Maybe the system model in form of a KMTS represents the surroundings of an agent and the agent gets contradictory information from different sources. For example, a sensor in a self-driving car detects an obstacle, other sensors do not report it. Can an automatic revision take place and what should be the result? Revising this conflict would solve the problem in the software of the car. No matter whether the solution was to remove the obstacle from the agents, respective the cars, belief or not, if it was the wrong decision it could lead to a car accident. Either through a collision with the obstacle or through an emergency braking that other road users could not have foreseen. Thus, in this case, pinpointing the problem is still important, but a simple revision may not be a solution or at least has to be handled with care. On the other hand, imagine the KMTS represents the target system model to be created in a system development process. Would it not be nice to be able to check the interim results of this model against the pre-established system requirements and not only get back a right, a wrong or a do not know, but an automatically to the different requirements adapted new system model?

The sense and benefit of an automatic revision, therefore, depends heavily on the actual application. But there are certainly applications that could derive a great benefit from such an integrated revision. Thus, expanding this work in the future to include an automatic revision of the system model based on the conflicts found should definitely be considered.

References

[ABC+19] Elvio Amparore, Bernard Berthomieu, Gianfranco Ciardo, Silvano Dal Zilio, Francesco Gallà, Lom Messan Hillah, Francis Hulin-Hubard, Peter Gjøl Jensen, Loïg Jezequel, Fabrice Kordon, Didier Le Botlan, Torsten Liebke, Jeroen Meijer, Andrew Miner, Emmanuel Paviot-Adet, Jiří Srba, Yann Thierry-Mieg, Tom van Dijk, and Karsten Wolf. Presentation of the 9th Edition of the Model Checking Contest. In Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 50–68, Cham, 2019. Springer International Publishing.

[ABM19] Erman Acar, Massimo Benerecetti, and Fabio Mogavero. Satisfiability in Strategy Logic Can Be Easier than Model Checking. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33:2638–2645, 07 2019.

[AY01] Rajeev Alur and Mihalis Yannakakis. Model Checking of Hierarchical State Machines. In *ACM Transactions on Programming Languages and Systems*, volume 23, pages 273–303, [online] Available: https://doi.org/10.1145/503502.503503, accessed 02.05.2023, 2001.

[BD13] Santosh Biswas and Jatindra Kumar Deka. Model Checking with Fairness. In *Design Verification and Test of Digital VLSI Designs*, volume 4, Indian Institute of Technology, Guwahati, [online] Available: http://www.nitttrc.edu.in/nptel/courses/video/106103116/lec22.pdf, accessed 02.05.2023, 2013. nptel.

[BG99] Glenn Bruns and Patrice Godefroid. Model Checking Partial State Spaces with 3-Valued Temporal Logic. *CAV 1999*, pages 247–287, 1999.

[BGR01] Michael Benedikt, Patrice Godefroid, and Thomas Reps. Model Checking of Unrestricted Hierarchical State Machines. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 652–666, Berlin, Heidelberg, 2001. Springer.

[BMPAFV04] Ana Belén Barragáns Martínez, José J. Pazos Arias, and Ana Fernández Vilas. On Measuring Levels of Inconsistency in Multi-Perspective Requirements Specification. In *1st Conference on the Principles of Software Engineering (PRISE)*, pages 21–30, 2004.

[Bre22] Jan Bretschneider. modelcheck 1.0.0, 2022. Url: https://github.com/jbretsch/modelcheck, accessed 02.05.2023 .

[Bun15] Daniel Bundala. Satisfiability Modulo Theories, Part I. *Computer-Aided Verification, Berkley EECS*, pages 1–5, 2015. Url:

https://people.eecs.berkeley.edu/~sseshia/219c/EECS219C-ScribedNotes-Spr2015.pdf, accessed 02.05.2023 .

[BW18] Julian Bradfield and Igor Walukiewicz. The mu-calculus and Model Checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 871–919. Springer International Publishing, Cham, 2018.

[CDEG04] Marsha Chechik, Benet Devereux, Steve Easterbrook, and Arie Gurfinkel. Multi-Valued Symbolic Model-Checking. *ACM Transactions on Software Engineering and Methodology*, 12:1–38, 2004.

[CFR22] Remy Cherkhaoui, Quentin Felten, and Daniil Rosso. Formal Modeling Mini Project, Paris 2022. Url: https://github.com/eldondanonino/Formal-Modeling-Mini-Project, accessed 02.05.2023 .

[CG18] Sagar Chaki and Arie Gurfinkel. BDD-Based Symbolic Model Checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 219–245. Springer International Publishing, Cham, 2018.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, 1999.

[CHV18] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to Model Checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 1–26. Springer International Publishing, Cham, 2018.

[Eme91] Allen E. Emerson. Temporal And Modal Logic. In *Handbook of Theoretical Computer Science: Formal Models and Semantics*, pages 995–1072. MIT Press, Cambridge, 1991.

[FG08] Alessandro Fantechi and Stefania Gnesi. Formal Modeling for Product Families Engineering. In *12th International Software Product Line Conference*, pages 193–202, 2008.

[FGHHO20] Bernd Finkbeiner, Manuel Gieseking, Jesko Hecking-Harbusch, and Ernst-Rüdiger Olderog. Model Checking Branching Properties on Petri Nets with Transits. In Dang Van Hung and Oleg Sokolsky, editors, *Automated Technology for Verification and Analysis*, pages 394–410, Cham, 2020. Springer International Publishing.

[Fin12] Jan Finis. Incremental Model Checking of Recursive Kripke Structures. Master's thesis, University of Augsburg, Institute of Computer Science, Augsburg, Germany, 2012. Url:

https://db.in.tum.de/people/sites/finis/papers/MT Jan Finis - Incremental MCing of RKSs.pdf, accessed 02.05.2023 .

[GAW13] Paulo T. Guerra, Aline Andreade, and Renata Wassermann. Toward the Revision of CTL Models through Kripke Modal Transition Systems. In Juliano Iyoda and Leonardo de Moura, editors, *Formal Methods: Foundations and Applications - 16th Brazilian Symposium*, pages 115–130, Heidelberg, 2013. Springer.

[GLLS07] Orna Grumberg, Martin Lange, Martin Leucker, and Sharon Shoham. When Not Losing Is Better than Winning: Abstraction and Refinement for the Full μ-Calculus. *Information and Computation*, 205:1130–1148, 2007.

[GR22] Valentin Goranko and Antje Rumberg. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Summer 2022 edition, 2022.

[Gru10] Orna Grumberg. 2-Valued and 3-Valued Abstraction-Refinement in Model Checking. *Logics and Languages for Reliability and Security*, 25:105–128, 2010.

[HJS01] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal Transition Systems: A Foundation for Three-Valued Program Analysis. In David Sands, editor, *Programming Languages and Systems*, pages 155–169, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[HKV96] Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. A Space-Efficient On-the-fly Algorithm for Real-Time Model Checking. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 514–529, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[HR04] Michael Huth and Mark Ryan. *Logic in Computer Science*. Cambridge University Press, Cambridge, 2004.

[IM13] Zahidul Islam and Wendy Maccaull. A One-Pass Tableau-Based Workflow Verification Framework. In Pascal Fontaine, Renate A. Schmidt, and Stephan Schulz, editors, *PAAR-2012. Third Workshop on Practical Aspects of Automated Reasoning*, volume 21 of *EPiC Series in Computing*, pages 58–71. EasyChair, 2013.

[Kri63] Saul A. Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[LIP23] LIP6. Model Checking Contest 2023 - Rules, 2023. Url: https://mcc.lip6.fr/2023/pdf/rules.pdf, accessed 02.05.2023 .

[LLM14] Diego Latella, Michele Loreti, and Mieke Massink. On-the-fly Probabilistic Model Checking. *EPTCS 166*, pages 45–95, 2014.

[Mar01] Alan Martin. Adequate Sets of Temporal Connectives in CTL. *Electr. Notes Theor. Comput. Sci.*, 52:21–31, 01 2001.

[NC13] Boldizsár Németh and Zolgán Csörnyei. Stackless Programming in Miller. *Acta Universitatis Sapientiae. Informatica*, 5:167–183, 08 2013.

[Oli21] Olof Olivecrona. Efficient Algorithms for Temporal Logic Verification. Master's thesis, Chalmers University of Technology, Department of Electrical Engineering, Gothenburg, Sweden, 2021. Url: https://odr.chalmers.se/server/api/core/bitstreams/9a2219f0-8338-4ee6-973d-9032dc6c602b/content, accessed 02.05.2023 .

[Ora23] Oracle. JDK Releases, 2023. Url: https://www.java.com/releases/, accessed 02.05.2023 .

[Par20] David Parsons. *Foundational Java: Key Elements and Practical Programming*. Springer International Publishing, second edition, 2020.

[Pet81] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, 1981.

[RA15] Jandson S. Ribeiro and Aline Andrade. A 3-Valued Contraction Model Checking Game. In Michael Butler, Sylvain Conchon, and Fatiha Zaïdi, editors, *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods*, pages 85–99, Heidelberg, 2015. Springer.

[Ren23] Burkhardt Renz. Logik und formale Methoden. Lecture script, Technische Hochschule Mittelhessen, 2022. Url: https://esb-dev.github.io/mat/lfm.pdf, accessed 02.05.2023.

[RJ16] Shubha Raj and Suryaprasad Jayadevappa. Model Checkers - Tools and Languages for System Design - A Survey. In *Fifth International Conference on Advanced Information Technologies and Applications*, pages 39–51, 11 2016.

[Sch02] Philippe Schnoebelen. The Complexity of Temporal Logic Model Checking. In *Advances in Modal Logic*, volume 4, pages 1–44. World Scientific Publishing, 2002.

[SG03] Sharon Shoham and Orna Grumberg. A Game-Based Framework for CTL Counterexamples and 3-Valued Abstraction-Refinement. *CAV 2003*, page 275–287, 2003.

[SG12] Sharon Shoham and Orna Grumberg. Multi-valued Model Checking Games. *Journal of Computer and System Sciences*, 78:414–429, 2012.

[SK05] Chaiwat Sathawornwichit and Takuya Katayama. A Parametric Model Checking Approach for Real-Time Systems Design. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 584–595, 2005.

[SLW23] Inderjeet Singh, Joel Leitch, and Jesse Wilson. Gson User Guide, 2023. Url: https://github.com/google/gson/blob/master/ UserGuide.md, accessed 02.05.2023 .

[Sti01] Colin Stirling. *Modal and Temporal Properties of Processes*. Springer Verlag, 2001.

[TIO23] TIOBE Software BV. TIOBE Index for March, Eindhoven 2023. Url: https://www.tiobe.com/tiobe-index/, accessed 02.05.2023 .

A Appendix

A.1 CTL Files

This is an example with content provided by Ribeiro and Andrade [RA15].

Listing A.1: CTL_RA15.json

```
/Input CTL Formula (Example by Ribeiro and Andrade [RA15])
{
/CTL Formula)
"CTL":             "OR(EX('m'),EU('m',NOT('m')))"
}
```

These are the experimental CTL Json files used in Section 6 (Experiment and Evaluation).

Listing A.2: CTL1.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":             "EU('m',NOT('m'))"
}
```

Listing A.3: CTL2.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":             "EU(NOT('m'),'m')"
}
```

Listing A.4: CTL3.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":             "AU('m',NOT('m'))"
}
```

Listing A.5: CTL4.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":             "AU(NOT('m'),'m')"
}
```

Listing A.6: CTL5.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":             "OR('m',EX('m'))"
}
```

Listing A.7: CTL6.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":            "OR('m',AX('m'))"
}
```

Listing A.8: CTL7.json

```
/Input CTL Formula
{
/CTL Formula
"CTL":            "EU('m','n')"
}
```

A.2 KMTS Files

This is an example with content provided by Ribeiro and Andrade [RA15].

Listing A.9: KMTS_RA15.json

```
/Input KMTS (Example by Ribeiro and Andrade [RA15])
{
/Atomic Propositions
"AP":       [{"label":"m"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"1"}]},
                  {"name":"s1","label":[]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"1"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"0"},
                  {"transition":"1"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"}]
}
```

These are the eperimental KMTS Json files used in Section 6 (Experiment and Evaluation).

Listing A.10: KMTS1.json

```
/Input KMTS
{
/Atomic Propositions
"AP":       [{"label":"m"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"1"}]},
                  {"name":"s1","label":[{"value":"1"}]},
                  {"name":"s2","label":[{"value":"1"}]},
                  {"name":"s3","label":[{"value":"1"}]},
                  {"name":"s4","label":[{"value":"1"}]},
                  {"name":"s5","label":[{"value":"-1"}]},
                  {"name":"s6","label":[{"value":"1"}]},
                  {"name":"s7","label":[{"value":"1"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"2"},
                  {"stateFrom":"2","stateTo":"3"},
                  {"stateFrom":"3","stateTo":"4"},
                  {"stateFrom":"1","stateTo":"5"},
                  {"stateFrom":"5","stateTo":"6"},
                  {"stateFrom":"6","stateTo":"7"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"}]
}
```

Listing A.11: KMTS2.json

```
/Input KMTS
{
/Atomic Propositions
"AP":       [{"label":"m"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"1"}]},
                  {"name":"s1","label":[{"value":"1"}]},
                  {"name":"s2","label":[{"value":"1"}]},
                  {"name":"s3","label":[{"value":"1"}]},
                  {"name":"s4","label":[{"value":"1"}]},
                  {"name":"s5","label":[]},
                  {"name":"s6","label":[{"value":"1"}]},
                  {"name":"s7","label":[{"value":"1"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"2"},
                  {"stateFrom":"2","stateTo":"3"},
                  {"stateFrom":"3","stateTo":"4"},
                  {"stateFrom":"1","stateTo":"5"},
                  {"stateFrom":"5","stateTo":"6"},
                  {"stateFrom":"6","stateTo":"7"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"}]
}
```

Listing A.12: KMTS3.json

```
/Input KMTS
{
/Atomic Propositions
"AP":       [{"label":"m"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"-1"}]},
                  {"name":"s1","label":[{"value":"1"}]},
                  {"name":"s2","label":[{"value":"1"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"2"},
                  {"stateFrom":"2","stateTo":"2"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"1"},
                  {"transition":"2"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"}]
}
```

Listing A.13: KMTS4.json

```
/Input KMTS
{
/Atomic Propositions
"AP":       [{"label":"m"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"-1"}]},
                  {"name":"s1","label":[{"value":"1"}]},
                  {"name":"s2","label":[{"value":"1"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"2"},
                  {"stateFrom":"2","stateTo":"2"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"0"},
                  {"transition":"2"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"}]
}
```

Listing A.14: KMTS5.json

```
/Input KMTS
{
/Atomic Propositions
"AP":       [{"label":"m"},{"label":"n"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s1","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s2","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s3","label":[{"value":"-1"},{"value":"-2"}]},
                  {"name":"s4","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s5","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s6","label":[{"value":"-1"},{"value":"-2"}]},
                  {"name":"s7","label":[{"value":"2"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"0","stateTo":"2"},
                  {"stateFrom":"1","stateTo":"3"},
                  {"stateFrom":"1","stateTo":"4"},
                  {"stateFrom":"4","stateTo":"7"},
                  {"stateFrom":"2","stateTo":"5"},
                  {"stateFrom":"2","stateTo":"6"},
                  {"stateFrom":"5","stateTo":"7"},
                  {"stateFrom":"7","stateTo":"0"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"},
                  {"transition":"7"},
                  {"transition":"8"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"},
                  {"transition":"7"},
                  {"transition":"8"}]
}
```

Listing A.15: KMTS6.json

```
/Input KMTS
{
/Atomic Propositions
"AP":       [{"label":"m"},{"label":"n"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":        [{"name":"s0","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s1","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s2","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s3","label":[{"value":"-1"},{"value":"-2"}]},
                  {"name":"s4","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s5","label":[{"value":"1"},{"value":"-2"}]},
                  {"name":"s6","label":[{"value":"-1"},{"value":"-2"}]},
                  {"name":"s7","label":[{"value":"2"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",

/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"0","stateTo":"2"},
                  {"stateFrom":"1","stateTo":"3"},
                  {"stateFrom":"1","stateTo":"4"},
                  {"stateFrom":"4","stateTo":"7"},
                  {"stateFrom":"2","stateTo":"5"},
                  {"stateFrom":"2","stateTo":"6"},
                  {"stateFrom":"5","stateTo":"7"},
                  {"stateFrom":"7","stateTo":"0"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"},
                  {"transition":"8"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"},
                  {"transition":"7"},
                  {"transition":"8"}]
}
```

A.3 The Microwave Oven Scenario

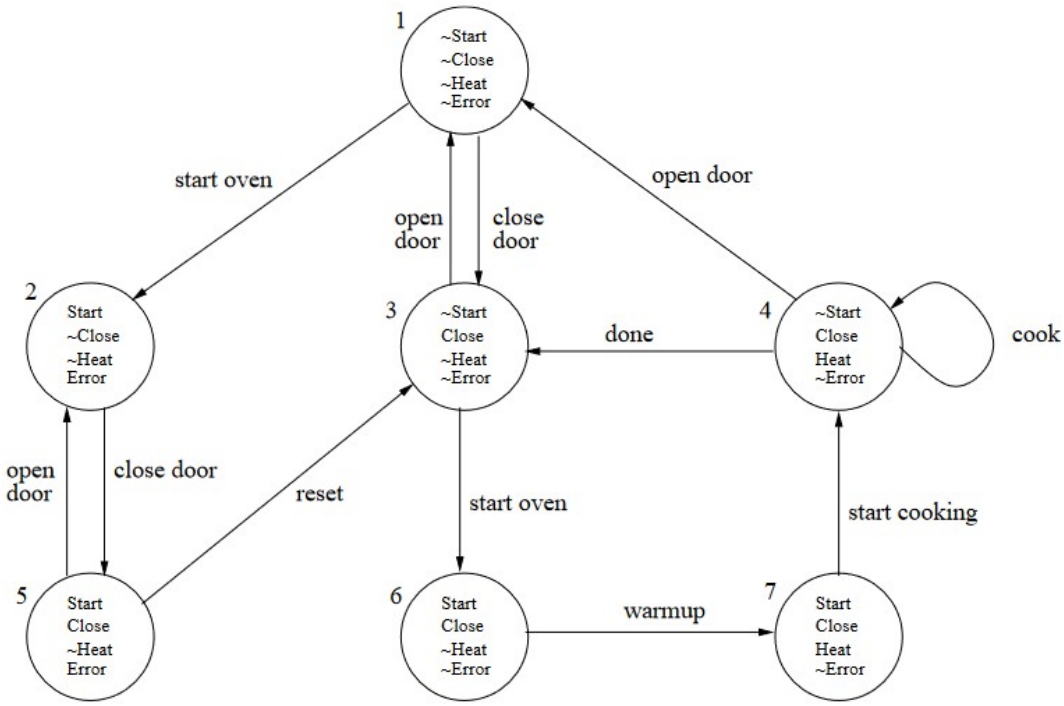

Figure A.1: KMTS_Microwave: The Microwave Oven Scenario.

Listing A.16: KMTS_Microwave.json

```
/Input KMTS (Example by )
{
/Atomic Propositions
"AP": [{"label":"Start"},{"label":"Close"},{"label":"Heat"},{"label":"Error"}],

/States (Labels defined by "value" as index of AP starting at 1)
/       ("not" marked as negativ value)
"S":[
{"name":"s0","label":[{"value":"-1"},{"value":"-2"},{"value":"-3"},{"value":"-4"}]},
{"name":"s1","label":[{"value":"-1"},{"value":"2"},{"value":"-3"},{"value":"-4"}]},
{"name":"s2","label":[{"value":"-1"},{"value":"2"},{"value":"3"},{"value":"-4"}]},
{"name":"s3","label":[{"value":"1"},{"value":"2"},{"value":"-3"},{"value":"-4"}]},
{"name":"s4","label":[{"value":"1"},{"value":"2"},{"value":"3"},{"value":"-4"}]},
{"name":"s5","label":[{"value":"1"},{"value":"-2"},{"value":"-3"},{"value":"4"}]},
{"name":"s6","label":[{"value":"1"},{"value":"2"},{"value":"-3"},{"value":"4"}]}],

/Initial State (defined by index of S starting at 0)
"SI":       "0",
```

```
/Transitions (States defined by index of S starting at 0)
"T":        [{"stateFrom":"0","stateTo":"1"},
                  {"stateFrom":"2","stateTo":"0"},
                  {"stateFrom":"2","stateTo":"1"},
                  {"stateFrom":"1","stateTo":"0"},
                  {"stateFrom":"1","stateTo":"3"},
                  {"stateFrom":"2","stateTo":"2"},
                  {"stateFrom":"3","stateTo":"4"},
                  {"stateFrom":"4","stateTo":"2"},
                  {"stateFrom":"0","stateTo":"5"},
                  {"stateFrom":"5","stateTo":"6"},
                  {"stateFrom":"6","stateTo":"5"},
                  {"stateFrom":"6","stateTo":"1"}],

/Must Transitions (Transitions defined by index of T starting at 0)
"Rplus":    [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"},
                  {"transition":"7"},
                  {"transition":"8"},
                  {"transition":"9"},
                  {"transition":"10"},
                  {"transition":"11"}],

/May Transitions (Transitions defined by index of T starting at 0)
"Rminus":   [{"transition":"0"},
                  {"transition":"1"},
                  {"transition":"2"},
                  {"transition":"3"},
                  {"transition":"4"},
                  {"transition":"5"},
                  {"transition":"6"},
                  {"transition":"7"},
                  {"transition":"8"},
                  {"transition":"9"},
                  {"transition":"10"},
                  {"transition":"11"}]
}
```

Listing A.17: CTL_Microwave1.json

```
/Input CTL Formula (Microwave oven: no heating with open door)
{
/CTL Formula)
"CTL":            "AG(NOT(AND(NOT('Close'),'Heat')))"
}
```

Listing A.18: CTL_Microwave2.json

```
/Input CTL Formula (Microwave oven: starting at some point results in heating)
{
/CTL Formula)
"CTL":            "AG(IMP('Start',AF('Heat')))"
}
```

Listing A.19: Logfile Microwave Oven Scenario 1

```
********** Runcontrol:
kmtsfile: KMTS_Microwave.json
ctlfile : CTL_Microwave1.json
opti    : 1
rc      : 1
rt      : 0
prepro  : 1
fploop  : 0
arena   : 1
stats   : 1
****************************************
Input CTL formula: AG(¬(¬Close ∧ Heat))
********** Input Data:
Input initial state: s0
Input KMTS.AP: Start, Close, Heat, Error,
Contraction Model Checking Form: A(⊥ R ¬(¬Close ∧ Heat))
Negation Normal Form: A(⊥ R (Close ∨ ¬Heat))
****************************************
********** Coloured Arena:
configuration, parent, fploop, delta, colour, state ⊢ formula, info
c0 ,    , c14 , {M} , T , s0 ⊢ A(⊥ R (Close ∨ ¬Heat))
c1 , c0 ,     , {M} , T , s0 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))
    ))))
c2 , c1 ,     , {M} , T , s0 ⊢ (Close ∨ ¬Heat)
c3 , c2 ,     , ∅ , F , s0 ⊢ Close , deadend
c4 , c2 ,     , {M} , T , s0 ⊢ ¬Heat , deadend
c5 , c1 ,     , {M} , T , s0 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c6 , c5 ,     , ∅ , F , s0 ⊢ ⊥ , deadend
c7 , c5 ,     , {M} , T , s0 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c8 , c7 , c35 , {M} , T , s1 ⊢ A(⊥ R (Close ∨ ¬Heat))
c9 , c8 ,     , {M} , T , s1 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))
    ))))
c10 , c9 ,     , {M} , T , s1 ⊢ (Close ∨ ¬Heat) , shortcut
c11 , c10 ,     , {M} , T , s1 ⊢ Close , deadend
c12 , c9 ,     , {M} , T , s1 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c13 , c12 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c14 , c12 , VVV , {M} , T , s1 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c15 , c14 ,     , {M} , T , s3 ⊢ A(⊥ R (Close ∨ ¬Heat))
c16 , c15 ,     , {M} , T , s3 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c17 , c16 ,     , {M} , T , s3 ⊢ (Close ∨ ¬Heat) , shortcut
c18 , c17 ,     , {M} , T , s3 ⊢ Close , deadend
c19 , c16 ,     , {M} , T , s3 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c20 , c19 ,     , ∅ , F , s3 ⊢ ⊥ , deadend
c21 , c19 ,     , {M} , T , s3 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c22 , c21 ,     , {M} , T , s4 ⊢ A(⊥ R (Close ∨ ¬Heat))
c23 , c22 ,     , {M} , T , s4 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c24 , c23 ,     , {M} , T , s4 ⊢ (Close ∨ ¬Heat) , shortcut
c25 , c24 ,     , {M} , T , s4 ⊢ Close , deadend
c26 , c23 ,     , {M} , T , s4 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c27 , c26 ,     , ∅ , F , s4 ⊢ ⊥ , deadend
c28 , c26 ,     , {M} , T , s4 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c29 , c28 , c35 , {M} , T , s2 ⊢ A(⊥ R (Close ∨ ¬Heat))
c30 , c29 ,     , {M} , T , s2 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c31 , c30 ,     , {M} , T , s2 ⊢ (Close ∨ ¬Heat) , shortcut
c32 , c31 ,     , {M} , T , s2 ⊢ Close , deadend
c33 , c30 ,     , null , U , s2 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
```

```
c34 , c33 ,     , ∅ , F , s2 ⊢ ⊥ , deadend
c35 , c33 , VVV , null , U , s2 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c36 , c7 , c50 , {M} , T , s5 ⊢ A(⊥ R (Close ∨ ¬Heat))
c37 , c36 ,     , {M} , T , s5 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c38 , c37 ,     , {M} , T , s5 ⊢ (Close ∨ ¬Heat)
c39 , c38 ,     , ∅ , F , s5 ⊢ Close , deadend
c40 , c38 ,     , {M} , T , s5 ⊢ ¬Heat , deadend
c41 , c37 ,     , {M} , T , s5 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c42 , c41 ,     , ∅ , F , s5 ⊢ ⊥ , deadend
c43 , c41 ,     , {M} , T , s5 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c44 , c43 ,     , {M} , T , s6 ⊢ A(⊥ R (Close ∨ ¬Heat))
c45 , c44 ,     , {M} , T , s6 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c46 , c45 ,     , {M} , T , s6 ⊢ (Close ∨ ¬Heat) , shortcut
c47 , c46 ,     , {M} , T , s6 ⊢ Close , deadend
c48 , c45 ,     , {M} , T , s6 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c49 , c48 ,     , ∅ , F , s6 ⊢ ⊥ , deadend
c50 , c48 , VVV , {M} , T , s6 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c51 , c50 ,     , {M} , T , s1 ⊢ A(⊥ R (Close ∨ ¬Heat))
c52 , c51 ,     , {M} , T , s1 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c53 , c52 ,     , {M} , T , s1 ⊢ (Close ∨ ¬Heat) , shortcut
c54 , c53 ,     , {M} , T , s1 ⊢ Close , deadend
c55 , c52 ,     , null , U , s1 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c56 , c55 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c57 , c55 , VVV , null , U , s1 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c58 , c57 ,     , {M} , T , s3 ⊢ A(⊥ R (Close ∨ ¬Heat))
c59 , c58 ,     , {M} , T , s3 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c60 , c59 ,     , {M} , T , s3 ⊢ (Close ∨ ¬Heat) , shortcut
c61 , c60 ,     , {M} , T , s3 ⊢ Close , deadend
c62 , c59 ,     , {M} , T , s3 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c63 , c62 ,     , ∅ , F , s3 ⊢ ⊥ , deadend
c64 , c62 ,     , {M} , T , s3 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c65 , c64 ,     , {M} , T , s4 ⊢ A(⊥ R (Close ∨ ¬Heat))
c66 , c65 ,     , {M} , T , s4 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c67 , c66 ,     , {M} , T , s4 ⊢ (Close ∨ ¬Heat) , shortcut
c68 , c67 ,     , {M} , T , s4 ⊢ Close , deadend
c69 , c66 ,     , {M} , T , s4 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c70 , c69 ,     , ∅ , F , s4 ⊢ ⊥ , deadend
c71 , c69 ,     , {M} , T , s4 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
c72 , c71 , c78 , {M} , T , s2 ⊢ A(⊥ R (Close ∨ ¬Heat))
c73 , c72 ,     , {M} , T , s2 ⊢ (((Close ∨ ¬Heat)) ∧ ((⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat
    ))))))
c74 , c73 ,     , {M} , T , s2 ⊢ (Close ∨ ¬Heat) , shortcut
c75 , c74 ,     , {M} , T , s2 ⊢ Close , deadend
c76 , c73 ,     , null , U , s2 ⊢ (⊥ ∨ AX(A(⊥ R (Close ∨ ¬Heat))))
c77 , c76 ,     , ∅ , F , s2 ⊢ ⊥ , deadend
c78 , c76 , VVV , null , U , s2 ⊢ AX(A(⊥ R (Close ∨ ¬Heat)))
****************************************
Configurations: 79  , Thereof Deadends: 24
Junctions: 39       , Junction Short Cuts used: 9
Fixpoint Loops: 5   , Calculations within Fixpoint Loops: 50
MPS generated: 51   , Contraction Operations: 0
Recursion Calls: 88 , Max Recursion Depth: 29
****************************************
********** Model Checking Result:
c0 ,    , c14 , {M} , T , s0 ⊢ A(⊥ R (Close ∨ ¬Heat))
****************************************
```

Listing A.20: Logfile Microwave Oven Scenario 2

```
********** Runcontrol:
kmtsfile: KMTS_Microwave.json
ctlfile : CTL_Microwave2.json
opti    : 1
rc      : 1
rt      : 0
prepro  : 1
fploop  : 0
arena   : 1
stats   : 1
****************************************
Input CTL formula: AG(Start → AF(Heat))
********** Input Data:
Input initial state: s0
Input KMTS.AP: Start, Close, Heat, Error,
Contraction Model Checking Form: A(⊥ R (¬Start ∨ A(⊤ U Heat)))
Negation Normal Form: A(⊥ R (¬Start ∨ A(⊤ U Heat)))
****************************************
********** Coloured Arena:
configuration, parent, fploop, delta, colour, state ⊢ formula, info
c0 ,    , c13 , ∅ , F , s0 ⊢ A(⊥ R (¬Start ∨ A(⊤ U Heat)))
c1 , c0 ,     , ∅ , F , s0 ⊢ (((¬Start ∨ A(⊤ U Heat))) ∧ ((⊥ ∨ AX(A(⊥ R (¬Start ∨ A
    (⊤ U Heat)))))))
c2 , c1 ,     , {M} , T , s0 ⊢ (¬Start ∨ A(⊤ U Heat)) , shortcut
c3 , c2 ,     , {M} , T , s0 ⊢ ¬Start , deadend
c4 , c1 ,     , ∅ , F , s0 ⊢ (⊥ ∨ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat)))))
c5 , c4 ,     , ∅ , F , s0 ⊢ ⊥ , deadend
c6 , c4 ,     , ∅ , F , s0 ⊢ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat))))
c7 , c6 , c43 , ∅ , F , s1 ⊢ A(⊥ R (¬Start ∨ A(⊤ U Heat)))
c8 , c7 ,     , ∅ , F , s1 ⊢ (((¬Start ∨ A(⊤ U Heat))) ∧ ((⊥ ∨ AX(A(⊥ R (¬Start ∨ A
    (⊤ U Heat)))))))
c9 , c8 ,     , {M} , T , s1 ⊢ (¬Start ∨ A(⊤ U Heat)) , shortcut
c10 , c9 ,     , {M} , T , s1 ⊢ ¬Start , deadend
c11 , c8 ,     , ∅ , F , s1 ⊢ (⊥ ∨ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat)))))
c12 , c11 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c13 , c11 , VVV , ∅ , F , s1 ⊢ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat))))
c14 , c13 ,     , {M} , T , s3 ⊢ A(⊥ R (¬Start ∨ A(⊤ U Heat)))
c15 , c14 ,     , {M} , T , s3 ⊢ (((¬Start ∨ A(⊤ U Heat))) ∧ ((⊥ ∨ AX(A(⊥ R (¬Start
     ∨ A(⊤ U Heat)))))))
c16 , c15 ,     , {M} , T , s3 ⊢ (¬Start ∨ A(⊤ U Heat))
c17 , c16 ,     , ∅ , F , s3 ⊢ ¬Start , deadend
c18 , c16 ,     , {M} , T , s3 ⊢ A(⊤ U Heat)
c19 , c18 ,     , {M} , T , s3 ⊢ (Heat ∨ ((⊤ ∧ AX(A(⊤ U Heat)))))
c20 , c19 ,     , ∅ , F , s3 ⊢ Heat , deadend
c21 , c19 ,     , {M} , T , s3 ⊢ (⊤ ∧ AX(A(⊤ U Heat)))
c22 , c21 ,     , {M} , T , s3 ⊢ ⊤ , deadend
c23 , c21 ,     , {M} , T , s3 ⊢ AX(A(⊤ U Heat))
c24 , c23 , c32 , {M} , T , s4 ⊢ A(⊤ U Heat)
c25 , c24 ,     , {M} , T , s4 ⊢ (Heat ∨ ((⊤ ∧ AX(A(⊤ U Heat))))) , shortcut
c26 , c25 ,     , {M} , T , s4 ⊢ Heat , deadend
c27 , c15 ,     , {M} , T , s3 ⊢ (⊥ ∨ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat)))))
c28 , c27 ,     , ∅ , F , s3 ⊢ ⊥ , deadend
c29 , c27 ,     , {M} , T , s3 ⊢ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat))))
c30 , c29 ,     , {M} , T , s4 ⊢ A(⊥ R (¬Start ∨ A(⊤ U Heat)))
c31 , c30 ,     , {M} , T , s4 ⊢ (((¬Start ∨ A(⊤ U Heat))) ∧ ((⊥ ∨ AX(A(⊥ R (¬Start
     ∨ A(⊤ U Heat)))))))
c32 , c31 , ___ , {M} , T , s4 ⊢ (¬Start ∨ A(⊤ U Heat))
c33 , c32 ,     , ∅ , F , s4 ⊢ ¬Start , deadend
c34 , c31 ,     , {M} , T , s4 ⊢ (⊥ ∨ AX(A(⊥ R (¬Start ∨ A(⊤ U Heat)))))
```

```
c35 , c34 ,     , ∅ , F , s4 ⊢ ⊥ , deadend
c36 , c34 ,     , {M} , T , s4 ⊢ AX(A(⊥ R (¬Start ∨ A(T U Heat))))
c37 , c36 , c43 , {M} , T , s2 ⊢ A(⊥ R (¬Start ∨ A(T U Heat)))
c38 , c37 ,     , {M} , T , s2 ⊢ (((¬Start ∨ A(T U Heat))) ∧ ((⊥ ∨ AX(A(⊥ R (¬Start
     ∨ A(T U Heat)))))))
c39 , c38 ,     , {M} , T , s2 ⊢ (¬Start ∨ A(T U Heat)) , shortcut
c40 , c39 ,     , {M} , T , s2 ⊢ ¬Start , deadend
c41 , c38 ,     , null , U , s2 ⊢ (⊥ ∨ AX(A(⊥ R (¬Start ∨ A(T U Heat)))))
c42 , c41 ,     , ∅ , F , s2 ⊢ ⊥ , deadend
c43 , c41 , VVV , null , U , s2 ⊢ AX(A(⊥ R (¬Start ∨ A(T U Heat))))
c44 , c6 ,     , ∅ , F , s5 ⊢ A(⊥ R (¬Start ∨ A(T U Heat)))
c45 , c44 ,     , ∅ , F , s5 ⊢ (((¬Start ∨ A(T U Heat))) ∧ ((⊥ ∨ AX(A(⊥ R (¬Start ∨
     A(T U Heat)))))) , shortcut
c46 , c45 ,     , ∅ , F , s5 ⊢ (¬Start ∨ A(T U Heat))
c47 , c46 ,     , ∅ , F , s5 ⊢ ¬Start , deadend
c48 , c46 , c59 , ∅ , F , s5 ⊢ A(T U Heat)
c49 , c48 ,     , ∅ , F , s5 ⊢ (Heat ∨ ((T ∧ AX(A(T U Heat)))))
c50 , c49 ,     , ∅ , F , s5 ⊢ Heat , deadend
c51 , c49 ,     , ∅ , F , s5 ⊢ (T ∧ AX(A(T U Heat)))
c52 , c51 ,     , {M} , T , s5 ⊢ T , deadend
c53 , c51 ,     , ∅ , F , s5 ⊢ AX(A(T U Heat))
c54 , c53 ,     , ∅ , F , s6 ⊢ A(T U Heat)
c55 , c54 ,     , ∅ , F , s6 ⊢ (Heat ∨ ((T ∧ AX(A(T U Heat)))))
c56 , c55 ,     , ∅ , F , s6 ⊢ Heat , deadend
c57 , c55 ,     , ∅ , F , s6 ⊢ (T ∧ AX(A(T U Heat)))
c58 , c57 ,     , {M} , T , s6 ⊢ T , deadend
c59 , c57 , VVV , ∅ , F , s6 ⊢ AX(A(T U Heat))
c60 , c59 , c71 , ∅ , F , s1 ⊢ A(T U Heat)
c61 , c60 ,     , null , U , s1 ⊢ (Heat ∨ ((T ∧ AX(A(T U Heat)))))
c62 , c61 ,     , ∅ , F , s1 ⊢ Heat , deadend
c63 , c61 ,     , null , U , s1 ⊢ (T ∧ AX(A(T U Heat)))
c64 , c63 ,     , {M} , T , s1 ⊢ T , deadend
c65 , c63 ,     , null , U , s1 ⊢ AX(A(T U Heat))
c66 , c65 ,     , null , U , s0 ⊢ A(T U Heat)
c67 , c66 ,     , null , U , s0 ⊢ (Heat ∨ ((T ∧ AX(A(T U Heat)))))
c68 , c67 ,     , ∅ , F , s0 ⊢ Heat , deadend
c69 , c67 ,     , null , U , s0 ⊢ (T ∧ AX(A(T U Heat)))
c70 , c69 ,     , {M} , T , s0 ⊢ T , deadend
c71 , c69 , VVV , null , U , s0 ⊢ AX(A(T U Heat))
c72 , c65 ,     , {M} , T , s3 ⊢ A(T U Heat)
c73 , c72 ,     , {M} , T , s3 ⊢ (Heat ∨ ((T ∧ AX(A(T U Heat)))))
c74 , c73 ,     , ∅ , F , s3 ⊢ Heat , deadend
c75 , c73 ,     , {M} , T , s3 ⊢ (T ∧ AX(A(T U Heat)))
c76 , c75 ,     , {M} , T , s3 ⊢ T , deadend
c77 , c75 ,     , {M} , T , s3 ⊢ AX(A(T U Heat))
c78 , c77 ,     , {M} , T , s4 ⊢ A(T U Heat)
c79 , c78 ,     , {M} , T , s4 ⊢ (Heat ∨ ((T ∧ AX(A(T U Heat))))) , shortcut
c80 , c79 ,     , {M} , T , s4 ⊢ Heat , deadend
*****************************************
Configurations: 81  , Thereof Deadends: 25
Junctions: 37       , Junction Short Cuts used: 6
Fixpoint Loops: 6   , Calculations within Fixpoint Loops: 129
MPS generated: 88   , Contraction Operations: 0
Recursion Calls: 89 , Max Recursion Depth: 26
*****************************************
********** Model Checking Result:
c0 ,    , c13 , ∅ , F , s0 ⊢ A(⊥ R (¬Start ∨ A(T U Heat)))
*****************************************
```

A.4 Logfiles

These are logfiles generated by experimental execution of model checking use cases from Section 6.4 (Literature Examples).

Listing A.21: Logfile use case 6.4.1 (a)

```
********** Runcontrol:
kmtsfile: KMTS_RA15.json
ctlfile : CTL_RA15.json
opti    : 1
prepro  : 1
fploop  : 1
arena   : 1
stats   : 1
****************************************
Input CTL formula: EX(m) ∨ E(m U ¬m)
********** Input Data:
Input initial state: s0
Input KMTS.AP: m,
Contraction Model Checking Form: EX(m) ∨ E(m U ¬m)
Negation Normal Form: EX(m) ∨ E(m U ¬m)
****************************************
time since start program - 1 - before createArena
time since start program - 1 - 0  configurations created
c14 , c12 , ___ , null , U , s1 ⊢ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c12 , c10 ,     , null , U , s1 ⊢ m ∧ EX(E(m U ¬m))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c10 , c9 ,     , null , U , s1 ⊢ ¬m ∨ (m ∧ EX(E(m U ¬m)))
c9 , c8 , c14 , ∅ , U , s1 ⊢ E(m U ¬m)
********** Loop starting: c9 , c14
c14 , c12 , ___ , ∅ , F , s1 ⊢ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c12 , c10 ,     , ∅ , F , s1 ⊢ m ∧ EX(E(m U ¬m))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c10 , c9 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m ∨ (m ∧ EX(E(m U ¬m)))
c9 , c8 , c14 , {M({P3(s1,¬m)})} , I , s1 ⊢ E(m U ¬m)
********** Loop starting: c9 , c14
c14 , c12 , ___ , {M({P3(s1,¬m)})} , I , s1 ⊢ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c12 , c10 ,     , ∅ , F , s1 ⊢ m ∧ EX(E(m U ¬m))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c10 , c9 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m ∨ (m ∧ EX(E(m U ¬m)))
c9 , c8 , c14 , {M({P3(s1,¬m)})} , I , s1 ⊢ E(m U ¬m)
********** Loop ended:
****************************************
********** Coloured Arena:
configuration, parent, fploop, delta, colour, state ⊢ formula, info
c0 ,    ,     , {M} , T , s0 ⊢ EX(m) ∨ E(m U ¬m)
c1 , c0 ,     , {M({P3(s1,m)})} , I , s0 ⊢ EX(m)
c2 , c1 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c3 , c0 ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ E(m U ¬m)
c4 , c3 ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ ¬m ∨ (m ∧ EX(E(m U ¬m)))
c5 , c4 ,     , ∅ , F , s0 ⊢ ¬m , deadend
c6 , c4 ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ m ∧ EX(E(m U ¬m))
c7 , c6 ,     , {M} , T , s0 ⊢ m , deadend
c8 , c6 ,     , {M({P3(s1,¬m)})} , I , s0 ⊢ EX(E(m U ¬m))
c9 , c8 , c14 , {M({P3(s1,¬m)})} , I , s1 ⊢ E(m U ¬m)
c10 , c9 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m ∨ (m ∧ EX(E(m U ¬m)))
c11 , c10 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
```

```
c12 , c10 ,     , ∅ , F , s1 ⊢ m ∧ EX(E(m U ¬m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c14 , c12 , ___ , {M({P3(s1,¬m)})} , I , s1 ⊢ EX(E(m U ¬m))
*****************************************
Configurations: 15  , Thereof Deadends: 5
Junctions: 5       , Junction Short Cuts used: 0
Fixpoint Loops: 1   , Calculations within Fixpoint Loops: 8
MPS generated: 12   , Contraction Operations: 1
Recursion Calls: 16 , Max Recursion Depth: 10
*****************************************
********** Model Checking Result:
c0 ,    ,     , {M} , T , s0 ⊢ EX(m) ∨ E(m U ¬m)
*****************************************
time since start program - 9 - end program
```

Listing A.22: Logfile use case 6.4.1 (b)

```
********** Runcontrol:
kmtsfile: KMTS_GAW13.json
ctlfile : CTL_GAW13.json
opti    : 1
prepro  : 1
fploop  : 1
arena   : 1
stats   : 1
*****************************************
Input CTL formula: AX(AG(¬m) ∨ AF(m))
********** Input Data:
Input initial state: s0
Input KMTS.AP: m,
Contraction Model Checking Form: AX(A(⊥ R ¬m) ∨ A(⊤ U m))
Negation Normal Form: AX(A(⊥ R ¬m) ∨ A(⊤ U m))
*****************************************
time since start program - 4 - before createArena
time since start program - 4 - 0  configurations created
c7 , c5 , ___ , null , U , s1 ⊢ AX(A(⊥ R ¬m))
c6 , c5 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c5 , c3 ,     , null , U , s1 ⊢ ⊥ ∨ AX(A(⊥ R ¬m))
c4 , c3 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c3 , c2 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m ∧ (⊥ ∨ AX(A(⊥ R ¬m)))
c2 , c1 , c7 , {M({P3(s1,¬m)})} , I , s1 ⊢ A(⊥ R ¬m)
********** Loop starting: c2 , c7
c7 , c5 , ___ , {M({P1(s1,s0),P3(s1,¬m)}),M({P1(s1,s0),P1(s1,s1),P3(s1,m)})} , I ,
    s1 ⊢ AX(A(⊥ R ¬m))
c6 , c5 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c5 , c3 ,     , {M({P1(s1,s0),P3(s1,¬m)}),M({P1(s1,s0),P1(s1,s1),P3(s1,m)})} , I ,
    s1 ⊢ ⊥ ∨ AX(A(⊥ R ¬m))
c4 , c3 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c3 , c2 ,     , {M({P1(s1,s0),P3(s1,¬m)})} , I , s1 ⊢ ¬m ∧ (⊥ ∨ AX(A(⊥ R ¬m)))
c2 , c1 , c7 , {M({P1(s1,s0),P3(s1,¬m)})} , I , s1 ⊢ A(⊥ R ¬m)
********** Loop starting: c2 , c7
c7 , c5 , ___ , {M({P1(s1,s0),P3(s1,¬m)}),M({P1(s1,s0),P1(s1,s1),P3(s1,m)})} , I ,
    s1 ⊢ AX(A(⊥ R ¬m))
c6 , c5 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c5 , c3 ,     , {M({P1(s1,s0),P3(s1,¬m)}),M({P1(s1,s0),P1(s1,s1),P3(s1,m)})} , I ,
    s1 ⊢ ⊥ ∨ AX(A(⊥ R ¬m))
c4 , c3 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c3 , c2 ,     , {M({P1(s1,s0),P3(s1,¬m)})} , I , s1 ⊢ ¬m ∧ (⊥ ∨ AX(A(⊥ R ¬m)))
c2 , c1 , c7 , {M({P1(s1,s0),P3(s1,¬m)})} , I , s1 ⊢ A(⊥ R ¬m)
********** Loop ended:
c16 , c14 , ___ , null , U , s1 ⊢ AX(A(⊤ U m))
```

```
c15 , c14 ,     , {M} , T , s1 ⊢ T , deadend
c14 , c12 ,     , null , U , s1 ⊢ T ∧ AX(A(T U m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c12 , c11 ,     , null , U , s1 ⊢ m ∨ (T ∧ AX(A(T U m)))
c11 , c1 , c16 , ∅ , U , s1 ⊢ A(T U m)
********** Loop starting: c11 , c16
c16 , c14 , ___ , {M({P1(s1,s1)})} , I , s1 ⊢ AX(A(T U m))
c15 , c14 ,     , {M} , T , s1 ⊢ T , deadend
c14 , c12 ,     , {M({P1(s1,s1)})} , I , s1 ⊢ T ∧ AX(A(T U m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c12 , c11 ,     , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ m ∨ (T ∧ AX(A
    (T U m)))
c11 , c1 , c16 , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ A(T U m)
********** Loop starting: c11 , c16
c16 , c14 , ___ , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ AX(A(T U m))
c15 , c14 ,     , {M} , T , s1 ⊢ T , deadend
c14 , c12 ,     , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ T ∧ AX(A(T U
    m))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c12 , c11 ,     , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ m ∨ (T ∧ AX(A
    (T U m)))
c11 , c1 , c16 , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ A(T U m)
********** Loop ended:
****************************************
********** Coloured Arena:
configuration, parent, fploop, delta, colour, state ⊢ formula, info
c0 ,    ,     , {M({P1(s1,s0),P3(s1,¬m)}),M({P3(s1,m)})} , I , s0 ⊢ AX(A(⊥ R ¬m) ∨
    A(T U m))
c1 , c0 ,     , {M({P1(s1,s0),P3(s1,¬m)}),M({P3(s1,m)})} , I , s1 ⊢ A(⊥ R ¬m) ∨ A(T
     U m)
c2 , c1 , c7 , {M({P1(s1,s0),P3(s1,¬m)})} , I , s1 ⊢ A(⊥ R ¬m)
c3 , c2 ,     , {M({P1(s1,s0),P3(s1,¬m)})} , I , s1 ⊢ ¬m ∧ (⊥ ∨ AX(A(⊥ R ¬m)))
c4 , c3 ,     , {M({P3(s1,¬m)})} , I , s1 ⊢ ¬m , deadend
c5 , c3 ,     , {M({P1(s1,s0),P3(s1,¬m)}),M({P1(s1,s1),P3(s1,m)})} , I ,
    s1 ⊢ ⊥ ∨ AX(A(⊥ R ¬m))
c6 , c5 ,     , ∅ , F , s1 ⊢ ⊥ , deadend
c7 , c5 , ___ , {M({P1(s1,s0),P3(s1,¬m)}),M({P1(s1,s0),P1(s1,s1),P3(s1,m)})} , I ,
    s1 ⊢ AX(A(⊥ R ¬m))
c8 , c7 ,     , ∅ , F , s0 ⊢ A(⊥ R ¬m) , may
c9 , c8 ,     , ∅ , F , s0 ⊢ ¬m ∧ (⊥ ∨ AX(A(⊥ R ¬m))) , shortcut
c10 , c9 ,     , ∅ , F , s0 ⊢ ¬m , deadend
c11 , c1 , c16 , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ A(T U m)
c12 , c11 ,     , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ m ∨ (T ∧ AX(A
    (T U m)))
c13 , c12 ,     , {M({P3(s1,m)})} , I , s1 ⊢ m , deadend
c14 , c12 ,     , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ T ∧ AX(A(T U
    m))
c15 , c14 ,     , {M} , T , s1 ⊢ T , deadend
c16 , c14 , ___ , {M({P3(s1,m)}),M({P1(s1,s1),P3(s1,¬m)})} , I , s1 ⊢ AX(A(T U m))
c17 , c16 ,     , {M} , T , s0 ⊢ A(T U m) , may
c18 , c17 ,     , {M} , T , s0 ⊢ m ∨ (T ∧ AX(A(T U m))) , shortcut
c19 , c18 ,     , {M} , T , s0 ⊢ m , deadend
****************************************
Configurations: 20  , Thereof Deadends: 6
Junctions: 9       , Junction Short Cuts used: 2
Fixpoint Loops: 2   , Calculations within Fixpoint Loops: 16
MPS generated: 19   , Contraction Operations: 13
Recursion Calls: 22 , Max Recursion Depth: 9
****************************************
********** Model Checking Result:
c0 ,    ,     , {M({P1(s1,s0),P3(s1,¬m)}),M({P3(s1,m)})} , I , s0 ⊢ AX(A(⊥ R ¬m) ∨
    A(T U m))
```

```
****************************************
time since start program - 15 - before determineFailureWitnesses
********** Failure Witnesses:
from:state , to:state , rule , must
c16:s1 , c11:s1 , 1 , false
c12:s1 , c13:s1 , 5 , false
c7:s1 , c8:s0 , 1 , false
c7:s1 , c2:s1 , 1 , false
c3:s1 , c4:s1 , 4 , false
****************************************
time since start program - 16 - end program
```